BREAKING THE MEMORY WALL FOR HIGHLY MULTI-THREADED CORES

THIS DISSERTATION

IS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

Jiayuan Meng

Aug. 10, 2010

Approvals:

Kevin Skadron
(Advisor)

John Lach

Kim Hazelwood

Jason Lawrence

Nathan Binkert

Accepted by the School of Engineering and Applied Science:

James H. Aylor (Dean)

Aug. 10, 2010

Abstract

Emerging applications such as scientific computation, media processing, machine learning and data mining are commonly computation- and data- intensive [1], and they usually exhibit abundant parallelism. These applications motivate the design of throughput-oriented many- and multi-core architectures that employ many small and simple cores and scale up to large thread counts. The cores themselves are also typically multi-threaded. Such organizations are sometimes referred to as Chip Multi-Threading (CMT). However, with tens or hundreds of concurrent threads running on a single chip, throughput is not limited by the computation resources, but by the overhead in data movement. In fact, adding more cores or threads is likely to harm performance due to contention in the memory system.

It is therefore important to improve data management to either reduce or tolerate data movement and associated latencies. Several techniques have been proposed for conventional multi-core organizations. However, the large thread count per core in CMT poses new challenges: much lower cache capacity per thread, nonuniform overhead in thread communication, inability of aggressive out-of-order execution to hide latency, and additional memory latency caused by SIMD constraints. To address the above challenges, we propose several techniques. Some reduce contention in either private caches or shared caches; some identify the right amount of computation to replicate to reduce communication, and some reconfigure SIMD architectures at runtime. Our objective is to allow CMTs

to achieve scalable performance along with the thread count, despite limited energy budget for their memory system.

Acknowledgments

I am fortunate to have Prof. Kevin Skadron as my Ph.D. advisor. His enlightening guidance and thorough training has led me through my Ph.D. study. But that is not all. He has also been a great high school English teacher and a great mentor — at the time that I was admitted to graduate school, I never thought I would devote my research to computer architecture and parallel computing. I have learned much from him, not only about how to research, but also about presenting ideas, debating with logic, collaborating with others, as well as time management. I am grateful about his tremendous support for my personal life, which gave time to visit my family and balance life with work. His patience has given me time to seek ideas without being blamed for nonproductivity. I am also deeply thankful for his continuous encouragement along the way, and his generous guidance on my career development.

I would also like to thank Dr. Dee A. B. Weikle, who has co-advised me during my first two years of graduate study. She has been a great mentor on scientific research and analysis, and her kindness helped me settle down at UVA and start my research. She has been immensely supportive.

I am indebted to the LAVA group and my outstanding colleagues for their tremendous help and enlightening discussions. They have generously shared their opinions and supported me greatly in both research and technical writing. They have also made my life at UVA more fun. Especially, I want to thank Jeremy W. Sheaffer, Karthik Sankaranarayanan,

Yingmin Li, David Tarjan, Shuai Che, Michael Boyer, Mario D. Marino, Lukasz Szafaryn, and Greg Faust.

I would like to thank my Ph.D. committee, Dr. Nathan Binkert, Prof. Kim Hazelwood, Prof. John Lach, and Prof. Jason Lawrence for their valuable comments. I also enjoy exchanging thoughts with Prof. Sudhanva Gurumurthi, Prof. Marty Humphrey, Prof. Andrew Grimshaw, and Prof. Lieven Eeckhout. Their insightful suggestions have helped me clear many of my doubts.

I am also deeply thankful to my supervisor Dr. Srimat Chakradhar, Prof. Anand Raghunathan, and my colleague Dr. Surendra Byna during my internship at NEC Laboratories America. I have learned a great deal from them, and I enjoyed the time working together with them. I also appreciate their tremendous support for my career development.

I have also made many good friends at University of Virginia, too many to name here, but in particular I want to mention Rui Wang, Lingjia Tang, Yafeng Wu, Wei Le, Jiakang Lu, Jie Li, Jiawei Huang, Jason Mars, and Jiajun Zhu, all of whom provided valuable friendship and support. I also want to thank Kenneth Hoste from Ghent University, who has been a good friend and collaborator.

Moreover, to the staff members and computing facilities committee, I offer thanks for tolerating me and my hungry simulations: Andrew Grimshaw, Scott Ruffner, Essex Scales, Jessica Greer, and Rick Stillings. I cannot count how many times I have crashed the department's machines, and if it were not for their timely and skillful support, I would have halted the department. I would also like to thank their patience in teaching me how to use the resources in a right manner. I would also like to thank Brenda Perkins and Wendy Morris for their countless help. In addition, much of my work depends on the M5 simulator, and I thank the M5 team for their time and effort in answering my many questions, good or bad.

Last but not least, I would like to thank my family. Their unconditional love has been

granting me the strength and momentum to pursue excellence. Without the encouragement and inspiration from my fiancée, Wa Yuan, I would not have been able to complete my study. I also greatly appreciate the understanding and support from my parents, who always back me up they are on the other side of the globe. I dedicate this dissertation to all of them.

This work was supported in part by SRC grant no. 1607, NSF grant nos. IIS-0612049 and CNS-0615277, a grant from Intel Research, a professor partnership award from NVIDIA Research, and an NVIDIA Ph.D. fellowship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction to Chip Multi-Threading

The era of instruction level parallelism (ILP) and single processor performance scaling has ended: power constraints prevent clock cycles from increasing, and the point of diminishing return has reached for adding more transistors to explore ILP. We are now in the era of *Chip Multiprocessor* (CMP), where each chip typically consists of a number of simple cores on a single chip. Compared to single processors, CMPs generally have higher throughput since they better exploit data parallelism and task parallelism, both exist extensively in many emerging applications. Examples of such application domains include scientific computation, media processing, machine learning and data mining [1].

However, having multiple cores on a single chip leads to more communication and data movement, which in turn demands higher memory bandwidth. Even worse, the memory speed doubles only every six years, whereas the processor speed historically doubles every two years [2]. This inevitably leads to the *memory wall* where CMP pipelines often stall waiting for memory accesses.

To increase throughput in spite of the increasing gap between processor speed and memory speed, each core in a CMP architecture can have multiple threads; this type of architecture is often named *Chip Multi-Threading* (CMT). Comparing to CMPs with single threaded cores, a CMT architecture has several advantages. First, CMTs allow threads with overlapping working sets to share the same cache, so that the on-chip storage can be utilized better. Second, CMTs increase pipeline utilization by overlapping the outgoing memory access of one thread with the computation of another thread on the same core. Third, by having more threads per storage, CMTs can exploit more data and task parallelism within the same area budget. Figure 1.1 summarizes the trend of shifting from single processors to CMT.

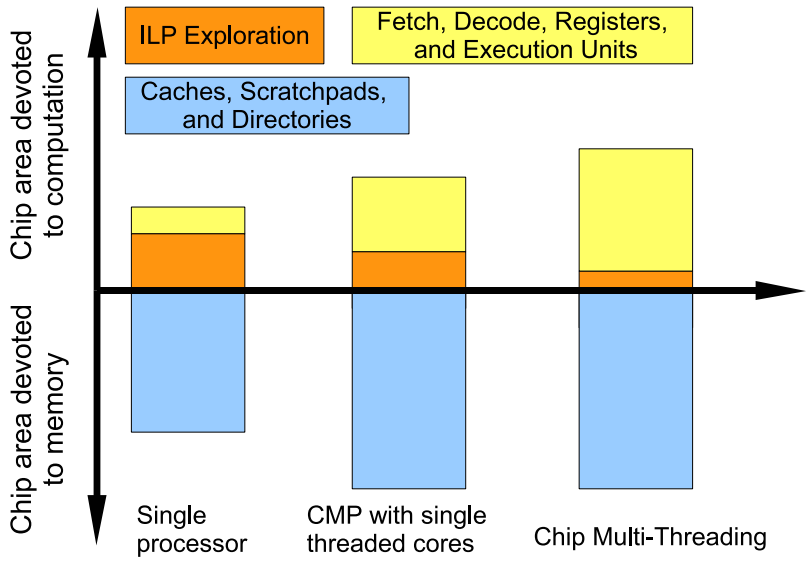


Figure 1.1: Trends for area breakdown of a chip. CMP and CMT devotes less area for ILP exploration and more area for register files and execute units to improve throughput. CMT multiply the number of threads per core to further increase throughput without significantly increasing storage area.

CMT commonly comes in the forms of Simultaneous Multi-Threading (SMT) [3] and Single Instruction Multiple Data (SIMD) [4]. SMT can fetch instructions from multiple

independent threads in one cycle; each instruction can be fed to the appropriate execute unit if available. This increases the efficiency of the pipeline and improves utilization of execute units. Examples of SMT include Intel's Pentium IV [5], IBM's POWER 5 [6], and Sun's UltraSPARC T1 [7].

With SIMD architectures, a group of threads share the same instruction sequencer and they operate in lockstep. SIMD organizations amortize the area and power of fetch, decode, and issue logic across multiple processing units in order to maximize throughput for a given area and power budget. SIMD lockstep operation of multiple datapaths can be implemented with vector units, where the SIMD operation is explicit in the instruction set and a single thread operates on wide vector registers. SIMD lockstep can also be *implicit*, where each lane executes distinct threads that operate on scalar registers, but the threads progress in lockstep. The latter is also referred to by NVIDIA as *single instruction multiple threads* (SIMT). Examples of SIMD include Sun's Niagara [8], the Cell Broadband Engine (CBE) [9], Clearspeed's CSX600 [10], Cray [11], Tarantula [12], NVIDIA's Tesla [13] and Fermi [14], and Intel's Larrabee [15]. Academic researchers have also been exploring this design space and several prototypes have been produced, including Stanford's Imagine [16] or Merrimac [17].

CMT architectures—especially SIMD architectures—drastically increase the thread count per core. Therefore, they often subject to more intensive data movement and communication. We study the memory wall in the context of CMT in more details below.

1.2 The Memory Wall for Chip Multi-Threading

CMT architectures come with various types of memory system organizations. *Cache-based* memory systems provide a hierarchy of caches that are implicitly managed by a coherent protocol. Data is automatically fetched into caches or spilled into the memory according to

locality heuristics. On the other hand, *stream-based* memory systems require programmers to explicitly move data and specify which data to be stored closer to the execute units. The additional programming effort enables application specific optimizations.

Despite the variety of memory systems, we have been consistently observing that the number of threads per L1 storage has been increasing dramatically for CMT: Power 5 has two threads sharing a 32 KB D-cache; Niagara runs four threads over an 8 KB D-cache; and NVIDIA's GTX280 [18] can operate up to 32 warps each have 32 threads over a 16 KB scratchpad (officially referred to as the per-block shared memory (PBSM))! This reflects the fact that given a fixed area budget, more parallelism can be exploited by increasing the threads per storage ratio. However, the system has to be designed in a manageable and balanced way; the on-chip storage needs to be reduced to make room for more threads, but not to such an extent that the memory system easily gets thrashed (*i.e.*, frequent data movement to and from the memory resulted from severe contention). In other words, the more efficient data is managed, the more hardware threads can be accommodated on a single chip and the more throughput the architecture can produce.

Evidence has already shown that parallel architectures have already hit the memory wall: as Mckee et al. conducted [19] in 2004, commercial applications such as transaction processing workloads show 65% percent node idle times; scientific applications show 95% node idle times; much of this is due to memory bottlenecks. Therefore, efficient data management is critical for CMT to achieve better performance.

1.3 Challenges in CMT Data Management

The unique properties of CMT poses several new challenges for data management. First, not only the shared storage, but also the per-core private storage is contended for by multiple threads. The aggregate working set of all active threads on the same core has to fit

in the L1 storage, otherwise the L1 storage may be overwhelmed and performance risks severe degradation. As a result, multi-threaded cores benefit from extensive data sharing among threads on the same core.

Second, communication latency among threads is no longer uniform; the overhead is much longer among threads on different cores than those on the same core. While threads on the same core can communicate by writing and reading from the per core L1 storage, data communication between threads on different cores has to go through the shared storage, and it may require several coherence messages. In other words, the set of all active threads can no longer be viewed as a group of homogeneous, independent entities; instead, they demonstrate heterogeneity and appropriately collocating threads leads to significantly less data communication between cores.

Third, highly multi-threaded cores are often in-order and they lack the ability to execute beyond pending memory instructions to hide memory latency. A conventional single processor devotes significant area to out-of-order execution and ILP exploration with the purpose of shortening the *latency of an individual thread*. It often comes with additional units such as rename registers, scoreboards, re-order buffer (ROB), and load store queues. On the other hand, multi-threaded cores aim at *leveraging the overall throughput* and therefore they devote more area for hardware thread contexts—such as the register files—and less for ILP exploration. However, little ILP exploration limits the ability hide memory access latency. In many CMT architectures, limited ILP exploration is compensated by *multi-threading*, the ability to switch to other threads when the current thread experiences long latency memory accesses. However, the effectiveness of multi-threading depends on various runtime dynamics and has not be well studied. We will discuss it in more detail in Chapter 6 and 7.

Finally, SIMD enforces threads to operate in lockstep, which may incur additional overhead when some threads have to wait for their peers to be ready. This often occurs in two

scenarios. First, upon *branch divergence*, SIMD threads take different paths after a conditional branch. SIMD organizations have to execute one path of the branch at a time, with threads falling into the alternate path suspended. Second, upon *memory latency divergence*, SIMD threads experience different memory-reference latencies caused by cache misses or accessing different DRAM banks. Threads that operate in SIMD must wait until the last thread has its reference satisfied. In Chapter 6, we propose several ways to address the above issues.

1.4 Data Management for Multi-Threaded Cores

Data management has been extensively studied for decades. We list general principles in conventional data management that have applied well to single processors and CMPs with single-threaded cores. We then discuss the new problems and opportunities posed by multi-threaded cores.

1.4.1 Background

The general ways to cope with limited bandwidth and storage capacity are to either reduce or tolerate data movement and associated latencies. Such data management can occur in software (*i.e.* applications), middleware(*i.e.* compilers and run-time systems), and hardware (*i.e.* memory systems, interconnects, scheduling units, and the computation pipeline).

Data movement can be reduced by:

- Algorithmically modifying target applications to reduce data transfer. There are also application-specific methods to either organize data or restructure the computation so that data movement is minimized [20]. These techniques are not the focus of this dissertation.

- Wisely distributing data using compilers or run-time systems to increase data reuse and reduce data communication. Researchers have proposed various techniques to partition the data set in a way that maximizes spatial locality [21, 22, 23, 24, 25, 26, 27, 28, 29, 30]. There are also several techniques that attempt to map an individual thread closer to the data to be consumed [31, 32, 33, 34, 35, 36, 37, 38, 39]. Moreover, researchers have designed techniques to reduce contention of the shared cache by wisely selecting concurrent threads [40, 41, 42, 43] or partitioning the cache [44, 45, 46].

- Employing efficient data movement protocols or interconnects in the hardware. Architects have explored various cache organizations [47, 48], data replacement and communication strategies [49, 50, 51, 52], cache inclusion properties [53, 54], and various scalable coherence protocols [55, 56].

- Designing efficient storages to increase the amount of data they can accommodate. For example, caches can be compressed [57], and data from multiple cache lines can be collapsed into one line [58].

Latency in data movement can be tolerated with:

- Asynchronous data transfer such as direct memory-access (DMA). The DMA engine can read and write memory independently of the processing elements. Typically, programs specify the address and size of the data transfer, start the DMA engine, and continue with their execution while data is being transferred in parallel.

- Data prefetching, memory level parallelism, and latency hiding techniques. Researchers have proposed temporal and spatial streaming [59, 60, 61] to leverage the repetitive data access patterns in a CMP.

- Speculatively allowing a thread to run ahead beyond pending memory instructions and roll back if speculation fails [62]. Alternatively, speculative threads [63] can issue a stream of future memory references extracted from the original thread.

While many of the above techniques can still be applied to highly multi-threaded cores straightforwardly, some do not apply well due to the nature of simple cores (*e.g.* speculative execution). This dissertation proposes several solutions to address data distribution, task scheduling, and latency hiding in CMT. Moreover, CMT raises new topics such as the ability to trade abundant computing resources for less communication. The following subsections introduce each contribution.

1.4.2 Uniform Data Distribution in Shared Cache

The typical runtime system allocates data without considering how data distributes into cache sets in the shared cache. Overall, it is rare in single-threaded processors that contention over a few cache sets in the shared cache becomes the performance bottleneck. However, in the realm of multi-threaded cores, we found that even slight non-uniform data distribution caused by the run-time system can be magnified by the large thread count. This leads to severe contention on a few hot cache sets, which eventually becomes the bottleneck for performance scaling.

This phenomenon is particularly acute with stack allocation in common operating systems. Generally, a shared, inclusive cache can improve data sharing among cores; data in all upper level caches is replicated in the shared cache, therefore a different core can often reuse data directly in the shared cache instead of probing private caches of a remote core. However, the inclusion property also means that each cache set in the shared cache is contended for by all threads on the chip, which may reach hundreds or thousands in quantity. Current memory allocators align stack bases on page boundaries, which is accommodated

by only a few cache sets. The conflict misses caused by such nonuniform allocation is not obvious when the thread count is small, however, it emerges as a source of severe conflict misses when there are a large number of actives threads all trying to allocate stack data.

In Chapter 4, we propose stack-base randomization that eliminates the major source of conflict misses for large numbers of threads. However, when capacity becomes a limitation for the directory or last-level cache, this is not sufficient. We then propose non-inclusive, semi-coherent cache organization (NISC) that allows private data to be noninclusive in the shared cache. Our data-parallel benchmarks show that conventional systems cannot scale beyond 8 cores, while our techniques allow performance to scale to at least 32 cores for most benchmarks. At 8 cores, stack randomization provides a mean speedup of 1.2X, but stack randomization with 32 cores gives a speedup of 2.7X over the best baseline configuration. Comparing to conventional performance with a 2 MB shared cache, our technique achieves similar performance with a 256 KB shared cache, suggesting shared caches may be typically overprovisioned. When very limited Last Level Cache (LLC) resources are available, NISC can further improve system performance by 1.8X. This work was published in [64].

1.4.3 Fine grained, Inter-Thread Data Sharing

Even after shared cache contention is mitigated by uniform data distribution, the first level data caches are still contended for by multiple threads that collocate on the same core. Therefore, the thread schedulers must try to minimize cache contention among threads sharing the same D-cache. While this has been studied for concurrent threads with *disjoint* working sets, the problem has not been addressed for multi-threaded data-parallel workloads in which threads can be scheduled or constructed to improve inter-thread cache sharing. Chapter 3 proposes *symbiotic affinity scheduling (SAS)* algorithm in which work

is first partitioned according to the number of cores (i.e., the number of caches), and these partitions are then subdivided and scheduled among each core's available thread contexts so that threads sharing a core operate on neighboring elements to maximize cache locality.

We demonstrate this concept with a series of data-parallel benchmarks. Simulations achieve an average speedup of 1.69× and 36% energy savings over conventional scheduling techniques that are oblivious to whether threads share a cache. Even compared to an approach that extends oblivious scheduling to ensure that the sum of the threads' working sets fits in the cache, symbiotic affinity scheduling is able to exploit greater temporal locality and provide 30% performance gains on average. Symbiosis also outperforms adaptive contention reduction techniques by 17%. This work was published in [65].

1.4.4 Replicating Computation to Reduce Communication

Iterative stencil loops (ISL) [66] are widely used in image processing, data mining, and physical simulations. ISLs usually operate on multi-dimensional arrays, with each element computed as a function of some neighboring elements. These neighbors comprise the *stencil*. Multiple iterations across the array are usually required to achieve convergence and/or to simulate multiple time steps. Tiling [25, 30] is often used to partition the stencil loops among multiple processing elements (PEs) for parallel execution. It partitions a nested parallel *for* loop into continuous blocks and assign each block to an individual thread. Between two iterations, a tile needs to exchange its bordering data, or *halo*, with tiles on different PEs. Such communication is expensive. In addition, synchronization is often used to signal the completion of halo exchanges. Both communication and synchronization may incur significant overhead on parallel architectures with shared memory. This is especially true in the case of graphics processors (GPUs), which do not preserve the state of the per-core L1 storage across global synchronization. Techniques to enhance locality–including the

techniques described above—fail to capture the coarse-grained locality across iterations. To improve inter-iteration locality and reduce synchronization overheads, ghost zones can be created to replicate stencil operations, reduce communication and synchronization costs, by redundantly computing halos on multiple PEs. However, the selection of the optimal ghost zone size depends on the characteristics of both the architecture and the application.

To automate the process of optimizing the ghost zone size, Chapter 5 establishes a performance model for NVIDIA's Tesla architecture [13]. The performance model can be used to automatically select the ghost zone size that performs best. The modeling is validated by four diverse ISL applications, for which the predicted ghost zone configurations are able to achieve a speedup no less than 95% of the optimal speedup. This work was published in [67].

1.4.5 Overcoming SIMD Constraints for Better Latency Hiding

Even if data locality is extensively exploited, throughput can still be undermined when a set of threads operating in lockstep (a warp) are stalled due to long latency memory accesses. The wider SIMD it is, the more threads will have to wait when a thread misses the cache. Multi-threading can hide latencies by interleaving the execution of multiple warps, but deep multi-threading using many warps dramatically increases the cost of the register files (multi-threading depth $\times$ SIMD width), and increased cache contention can make performance worse. Instead, intra-warp latency hiding should first be exploited. This allows threads that are ready but stalled by SIMD restrictions to use these idle cycles and reduces the need for multi-threading among warps. Chapter 6 introduces *dynamic warp subdivision* (DWS), which allows a single warp to occupy more than one slot in the scheduler without requiring extra register file space. Independent scheduling entities allow divergent branch paths to interleave their execution, and allow threads that hit to run ahead. The result is

improved latency hiding and memory level parallelism (MLP) (*i.e.* more memory accesses that overlap with each other in time).

We evaluate the technique on a coherent cache hierarchy with private L1 caches and a shared L2 cache. With an area overhead of less than 1%, experiments with eight data-parallel benchmarks show our technique improves performance on average by 1.7X. This work was published in [68].

1.4.6 Dynamically Adapted SIMD for Robust Performance

Conventional SIMD organizations embed a fixed SIMD width and multi-threading depth. However, a static organization may not perform well for all applications or when the underlying memory system resizes itself for power saving or error resilience purposes. In fact, our experiments show that the best SIMD width and multi-threading depth depends heavily on the underlying memory system as well as the application.

This challenges designers in identifying the best combination of SIMD width and depth. Several tradeoffs are involved in determining the best combination of several parameters: wider SIMD may enable more throughput, but is also more likely to waste cycles due to divergent branches or memory accesses; deeper multi-threading may overlap more computation with memory accesses, but it is also likely to increase L1 contention due to multiplication of thread count, especially when combined with wide SIMD. Even with dynamic warp subdivision, threads may still incur severe cache contention for some applications.

Chapter 7 proposes *Robust SIMD*, which dynamically optimizes and reconfigures SIMD width and multi-threading depth according to runtime performance feedback. Robust SIMD can also trade wider SIMD for deeper multi-threading by splitting a wider SIMD group into multiple narrower SIMD groups. Compared to the performance generated by running every benchmark on its individually preferred SIMD organization, the *same* Robust SIMD

organization performs similarly—sometimes even better due to phase adaptation—and outperforms the best fixed SIMD organization by 17%. Our experiments also show that even with dynamic warp subdivision (DWS), a fixed SIMD organization that performs best with 32 KB D-caches can only yield less than 50% of the optimal performance when D-cache capacities reduce to 16 KB. By combining Robust SIMD and DWS, the proposed organization achieves 99% of the optimal performance persistently. In fact, it achieves 97% of the performance achievable by a conventional organization with D-caches that double in size. Overall, Robust SIMD's area overhead is less than 1%. This work was submitted to MICRO-43 [69].

1.5 Dissertation Organization

The rest of the dissertation is organized as follows: Chapter 2 presents the simulation tool we built to conduct our research. Chapter 3 introduces a technique to improve inter-thread spatial and temporal locality for multi-threaded cores. Chapter 4 demonstrates the need to distribute data uniformly to reduce shared cache contention. Chapter 5 establishes an analytical performance model to improve the effectiveness of the ghost zone technique which replicates computation to reduce communication. Chapter 6 proposes dynamic warp subdivision for SIMD architecture to address SIMD constraints and better latency hiding. Chapter 7 describes a reconfigurable SIMD organization that yields robust performance across applications and various memory configurations. Finally, Chapter 8 summarizes the lessons learned and proposes potential research extensions.

Chapter 2

Simulating General Purpose Multi-Threaded Cores

2.1 Introduction

There is an increasing demand for commodity computer systems to host modern applications that exhibit diverging characteristics. While many latency-oriented workloads such as compilers (e.g. GCC) and compressors (e.g. bzip2) still prefer sequential execution, many throughput-oriented workloads are transitioning to parallel execution. They include media processing, scientific computing, physics simulation, and data mining. Workloads can also be compute-intensive or data-intensive. Moreover, different types of workloads can execute simultaneously over the same system. Each workload may demonstrate different instruction-level (ILP), data-level (DLP), thread-level (TLP), and memory-level parallelism (MLP), which challenges future architectures to optimize for less execution time and more power- and area- efficiency.

Several architecture designs have been proposed to address the challenges posed by

heterogeneous workloads. In addition to conventional, general purpose processors, special purpose processors have been built for particular application domains (e.g. Graphics Processors (GPUs) [70], ParallAX [71], Anton [72]). However, the physical separation of the processors incurs significant latency in communication and data transfers. The lack of shared resources may also lead to power- and area- inefficiency. These hardware disadvantages, when exposed to the programmers, complicate coding effort and lead to poor productivity. Alternatively, the integration among heterogeneous processing elements can be made more efficient given a general purpose chip multiprocessor (CMP), where multiple cores reside on the same die and operate over a shared memory. General purpose CMP products have been developed by both academic and industrial groups. They include MIT's Raw [73], Compaq's Piranha [74], Sun's Niagara [8], IBM's Power5 [6], Cell [9], and Intel's Larrabee [15]. While most of these architectures present homogeneous cores, recent studies show that heterogeneous multicore architectures provide significant power and performance advantages [75, 76].

The two extremes over the spectrum of heterogeneous cores are out-of-order (OOO) cores and simple, in-order (IO) cores. While OOO cores dedicate more area to leverage ILP, IO cores remain simple and the equivalent area budget can host more cores, leveraging DLP and TLP. OOO cores and IO cores correspond to the two extremes of workloads — latency-critical and throughput-critical workloads, respectively. These two types of cores have been working complimentarily on traditional computer systems with separate CPU and GPU. Since the release of programmable GPUs [77], many researchers have found that many applications that used to execute sequentially on the CPU can actually benefit from parallel execution on the GPU. Recently, NVIDIA has released CUDA [78] that performs both graphics-specific tasks and general purpose computing. As the more general purpose applications benefit from off-loading throughput-oriented computation to GPU cores, merging the latency oriented OOO cores with the throughput oriented IO cores

can lead to more efficient inter-core communication, better area utilization, and lower energy consumption. AMD's Fusion [79], for example, targets integration of CPU and GPU, presenting a heterogeneous multicore design with additional graphics-purposed units. This leads to a large design space: cores may have different number of hardware thread contexts, SIMD (Single Instruction, Multiple Data) width; caches can present different hierarchies, and one cache can be private or shared among multiple cores; the on-chip network (OCN) may exhibit different topologies; and the coherence protocol has several variations as well.

In order to explore this large design space, a conveniently reconfigurable multicore simulator is needed. Despite existing multicore simulators (SMTSIM [80], Sesc [81], Simics [82], SimFlex [83]), none of them provide object-oriented modules that can be easily reconfigured. In addition, the functionality of the modules are largely bound to the physical layouts that connect the units. We build an easily reconfigurable, modularized multicore simulator named MV5 that targets at large-scale architectures with heterogeneous settings. MV5 is based upon M5 [84], an event driven simulator originally designed for a network of processors. It includes a set of different cores including OOO, SMT, IO, SIMD, and cores with several SIMD groups. Caches may have different sizes, associativity, and banking. Both snoop-based and directory-based coherence protocols (MSI and MESI) are available. Moreover, caches can be organized into various hierarchical layouts, and we have experimented with up to three levels of caches. In addition, MV5 provides point-to-point OCNs that include configurable 2-D meshes. The OCN template can be easily extended to accommodate other topologies such as torus and fat-trees as well.

2.2 Summary of Available General Purpose Simulators

A number of simulators have been developed to simulate various architectures. However, they are all limited in their capability to simulate large-scale CMPs with tens or hundreds of

Features	MV5/M5	SimFlex	Sesc	Simics/Gems/Garnet
Full-system Simualtion	√(M5)	√	×	√
System-call Emulation	√(M5)	×	√	×
I/O disk	√ (M5)	×	×	×
Ethernet	√ (M5)	√	×	√
ISA	various (M5)	Sparc	Mips	various
Emulated thread API	√ (MV5)	NA	√	NA
Category	Event-driven (M5)	Cycle-driven	Event-driven	Event- and Cycle-driven
IO core	√ (M5)	√	×	√
Multithreaded core	√ (MV5)	×	×	√
OOO core	√ (M5)	√	√	√
SIMD core	√ (MV5)	×	×	×
Cache Hierarchy	Configurable (M5)	Private L1, Shared L2	Configurable	Configurable
Coherence	Snoop/Directory (MV5)	Snoop/Directory	Snoop	Directory/Snoop
Bus	√ (M5)	√	√	√
2-D Mesh	√ (MV5)	×	√	√
Hypercube	×	×	√	×
hardware thread-management	√ (MV5)	×	×	×
stream-buffer	√ (MV5)	×	×	×

Table 2.1: Comparison with other simulators.

cores connected through a point-to-point OCN with directory-based cache coherence. Two well-known simulators that model single OOO cores and SMT cores are SimpleScalar [85] and SMTSIM [80], respectively. As modern architectures employ chip multiprocessors (CMPs), several other simulators have been released. They include PTLsim[86], Sesc [81], Simics [82], Gems [87], and SimFlex [83]. All of them provide limited types of cores: IO cores are not present on SMTSIM, PTLsim, and Sesc. SIMD cores are not modeled on any of the these simulators. OCN is not provided in PTLsim and SimFlex. Directory-based coherence is not available for PTLsim and Sesc. Simics can work with Gems to combine multiple core modules with a configurable cache hierarchy. More recently, Garnet [88] has been released to support OCN for Gems [87]. Nevertheless, Simics does not support system-call emulation, which is especially useful when a new hardware feature requires additional run-time library or middleware to interact it with the benchmarks. Table 2.1 summarizes the above comparison.

The MV5 simulator aims at exploring the large design space of future heterogeneous

CMPs with integrated CPU and GPU functionality. We pick M5 as our base simulator because of its well-designed modularity, which leads to convenience in both system reconfiguration and module extension. More importantly, M5 provides an integrated infrastructure from cores and caches to I/O devices and network. It supports both full-system simulation and system-call emulation. Finally, M5 provides a clean interface for debugging, checkpointing, fast-forwarding, and statistics. MV5 inherits the integrity, modularity, and flexibility from M5, and it incorporates essential modules that enable simulations for large-scale heterogeneous CMPs.

2.3 The MV5 Simulator

MV5 inherits the modular, configurable simulation environment from M5, which was originally designed to study I/O bandwidth for a network of processors. Modules' detailed functionalities are programmed in C++ and their parameters are encapsulated by high level python classes for easy reconfiguration. MV5 adds to M5 with several components which are necessary for modeling multicore architectures. MV5 also provide power and area modeling. In order to assist architects to efficiently explore the gigantic design space and organize the experiment results, an interactive batch system is developed and can operate with a cluster of machines.

2.3.1 Modules and their Extendability

Since M5 was originally designed to study I/O bandwidth for a network of processors, it lacks several components which are necessary for modeling a multicore system. The missing components that are added in MV5 include:

1. Point-to-point OCNs. A large-scale CMP need a distributed, point-to-point OCN

to lower communication overhead and mitigate the increasing bandwidth demand. MV5 provides a template for various OCN options. An OCN is constructed by a set of interconnected routers with a particular topology and a specific routing policy. Deadlock is guaranteed by a correct routing policy. Routers are modeled with receiver buffering and bi-directional links. Two latency models are provided for store-forward and worm-hole routing. Using this OCN template, MV5 builds a 2-D mesh with X-Y routing. The 2-D mesh is configured according a file that depicts the layout of the mesh and which unit connects to which router in an graphical way. The OCN template can be easily extended to other topologies given the corresponding routing policy.

2. Directory-based coherence. Snooping is not available on a distributed OCN and thus cache coherence has to be directory based. MV5 inherits Directory-based caches from default caches in M5 and provides a template for various directory entries and coherence protocols. As a result, coherence handlers form separate modules and are isolated from the cache's basic functionalities. Each coherence protocol presents two sets of coherence handlers describing the block state transition diagram (BSTD) and the directory state transition diagram (DSTD), respectively. Coherence among multiple caches is constructed jointly by their BSTDs and the DSTD of their shared lower level cache. In turn, each cache interfaces with two coherence protocols: its BSTD interfaces with a lower level cache, and its DSTD interfaces with multiple upper level caches. Using this template for coherent caches, we build two inclusive coherence protocols, MSI and MESI.

3. Caches with multiple banks or multiple ports. A cache shared by multiple cores or multiple thread contexts often have multiple banks or multiple ports to meet the bandwidth demand. Users can still choose from various replacement policies.

4. Multi-threaded in-order cores. Data-intensive workloads have frequent memory accesses and may experience frequent cache misses as well. Memory level parallelism (MLP) is able to hide the memory latency by allowing multiple outstanding memory requests and enabling subsequent, independent instructions to continue execution. While out-of-order execution lead to MLP naturally, in-order cores may switch among the instruction streams from multiple hardware thread contexts to achieve MLP. The immediate hardware context switch involves no more than switching to another register file, as modern GPU cores do [70].

5. SIMD cores. Many data-parallel workloads can achieve more throughput from SIMD execution. SIMD has been widely employed by GPUs and the Cell. Traditionally, SIMD is provided in the form of vectorization, which imposes coding complexities when programmers are confronted with divergent branches and address alignment. MV5 simulate a different form of SIMD in which multiple hardware thread contexts form a SIMD group with a set of scalar data-paths operating under a common instruction sequencer, such as the Illiac-IV or modern GPUs, where a non-vectorizing compiler generates SPMD code. It provides simplicity in both hardware implementation and user programming, and it allows threads to have independent control flows. Upon branch divergence, the hardware records in a table about threads' diverged PCs and continues to execute the subgroup of threads with the same PC until they reach a convergence point. At this time, the remaining threads' execution will be continued. Because of the lack of compiler support, we annotate part of the benchmarks that performs data-parallel execution with instructions that signal convergence points. To leverage MLP, hardware thread contexts can statically form multiple SIMD groups and the core can switch among them upon memory accesses.

6. Threading API for system-call emulation. Because system-call emulation does not

provide a fully functional operating system, API calls to create threads over existing operating systems do not work properly, preventing simulation for multithreaded workloads. We implemented a set of threading API calls that include thread creation, join, and termination. Together with M5's existing synchronization API calls such as lock and barrier, they enable simulations for multi-threaded workloads with only minimum changes to the original code.

All the above modules are object-oriented. Each module isolates their external interfaces from their internal functional details. The standardized object interfaces allows object interchangeability: cores can be replaced by another type, coherence protocols can be switched from MSI to MESI, and a 2-D mesh can be substituted by a crossbar conveniently. Moreover, because an object fully encapsulate a component's state, it is easily replicated to form large-scale systems. We have shared the MV5 modules with the M5 community and will work to further integrate them with M5. Our modules have been used by researchers in University of California, San Diego and University of Hertfordshire.

2.3.2 Latency, Power, and Area Modeling and Validation

MV5 models cache latency using Cacti [89]. Pullini *et al.* provide the basis for our interconnect latency modeling [90]. Users can set a core's clock frequency. The instructions-per-cycle (IPC) of an in-order core is assumed to be one except for memory references, which are modeled faithfully through the memory hierarchy (although we do not model memory controller reordering effects). Cores switch SIMT groups in zero cycles upon a cache access.

Energy is modeled in four parts. We use Cacti4.2 [89] to calculate both the dynamic energy for reads and writes as well as the leakage power of the caches. We estimate the energy consumption of the cores using Wattch [91]. The pipeline energy is divided into

seven parts including fetch and decode, integer ALUs, floating point ALUs, register files, result bus, clock and leakage. Dynamic energy is accumulated each time a unit is accessed. The crossbar's switches and routers are modeled after the work of Pullini et al. [90], and we assume the physical memory consumes 220 nJ per access [92]. We neglect refresh power.

To have realistic area esitmates, we measure the sizes of different functional units in an AMD Opteron processor in 130nm technology from a publicly available die photo. We do not account for about 30% of the total area, which is dedicated to x86-specific circuites. We scaled the functional unit area to 65nm with a 0.7 scaling factor per generation. Final area estimates are calculated from their constituent units. We derive the L1 cache sizes from the die photo as well, and assume a 11 mm^2/MB area overhead for L2 caches.

2.3.3 System Reconfigurations and Data Collections

MV5 adopts M5's flexibility in reconfiguring various architectures. A Python script sets up a system by replicating and instantiating Python objects of different types and connecting them through ports. Objects may have different parameters even if they are of the same type. The ability to interchange different object types and different object parameters enables convenient heterogeneous settings. After the system is instantiated, the Python script starts the simulation with appointed benchmark binaries. No recompilation is required after system reconfiguration. Section 2.4 lists the major parameters for major modules.

Figure 2.1 illustrates three examples of system configuration. Figure 2.1a is a dual core architecture with two homogeneous cores and they share the same L2 cache through a bus. Figure 2.1b shows a larger scale multicore architecture with eight IO cores they share a distributed L2 with eight banks through a 2-D mesh. Figure 2.1c demonstrates a heterogeneous multicore system with a latency-oriented OOO core, and a group of throughput oriented SIMD cores. The memory system contains two levels of on-chip caches and an

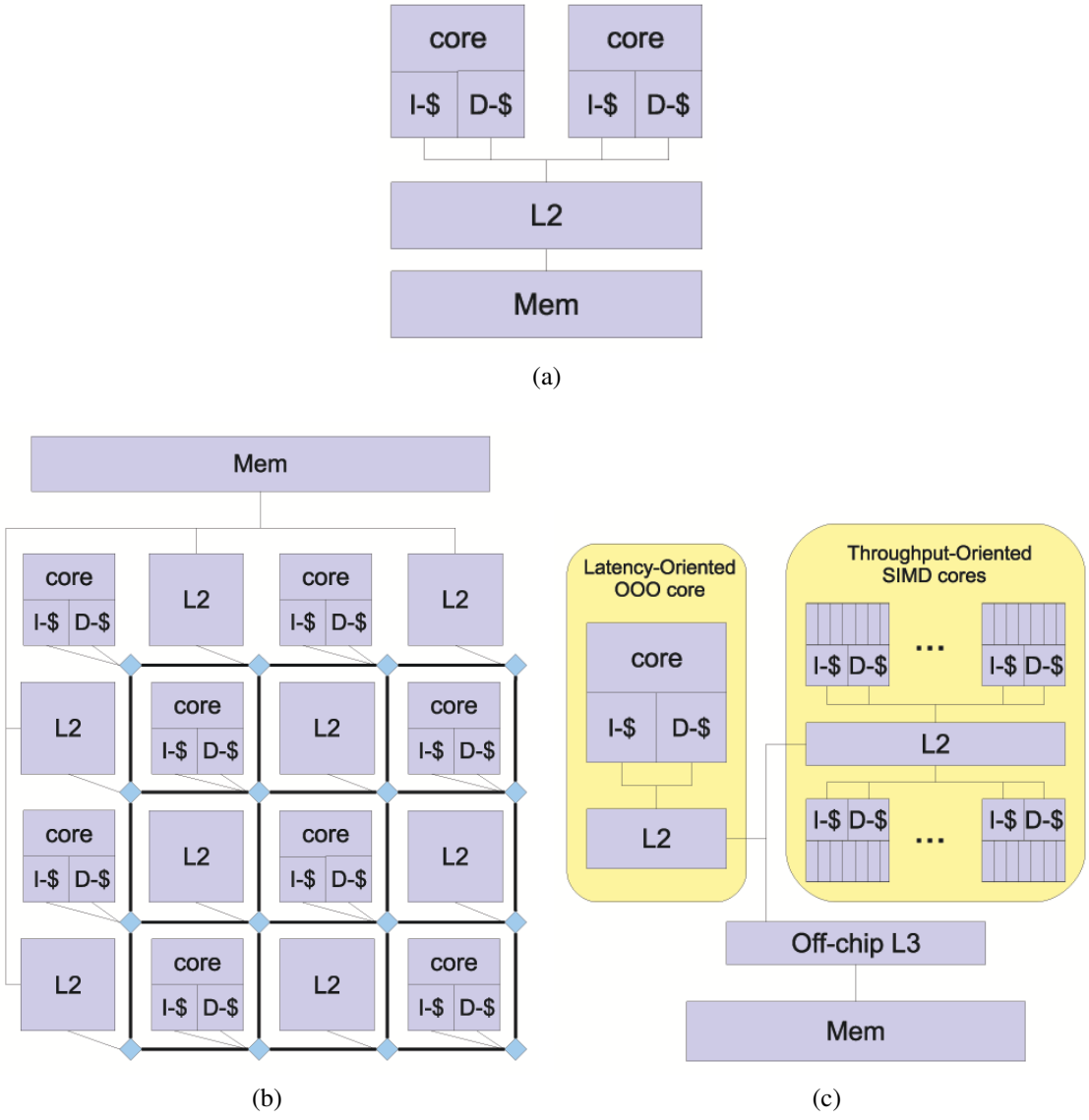


Figure 2.1: Various system configurations: (a) dual core; (b) tiled cores; (c) heterogeneous cores.

off-chip L3 cache.

M5 declares its own counter types for statistical analysis. Various counters are provided in the form of scalars, vectors, distributions, averages, and formulas. All the counters

have names and brief descriptions, and their values will be written to an output file upon termination of the simulation. MV5 embeds counters into each extended modules and users are able to add additional counters as well.

2.3.4 Manageable, Interactive Design Space Exploration

The design space of heterogeneous multicore architectures is already huge, let alone an architecture that integrate CPU and GPU with additional special purposed units dedicated to the graphics pipeline. For example with nine benchmarks, thousands of simulations are necessary only to explore various combinations of five different cache sizes and five different associativities for both private and shared caches.

MV5 scales the architects' ability to explore the gigantic design space of heterogeneous multicore architectures by using an interactive Python program to systematically manage the explosive number of simulation tasks. The program interacts with MV5's Python script and it performs the following tasks:

Creating batches of simulation tasks. By interpreting a user-provided description file listing the ranges of different parameters to explore, MV5 lists all the combinations of different parameter ranges and registers the batch of simulation tasks.

Monitoring and Executing simulation tasks in parallel. MV5 can operate over a cluster of machines. A server program records all the jobs and their status (e.g. pending, running, suspended, finished, failed, etc). Client programs running on the cluster of machines fetch simulation tasks and execute them in parallel. Users can interactively query the status of one or more simulation tasks by providing a task ID or a substring of the tasks' command line. Upon an aborted simulation, the server is notified and users can restart simulation later after debugging. MV5 can also interact with a machine cluster using PBS (Portable Batch System) — a queuing system for a networked multi-platform UNIX environments [93].

Data collection and post-processing. Although all counter values are written to the output file after each simulation, further statistical analysis are necessary to aggregate the detailed counter values from similar units. MV5 enables the user to formulate post-processing in a description file, which MV5 then uses to interpret and aggregate the shear amount of data resulted from a single simulation. This data collection and post-processing process can also be distributed to the cluster machines and execute in parallel.

Tabular and graphical analysis. To help architects analyze data generated by the large number of experiments, MV5 provides analytical tools that grab requested data from the ocean of experimental results and organize data into tables or directly plot them using gnuplot [94]. As a result, users only have to provide MV5 with a description file that defines the groups of experiments to compare, the scaling of parameters and the data values that interest them — completely bypassing the tedious work of data entry.

2.4 Major MV5 Simulation Parameters

Cores	clock frequency number of hardware thread contexts enabling latency hiding by switching thread contexts upon memory accesses context switch latency SIMD warp size
Caches	coherence protocol size line size associativity replacement policy lookup latency prefetch policy number of banks per-bank queue size number of ports number of MSHRs maximum number of outstanding request in each MSHR
OCN	clock frequency routing latency topology buffer size bandwidth forwarding scheme (store-forward or wormhole)
Bus	clock frequency bandwidth
Memory	access latency size
Misc	cache hierarchies number of cores connection topology among cores and caches

Table 2.2: MV5 Parameters

2.5 Conclusion

We conclude several lessons during the implementation of the MV5 simulator and the exploration of the design space. They are critical in managing a large amount of simulations with various configurations.

- Modularity is critical to explore a large design space. An object-oriented, modularized structure is the key to isolate units, replicate them, or extend their functionalities. Modularity also leads to the flexibility in connecting different units. This is especially useful when a large design space needs to be explored with various mutations of similar units connected in different ways.

- Systematic management of a large number of simulations enhances productivity significantly. It integrates creating simulation tasks in batches, launching simulations to a cluster of machines and monitoring their status, adjusting the priorities of different simulations, collecting experimental results, post-processing data, and intuitive graphical analysis. This complete toolchain relieves researchers from data entry and expedites design space exploration and the process of gaining insights.

- Incremental development helps identify and eliminate bugs at early stages. It also improves modeling quality by consolidating the bricks that eventually form a large simulator.

- Instruction traces with register values and load/store addresses help developers to find clues in debugging a large-scale simulator. For example, developers may find inconsistent values between a load and a previous store, he or she can then further investigate traces for the coherence caches to find out what is wrong.

- It is worthwhile to duplicate pieces of code if this leads to better code-readability

and modularity. In MV5, coherence protocols are programmed as separate finite state machines and each transition is described by a distinct class. Although multiple transitions may appear similar, separating them leads to a clean description of the protocol and we are able to easily extend the protocol from MSI to MESI.

Chapter 3

Exploiting Inter-thread Spatial and Temporal Locality

3.1 Introduction

Workloads with significant data parallelism are gaining commercial and social importance and driving processor design in many markets. For applications with sufficient parallelism, it has been shown that a *chip multithreading* (CMT) organization comprising many simple, multithreaded cores maximizes performance within a given area budget [95]. Multithreading hides latencies and can better utilize precious memory bandwidth. This design philosophy is apparent in a number of contemporary and proposed architectures, including Niagara [8], the Cell Broadband Engine (CBE) [9], and GPUs [13, 14] (which despite their origin in rendering, have proved effective at a variety of non-rendering, general-purpose workloads [20]). The addition of single-instruction, multiple-data (SIMD) hardware further increases thread count. SIMD organizations are appealing because they boost area efficiency for data-parallel workloads and amortize the cost of instruction storage, fetch,

decode, and sequencing across multiple processing elements. Wide SIMD loads can also make more efficient use of memory bandwidth [17][1].

Thread counts per core seem likely to increase, even though L1 cache sizes are unlikely to keep pace. Unfortunately, the benefits of increasing threads per core will be limited by cache contention unless threads are carefully co-scheduled. Unfortunately, current schedulers typically treat each thread as if it ran on a separate virtual processor and hence are oblivious to interactions when threads share a cache. Previous techniques attempt to mitigate cache contention by reconfiguring the cache [44, 45, 46] or selecting the best combination of heterogeneous parallel threads that provide the best throughput [40, 41, 42]. Yet, these techniques view threads as competing entities working on disjoint sets of data. Data-parallel tasks share a common data set and typically exhibit predictable access patterns. Work should be partitioned first according to the number of caches, and threads within a core should then be cooperative and access data in a pattern maximizing spatial and temporal locality. We name this concept *Symbiotic Affinity Scheduling* (SAS) and demonstrate its benefits on a wide spectrum of data-parallel applications ranging from media processing and scientific computation, to mining and machine learning.

It may appear that conventional affinity-aware techniques can be straightforwardly adapted to symbiotic threads as a form of SAS. However, as we will discuss in Section 3.2.1, these techniques only address cache affinity for an *individual* thread or among *dependent* threads, rather than among *concurrently executed* symbiotic threads. Even if some cache-aware techniques can be adapted to ensure the joint working set of symbiotic threads fit in the local cache, they may still suffer from conflict misses and a cache block may not be reused

[1] Vectors are one form of SIMD. Although they are currently only four lanes wide in most commodity processor ISAs, they will grow to 8- and 16-wide with the introduction of future ISAs [96, 15]. An alternative SIMD organization is the array organization (dubbed "SIMT" by NVIDIA for Single Instruction, Multiple Threads), in which each lane is scalar and executes a separate thread context, and the SIMD lockstep operation is implicit and not directly exposed in the ISA. Array processing has a rich history in high performance computing, and GPUs are the commodity exemplar of this architecture.

in time before it gets replaced.

The solution to this issue is to also exploit *temporal* locality among symbiotic threads to maximize sharing and data reuse. By allowing symbiotic threads to simultaneously process *adjacent* data, the overall sequence of memory addresses they access is similar to that of a single thread that iterates through tasks in order. As a result, they can achieve parallel performance scaling without sacrificing locality. However, due to run-time dynamics, fine-grained coordination among symbiotic threads is required to ensure inter-thread temporal locality; some data may incur more computation or cache misses than others. In such cases, it is important for symbiotic threads to re-adjust their task[2] distribution so that adjacent data can still be reused in time. Therefore, we propose SAS as a run-time approach with two stages. First, independent tasks are grouped into *blocks* using cache-affinity optimizations based on the layout and capabilities of the cache hierarchy; each block is assigned to an individual core to leverage spatial locality. Secondly, because nearby tasks in the same block are likely to share data for regular access patterns, each core then traverses its block and dispatches *neighboring* tasks to multiple *concurrent* symbiotic threads on the same core to leverage temporal locality.

In SAS, a block of tasks is traversed by a per-core scheduler according to an *affinity graph of independent tasks (AGIT)* as an indicator of locality among tasks. In an AGIT, tasks are represented as vertices and those that share data are connected by undirected edges. For many parallel applications with regular data access patterns, the AGIT may be constructed implicitly without programmer intervention (e.g. the AGIT can be formed into regular meshes, lists or trees according to different data structures and access patterns). Otherwise, AGIT construction would rely on instrumentation.

In this chapter, we demonstrate SAS for data-parallel applications whose parallelism is

[2]We refer to a set of dynamic instructions that have to be executed sequentially as a *task*, and it often corresponds to the code section in the innermost parallel loop.

often expressed in nested, parallel `for` loops. Specifically in this scenario, a task refers to the data-parallel computation scoped within the innermost parallel `for` loop, where each task can be identified by a particular loop index. The AGIT is implicitly constructed as a multi-dimensional mesh according to the loop space. Conventional tiling or blocking techniques [21, 25, 30] group a set of neighboring tasks into blocks called *tiles* and execute each tile within an individual thread. These techniques are referred to in this chapter as *Individual Tiling* (I-tile). Using SAS, we propose *Symbiotic Tiling* (S-tile) that improves inter-thread temporal locality by allowing a tile to be *collaboratively* processed by symbiotic threads; concurrent threads execute neighboring tasks, which are likely to access adjacent data. We compare S-tile to I-tile on a CMT processor with 16 threads per core and a two-level coherent cache hierarchy. Due to the unavailability of general purpose CMT processors with a high degree of multi-threading, we simulate a set of nine benchmarks selected from Splash2 [97], MineBench [98] and Rodinia [20]. Experiments show that S-tile provides an average speedup of 1.69$\times$ and energy savings of 33% compared to I-tile. Even if I-tile is adapted to assign smaller tiles to each thread so that the aggregate can fit in the L1 cache capacity (I-tile(part)), S-tile still achieves greater locality and 30% performance gains. We also compare S-tile to SOS (Sample, Optimize, Symbios) scheduling, an adaptive contention reduction technique that reduces the number of active threads when contention is detected; results show S-tile outperforms SOS by 17%. This work was published in [65].

3.2 Background

To avoid cache contention among threads on the same core, we propose to regard threads on the same core as a joint set of work and improve their *joint* cache affinity. We investigate not only *inter-thread spatial locality* where the joint working set of symbiotic threads fits

in the cache capacity, but also *inter-thread temporal locality* where threads reuse the same data within a small time interval. In short, for a given partition of the data, both spatial and temporal locality are maximized if threads operate on immediately adjacent data elements, rather than further partitioning the data into disjoint blocks.

There are abundant studies that aim at effectively reordering or distributing computation tasks to maximize memory system throughput. They can be further divided into two complementary categories: techniques that map computation tasks to cores according to cache affinity or data locality, and techniques that reduce contention once tasks have already been mapped to cores. We present a road map of these techniques and study the differences between the proposed technique and the conventional techniques.

3.2.1 Affinity-Aware Task Scheduling

Affinity-aware task scheduling can occur during two stages: first, threads are *constructed* as virtually independent cores where each processes a disjoint block of neighboring tasks; and second, threads are *scheduled* on the underlying architecture with a particular order or mapping that optimizes cache-affinity.

Affinity-Aware Thread Construction

An *individual* thread can be created in a way that maximizes spatial locality by computing a sequence of neighboring tasks that are likely to access adjacent data. For tasks with regular access patterns, *Tiling* or *Blocking* can be employed to partition the workload into consecutive chunks that each map to a thread [21, 22, 23, 24, 25, 26, 27, 28, 29, 30] When data accesses are irregular, the underlying system can still reorder independent sections of code according to user-instrumented address hints that indicate data locality among code sections [31] or runtime discovery.

These techniques improve data locality within an individual thread. Nevertheless, when multiple threads share the same cache, their working sets do not necessarily overlap, and their joint data-accesses are likely scattered, leading to a higher possibility of conflict and capacity misses. Even if some cache-aware techniques may adapt to multi-threaded cores by subdividing a data partition that fits in the cache capacity among symbiotic threads, they still suffer from a lack of inter-thread temporal locality. For temporal locality within a thread, a cache block may have to persist in the cache for a long time to be reused again. On the other hand, for reuse across threads, if the cache block can be simultaneously reused by multiple threads (our approach), its risk of being replaced before fully reused becomes lower. This observation is justified in Section 3.6 by comparing the performance of two techniques (i.e. I-tile(part) and S-tile).

Affinity-aware Thread Scheduling

Once threads are created, there are several approaches to improve cache affinity:

- An *individual* thread is scheduled on the core where it has run previously to reuse remaining data in the private cache [32].
- An *individual* thread migrates closer to the cache or memory from which it frequently requests data [33, 34, 35].
- *Dependent* threads are scheduled on the same core to save data communication [36, 37, 38, 39].

These scheduling techniques are based on the history about where threads have executed or the knowledge of data dependency among threads, neither of which indicate affinity between concurrent threads sharing data, a challenge raised when data-parallel applications run on multi-threaded cores.

Affinity-aware Workload Partitioning

While the above techniques improve data locality either within a thread or among dependent threads, there is limited work in improving affinity among *concurrent* symbiotic threads, an issue raised by multi-threaded cores. Lo *et al.* proposed a workload partitioning technique that subdivides a page of data among concurrent threads on the same core to minimize TLB footprints [99]. However, such static approaches cannot be easily adapted to reduce cache footprints, which requires a more fine-grained orchestration between threads. A more detailed discussion can be found in Section 3.3.1. One adaptation of this technique to reduce cache footprints is to further divide a data partition that fits in the cache capacity among concurrent threads. We refer to it as *I-tile(part)* and Section 3.6 shows that it is inferior to S-tile.

3.2.2 Contention Reduction among Threads

Once threads are created and mapped to cores, it may occur that some storage resources may be shared among multiple threads. These resources include the shared last-level cache, private caches and TLBs. There are two main approaches to reduce such contention:

- Only a few threads are selected from a pool of threads whose joint working set minimizes cache contention [40, 41, 42, 43].
- Shared storage can be dynamically partitioned among threads according to their distinct demands [44, 45, 46].

These techniques are mostly intended for heterogeneous threads with different demands in cache capacity or associativity. While they are able to prevent cache thrashing and optimize cache throughput, they do not address data reuse among threads that would further

improve the computational throughput. Because only the overall performance is dynamically profiled, these techniques are not aware of data access patterns among concurrent threads and are not able to improve inter-thread data sharing.

Nevertheless, contention reduction techniques can serve as complementary optimizations to cache sharing; there can be severe contention even if data is intensively reused among threads. We study the impact of these techniques in Section 3.6 with I-tile(SOS) and S-tile(SOS).

3.2.3 Fine-grained Parallelism and Vector Processing

The trends for deeply multi-threaded cores to support fine-grained parallelism can be observed from Niagara [8], the Cell Broadband Engine (CBE) [9], and graphics processors such as NVIDIA's Tesla [13] that are sometimes used for general purposed computation. While hardware support has been proposed to assist fast dispatching of fine-trained tasks [100], we are not aware of any prior technique that aims to automatically improve affinity among concurrent, fine-grained tasks, even in the newest OpenMP specification [101].

Vector processing may appear similar to our proposed technique; "threads" are grouped into vectors and the need to access data in vector-sized chunks forces the programmer to build affinity among threads in the same vector. Vector processors may have multiple, vector-width threads so that the core can switch among them to hide memory latency, however, there is nothing to force the programmer to assign neighboring tasks to threads in different vectors, which requires an understanding of the data access patterns for each vector. Our proposed technique accomplishes this at runtime without burdening the programmer or compiler. Compared to CUDA [78], whose abstraction of the explicitly-managed, per-block shared memory requires the programmer to manually manage data affinity at a

greater effort, our technique works with implicitly managed caches, and it automatically schedules fine-grained tasks with inter-thread data affinity at runtime.

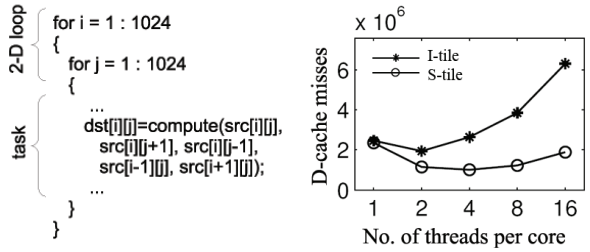
3.3 Symbiotic Affinity Scheduling for Data-Parallelism

We first introduce some common concepts about tiling. An example is then used to illustrate the conceptual benefits of our technique.

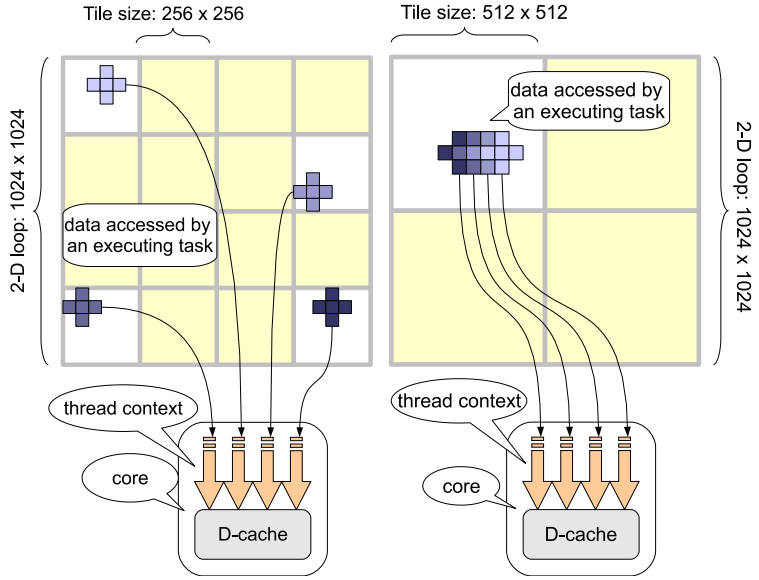
3.3.1 Tiling for Symbiotic Threads

Tiling targets data-parallel applications with nested, parallel `for` loops. An N level nested parallel `for` loop intuitively defines an N-dimensional grid of loop indices. An innermost iteration corresponds to a parallel *task*, which is associated with a particular loop index. A *tile* is a block of tasks that associates with a block of contiguous loop indices. Given regular data access patterns, a tile demonstrates good internal data locality. Conventionally, a tile is assigned to an *individual* thread, and because each thread is assumed to run on a separate processor with its own cache, the only locality is intra-thread data locality. These techniques are referred to in this chapter as individual tiling (I-tile) and are widely used in parallel programming models such as OpenMP [102] and TBB [103].

Sadly, I-tile does not address the temporal locality of data accesses among symbiotic threads; even if two threads on the same core are assigned neighboring tiles that have bordering tasks, the amount of data-sharing is negligible compared to threads' overall working set. In addition, because threads in I-tile traverse tiles independently, neighboring tasks belonging to different tiles are seldom executed close in time to reuse data. It may also appear that I-tile can be adapted to symbiotic threads by creating smaller partitions so that



(a) code sample (b) D-cache misses



(c) I-tile (d) S-tile

Figure 3.1: Comparing I-tile and S-tile in the context of HotSpot. (a) Pseudocode of HotSpot (b) D-cache misses as a function of number of threads per core on a quad core system. (c) I-tile: the nested parallel loop is partitioned according to the number of available *thread contexts*. Symbiotic threads work on distinct, scattered data. (d) S-tile: the nested parallel loop is partitioned according to the number of *D-caches*. Symbiotic threads work on neighboring tasks with data locality.

the sum of all the working sets of all threads on the same core will fit in the cache. However, it is very difficult for such an approach to cope with conflict misses, as we will show in Section 3.6 with I-tile(part).

We propose *symbiotic* tiling (S-tile) that performs tiling according to the number of D-caches rather than the number of threads and dispatches data sharing tasks to concurrent threads. Because tasks with neighboring loop indices are more likely to share data given regular access patterns, the conceptual AGIT corresponding to the nested, parallel `for` loop can be simply regarded as a multi-dimensional mesh defined according to the lower bound, upper bound, and stride in each dimension of the tile. The AGIT is then traversed by symbiotic threads in a round robin fashion to ensure threads execute neighboring tasks.

It may appear that S-tile can be achieved by static analysis: tasks can be distributed to symbiotic threads in a cyclic way, similar to what Lo *et al.* proposed in cyclic scheduling where a page of data is distributed cyclicly among simultaneous threads in order to reduce TLB footprints [99]. However, reuse of cache blocks, rather than pages, is more sensitive to run-time dynamics; threads with unbalanced workloads, due to control flow or memory latency, may fall behind, and static approaches fail to re-align their task assignment. These fall-behind threads may not able to reuse cache blocks in time. Threads may fall behind even in SIMD cores — although threads in the same SIMD group are synchronized with their tasks are aligned, there are usually multiple SIMD groups on the same core, and threads in different SIMD groups may execute asynchronously for latency hiding purposes. In addition, static scheduling in a cyclic way requires knowledge of cache capacity and tile size, which either varies in different processors or may not be known until runtime. This leads to high complexity and poor portability. Given all of these factors, we implement our solution as a user-level, runtime library.

3.3.2 Case Study: HotSpot

HotSpot [104] is a thermal simulator that estimates processor temperature based on block layout and performance measurement in architectural simulation. It is a member of the structured grid dwarf [105], in which computation can be regionally divided into sub-blocks with high spatial locality. Structured grid applications are at the core of many scientific computations. Other notable examples include Lattice Boltzmann hydrodynamics [106] and Cactus [107].

Figure 3.1a shows a simplified 2-D version of HotSpot's code. In each iteration, a task gathers five neighboring elements on a 2-D grid to produce a new value. Assuming data array is row-major, the 5 neighboring elements are scattered in three D-cache blocks.

When HotSpot is parallelized using I-tile, it queries for the number of existing thread contexts and tiles its parallel loop accordingly. As Figure 3.1c illustrates, on an example parallel system with four 4-way multithreaded cores, sixteen 256×256 tiles are created to operate over a 1024×1024 grid to match the number of *hardware thread contexts*. Each thread then executes a distinct tile. Since tiles have minimally overlapping working sets, there is hardly any cache sharing among threads on the same core. Concurrent tasks executing over the same D-cache request 20 elements scattered among 12 D-cache blocks.

S-tile, on the other hand, partitions the same parallel nested loop into four 512×512 tiles to match the number of *D-caches*. In our example of a four-core CMT, each tile is then assigned to a core with four threads sharing the same cache. Consecutive, *neighboring* tasks in the same tile are then dispatched to concurrent threads. As Figure 3.1d shows, concurrent neighboring tasks reuse adjacent data in a short span of time; in fact, at the same time in the case of SIMD access. As a whole, our example requests only 14 elements scattered in three D-cache blocks, assuming each D-cache block is 32 B and can host up to eight elements in a row. In this way, S-tile's L1 cache footprint is similar to that of a single threaded

core! Although it may seem that I-tile can achieve the same data locality by only activating one thread per core, this would lose all the benefits of multithreading. This comparison is further evaluated in Section 3.6.1.

As a result, we show in Figure 3.1b that as we increase the number of threads per core together with D-cache associativity from one to 16, HotSpot experiences dramatic increases in D-cache misses with I-tile because of D-cache contention. On the contrary, the number of D-cache misses remains relatively constant with S-tile. I-tile(part) helps, but because each thread works on disjoint tiles—even though these tiles were sized to try to fit in the cache—run time dynamics can lead to contention that S-tile does not suffer. The reduced D-cache contention leads to better speedup and energy efficiency, as will be shown in Section 3.6.

3.4 Implementation

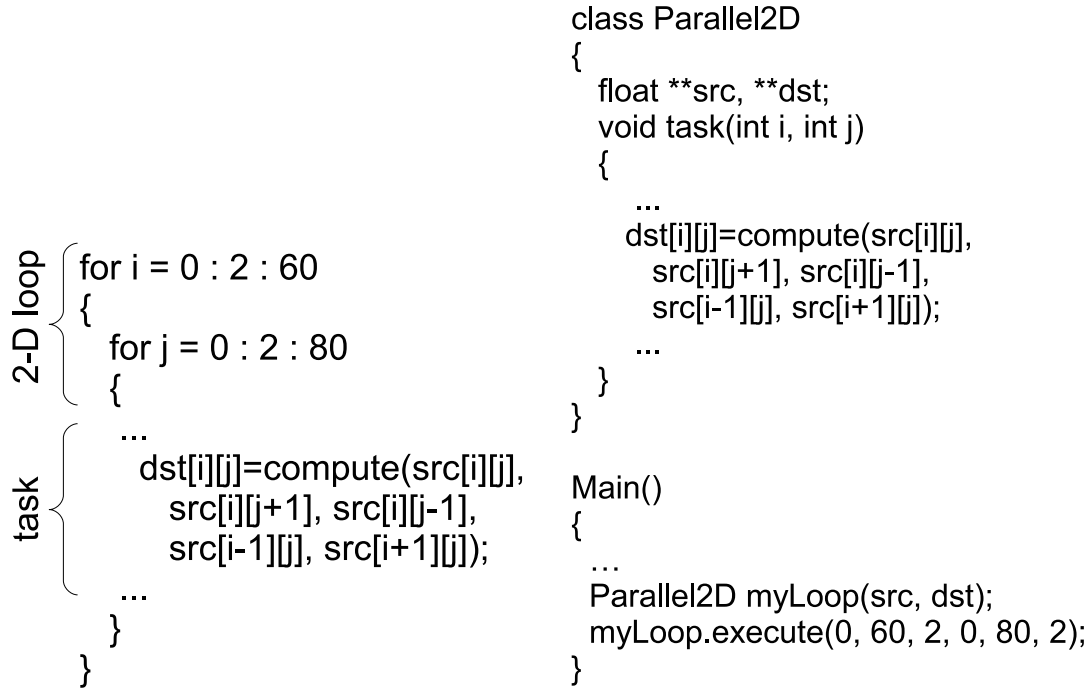
Due to the lack of commercialized systems with many general purpose cores that have a high degree of multithreading, we simulated our benchmarks on MV5, an event-driven, cycle-accurate multicore simulator based on M5 [64, 84]. Since TBB [103] and OpenMP [101] programs use the Pthread library, which does not execute properly in system emulation mode, we implemented a user-level runtime threading library supporting basic operations required for a split-join threading model. A nested, parallel `for` loop is abstracted as a generic C++ class whose member function can be derived to encapsulate code sections within the innermost loop. This member function computes an individual task given a particular loop index. When instantiated with loop boundaries and strides, the C++ object invokes the threading library which partitions the loop, schedules and executes all the tasks. Figure 3.2 gives an example of how our threading API transforms a parallel `for` loop. Such an API is not new, and there are existing techniques that can automatically extract

boundaries and strides of a parallel `for` loop such as employed by OpenMP [101]. Given appropriate compiler support or code transformation, our runtime technique can work with with existing APIs without modifying applications' source code; our threading API is only designed to mimic the programming interface in existing parallel APIs in our simulation. Benchmarks are all cross-compiled to the Alpha ISA using gcc 4.1.0.

Internally, the run-time library interprets the upper bounds, lower bounds, and strides of nested parallel `for` loops. The run-time library then creates a monitor thread that executes on a randomly chosen core. The monitor thread in turn spawns threads on all available hardware thread contexts and then acts as the *centralized tile scheduler*, which is responsible for constructing the AGIT and partitioning the loop according to either I-tile or S-tile.

The AGIT for regularly strided, parallel for loops is a two dimensional array with three rows, each representing the upper bounds, and lower bounds, and strides in all levels of the nested loop. It is allocated in the heap of the monitor thread, and the cost in the memory space is negligible (N words, where N is the number of levels in the nested loop. Note N does not increase with a larger input size). The monitor thread is only activated during the sequential phase right before the parallel phase. After it partitions the loop space and distributes the tiles, it is suspended with its context switched out. The amount of work it performs does not scale with the input size and is small compared to the actual tasks performed by the parallel threads. For heterogeneous architectures, the monitor thread can run on a latency-oriented out-of-order core, while other parallel threads run on a set of throughput-oriented cores.

In I-tile, the centralized tile scheduler evenly partitions the loop according to *the number of hardware thread contexts*. Tiles are represented concisely by their boundaries and strides, and are stored in a shared memory structure. On each core, an individual thread repeatedly checks a flag to see whether a new tile is available, and acquires the tile if so. It then computes the tile by repeatedly calling the function representing the innermost loop



(a) Conventional code with a parallel `for` loop.

(b) The expression of a parallel `for` loop in our threading API

Figure 3.2: Representation of a parallel `for` loop.

within the tile boundary.

In S-tile, the centralized tile scheduler evenly partitions the loop according to *the number of cores*. Each multi-threaded core has one of its threads accept the tile in a manner similar to I-tile. This thread then acts as the *per-core task scheduler*. It first allocates queues for storing loop indices for *each* thread on that core. It then iterates through the tile; instead of computing tasks, it only generates loop indices which are then pushed into the pre-allocated queues in a round robin fashion. Afterwards, threads on the same core, including the per-core task scheduler, fetch loop indices from their associated queues and execute tasks in parallel. As a result, multiple threads do not compete for the same task queue. The per-core task scheduler keeps filling a queue that is nearly empty. A thread will

be busy waiting once it finds its queue empty.

The runtime library in S-tile effectively dispatches fine-grained tasks to symbiotic threads; hardware support for fine-grained parallelism, such as that proposed by Kumar *et al.* [100], is helpful but not necessary for three reasons. First, overhead in the operating system is negligible because everything takes place at user-level except for creating the initial threads. Secondly, dispatching a task is as simple as copying the next loop index from a queue and calling the function representing the innermost loop. Finally, it usually takes the per-core task scheduler fewer than 10 instructions to advance to the next loop index and buffer it. Therefore it does not create noticeable overhead since each task usually takes hundreds or thousands of instructions to execute. By manually experimenting with the run-time overhead, we found that a latency of 10 cycles in scheduling each task yields an average performance overhead of 0.7%. Note that some of the computational latency resulting from task scheduling can be hidden due to the overlap of memory accesses. However, if tasks are only a few cycles long, the overhead in the run-time may still be significant. In such cases, a hardware task scheduler can be used instead. We have modeled it using a Virtex-II Pro XC2VP30 FPGA [108]. The hardware implementation requires an equivalent gate count of 3117, while even a simplest, single-threaded in-order core would still take at least tens of thousands of gates (35K gates for a 32-bit LEON2 processor compliant with the SPARC V8 architecture). The hardware implementation is able to generate a loop index every cycle at 1.0 GHz.

3.4.1 Over-Decomposition and Load Balancing

Workloads are *always* balanced among symbiotic threads because the per-core task scheduler keeps filling the queues of loop indices, and any idle thread can fetch a new task from

its queue. Under most circumstances, tiles are similar in size with similar amounts of computation, and workloads among cores are balanced as well. Therefore, the number of tiles by default equals the number of cores in S-tile. However, in the case of those applications that may have unbalanced workloads among tiles, programmers can over-decompose the parallel loop into smaller tiles with a larger quantity than the number of cores [27]. Load-balancing is then achieved by having idle cores fetch remaining tiles. Over-decomposition is not used when generating the results presented in Section 3.6.

3.5 Methodology

Our simulations model multi-threaded cores that operate over coherent cache hierarchies, resembling those found in Niagara [8], Larrabee [15], and Fermi []. Our core model can host hundreds of thread contexts. With such a degree of multi-threading, a deeply pipelined, out-of-order core may be neither area nor energy efficient. Instead, SIMT (Single Instruction, Multiple Threads) is more common for highly multi-threaded cores, and therefore we base our experiments upon SIMT modeling. SIMT cores group multiple scalar threads into SIMD batches that operate under a common instruction sequencer. Different from traditional SIMD structures based on vectorization, SIMT is formed by multiple scalar threads that maintain their own registers and they may follow different control flows. These properties entitle the processor to accommodate SPMD (Single Program, Multiple Data). SIMT is now used in commodity graphics processors such as NVIDIA's Tesla [13]. It is used not only for graphics but also for a wide range of general purpose applications [20]. While our evaluation focuses on SIMT, the principles of SAS and symbiotic tiling apply to multi-threaded cores of any width. To explore the applicability of symbiotic tiling on cores with various SIMD widths and multithreading depths, e.g. Niagara or Larrabee, we scale the SIMT width from one to eight in Section 3.6.1.

3.5.1 Modeling and Configuration

We model cache latency using Cacti [89]. Pullini *et al.* [90] provide the basis for our interconnect latency modeling. The per-thread IPC (instructions-per-cycle) is assumed to be one except for memory references, which are modeled faithfully through the memory hierarchy (although we do not model memory controller reordering effects). Cores switch SIMT groups in zero cycles upon a cache access by pointing to another set of register files, as commodity GPUs do [13].

The on-chip memory system has a two level hierarchy. Each core has a private I-cache and a private D-cache. D-caches are banked to cater to the bandwidth demands of multiple thread contexts. A thread context can access any D-cache bank. If bank conflicts occur, memory requests are serialized and a one cycle queuing overhead is charged; the queuing overhead is much smaller than the hit latency because we assume requests can be pipelined. I-caches are not banked because only one instruction is fetched every cycle for an entire SIMT group. All L1 caches share the L2 cache through a crossbar. The L2 cache is inclusive and can hold more than twice as much data as all of the L1 caches combined. Caches are physically indexed and physically tagged. We employ the MESI directory-based coherence protocol. Table 4.1 summarizes the main system parameters.

Energy is modeled in four parts. We use Cacti 4.2 [89] to calculate dynamic energy for reads and writes as well as the leakage power of the caches. We estimate the energy consumption of the cores using Wattch [91]. The pipeline energy is divided into seven parts including fetch and decode, integer ALUs, floating point ALUs, register files, result bus, clock and leakage. Dynamic energy is accumulated each time a unit is accessed. Energy in crossbar's switches and routers are also modeled after the work of Pullini *et al.* [90], and we assume the physical memory consumes 220 nJ per access [92]. We neglect refresh power.

Tech. Node	65 nm
Cores	Alpha ISA, 1.0 GHz, 0.9V Vdd. in-order. 16-way multithreaded: two SIMT groups of width eight
L1 Caches	physically indexed, physically tagged 16 KB I-cache and 16 KB D-cache, 32 B line size 16-way associative, 16 MSHRs, write-back 3 cycle hit latency, 4 banks, LRU
L2 Cache	physically indexed, physically tagged 16-way associative, 128 B line size, 16 banks 1024 KB, LRU, 32 cycle hit latency, write-back 64 MSHRs, $\leq$ 8 pending requests each
Interconnect	crossbar, 300 MHz, 57 GB/s
Memory Bus	266 MHz, 16 GB/s
Memory	50 ns access latency

Table 3.1: Default system configuration

3.5.2 Benchmarks

We select a set of parallel benchmarks from several benchmark suites. Our primary objective is to obtain representative data-parallel applications with distinct data access and communication patterns. To maintain manageable simulation times, the input size is carefully chosen so that it assigns sufficient work to each core and the overall working set is larger than the capacity of the L2 cache size. This is large enough, since our technique mainly addresses L1 cache misses. We have tried some experiments with larger input sizes and the speedups stayed almost the same. Table 4.2 summarizes our selected benchmarks, including the dominant application behavior.

3.6 Evaluation

We use I-tile as a baseline and compare its performance to S-tile. Two other implementations, adapted from conventional techniques, are also compared: I-tile(part) subdivides a data partition that fits the cache capacity among symbiotic threads; I-tile(SOS) is named after *Sample, Optimize, Symbios* (SOS) scheduling that selects an appropriate set of threads

	Benchmark Description
FFT	Fast Fourier Transform (Splash2 [97]) Spectral methods. Butterfly computation Input: a 1-D array of 32,768 (2^{15}) numbers
Filter	Edge Detection of an Input Image Convolution. Gathering a 3×3 neighborhood Input: a gray scale image of size 500×500
HotSpot	Thermal Simulation (Rodinia [20]) Iterative partial differential equation solver Input: a 300×300 2-D grid, 100 iterations
LU	LU Decomposition (Splash2 [97]). Dense linear algebra Alternating row-major and column-major computation Input: a 300×300 matrix
Merge	Merge Sort. Element aggregation and reordering Input: a 1-D array of 300,000 integers
N-W	Needleman-Wunsch DNA alignment (Rodinia [20]). Dynamic programming: Updating matrix with a diagonal wavefront Input: a 2-D array of size 512×512
Short	Winning Path Search for Chess. Dynamic programming. Neighborhood calculation based on the previous row Input: 6 steps each with 150,000 choices
KMeans	Unsupervised Classification (MineBench [98]). Map-Reduce. Distance aggregation. Input: 10,000 points in a 20-D space
SVM	Supervised Learning (MineBench [98]) Support vector machine's kernel computation. Input: 1,024 vectors with a 20-D space

Table 3.2: Simulated benchmarks with descriptions and input sizes.

to reduce cache contention [41]. In our adaptation, SOS first experiments with different SIMT widths and then chooses the one that yields the best throughput—wider SIMT execution increases parallelism but also increases cache contention. The optimal SIMT width can be equal to or less than the available *pipeline width* (i.e. the number of SIMD lanes) provided by the hardware. SOS can be applied to S-tile as well as a complementary technique, and we name it S-tile(SOS). Table 3.3 summarizes the combinations of partitioning and scheduling techniques that we study.

To take into account the run-time dynamics, the performance of each configuration is presented as the mean of the outputs from five simulations. Within each simulation we randomly assign tiles to threads (in I-tile systems) or cores (in S-tile systems). For simulation results belonging to the same configuration, the coefficient of variation for executed cycles

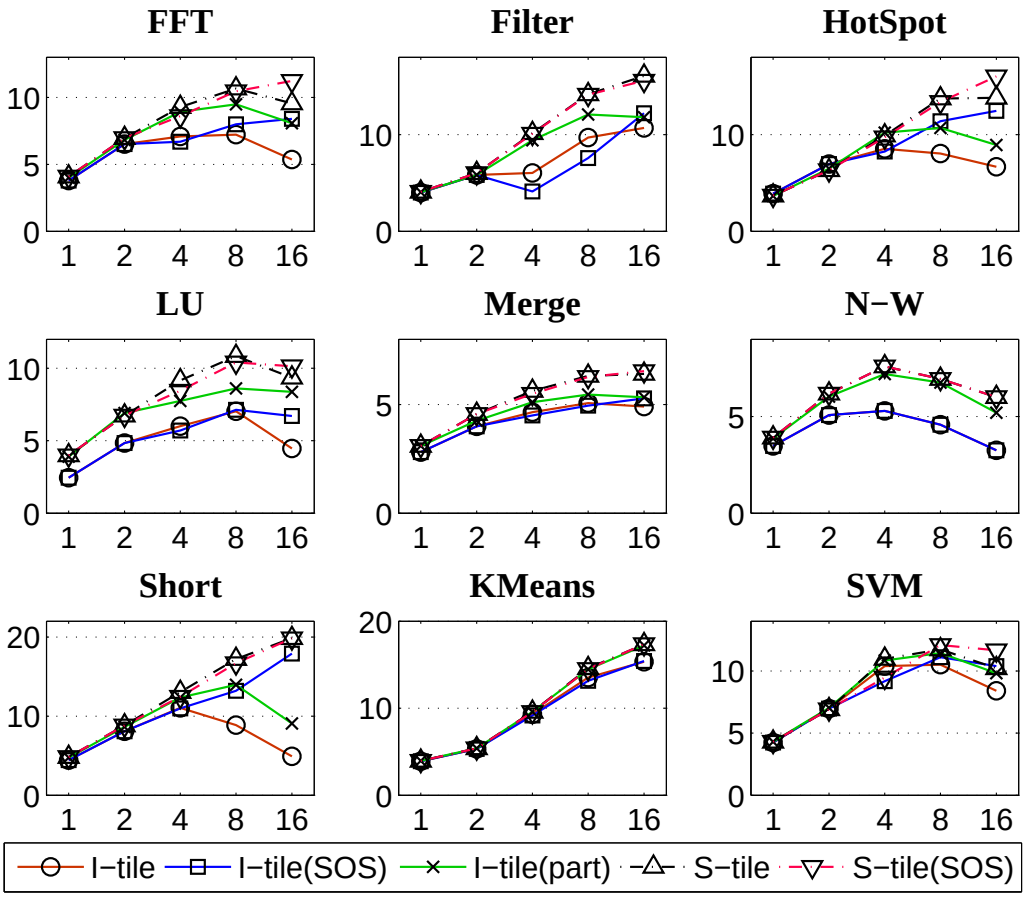


Figure 3.3: Speedup vs. Number of threads per core, measured on four cores each with two SIMT groups except for the case with one thread per core. D-cache associativity equals the number of threads. Speedup is normalized to single-threaded execution.

Configuration	SAS	Adaptive No. of Threads	Cache-aware Data Partition
I-tile	N	N	N
I-tile(SOS)	N	Y	N
I-tile(part)	N	N	Y
S-tile	Y	N	N
S-tile(SOS)	Y	Y	N

Table 3.3: Different partitioning and scheduling combinations

usually falls below 1%.

3.6.1 Performance Speedup

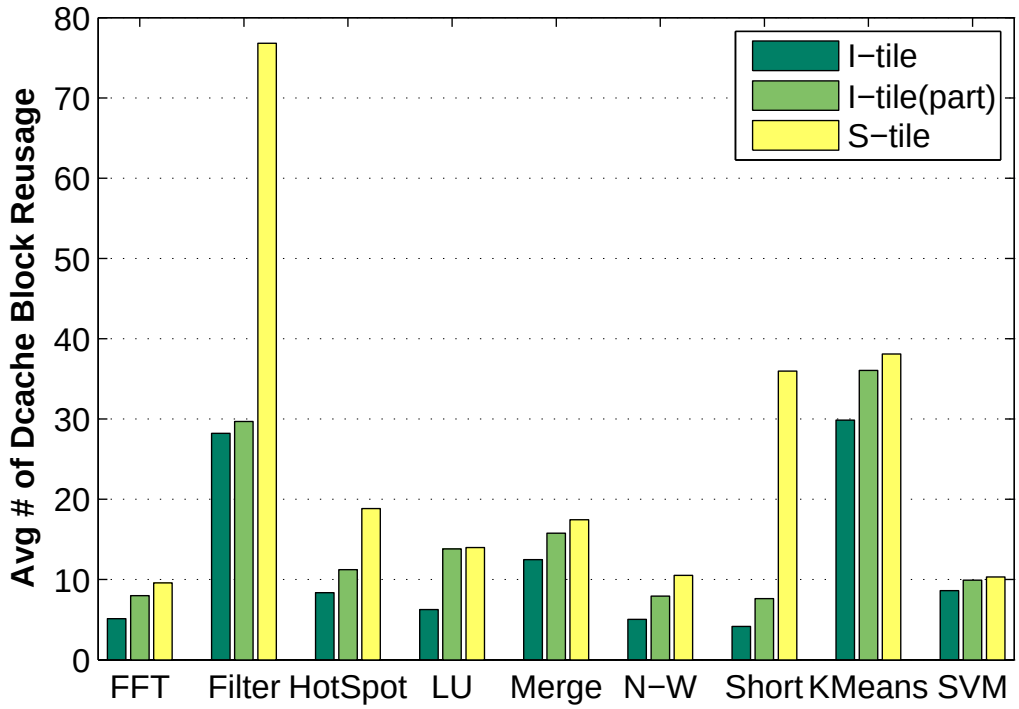


Figure 3.4: The average number of reuses for D-cache blocks. Data are measured with four cores each with two SIMT groups of width eight. The D-cache associativity is 16.

To compare the overall performance scalability for the five systems, we increase the number of threads per core from 1 to 16. Experiments are conducted on a simulated four-core CMT. Within each core, execution switches among active threads or thread groups to hide memory latency (except of course when only one thread context is available). Switching among threads costs zero cycles.

As Figure 3.3 shows, for two, four, eight and 16 threads, the average speedup of S-tile over I-tile is 1.08×, 1.25×, 1.43×, and 1.69×, respectively. Not surprisingly, S-tile is more beneficial to *high-dimensional access patterns* — strided accesses resulting from tasks nested in multilevel parallel loops (FFT, Filter, HotSpot, LU) or tasks that gather and scatter high-dimensional data (Filter, HotSpot, LU, N-W, Short). Tasks in these applications involve scatter or gather memory addresses with large strides, leading to less locality. Cache conflicts are more likely to happen and therefore exploiting cache affinity becomes more critical. This phenomenon is more evident in Figure 3.4 where D-cache misses and reuses are characterized; S-tile improves data reuse significantly in those workloads.

Although I-tile(SOS) is able to improve I-tile's performance to some extent, or to degrade more gracefully, it does not achieve as much performance gain as S-tile does alone; different from I-tile(SOS), S-tile reduces cache contention without decreasing SIMT pipeline utilization. I-tile(SOS) may also fail to adapt to runtime phase changes, and its performance may even degrade. In general, S-tile outperforms I-tile(SOS) by 31% on an 8-way CMT and 17% on a 16-way CMT.

On the other hand, Figure 3.3 shows that S-tile(SOS) performs similarly to S-tile, however, SOS may benefit S-tile where computational resources are extremely limited, as we will show in Figure 3.5. In such cases, S-tile and SOS can serve as complementary techniques; while S-tile improves cache sharing extensively, SOS reduces cache contention when necessary.

3.6.2 Data Reuse and Contention Reduction

Figure 3.4 shows S-tile's drastic improvement in data reuse. While I-tile(part) is able to improve cache reuse as well, its improvement falls far behind that of S-tile in many benchmarks due to the lack of temporal locality. As a result, cache blocks are more likely to

be replaced before they are reused again. Compared to I-tile(part), S-tile brings additional performance gains of 13% on an 8-way CMT and 30% on a 16-way CMT, as illustrated in Figure 3.3.

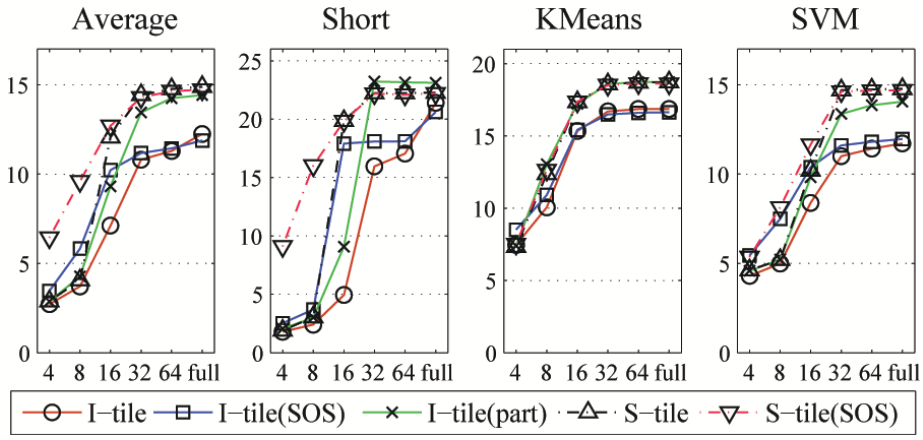
The effect of improved cache sharing is reflected on conflict and capacity misses. The D-cache associativity is varied from four-way associative to fully associative and the resulting performance is compared in Figure 6.14. As soon as the D-cache associativity increases to the number of symbiotic threads, S-tile's performance dramatically improves and reaches optimal, leaving I-tile behind. The performance gain with fully associative caches shows that S-tile saves capacity misses as well. On the other hand, while I-tile(part) reduces capacity misses in the case of fully associative caches, it suffers from conflict misses more than S-tile.

Cache sharing also leads to better storage utilization, and D-cache size is much less of a scaling bottleneck in S-tile than in I-tile. We compare the performance of I-tile and S-tile by varying the D-cache size from 4 KB to 128 KB. Results are shown in Figure 3.5b. With the exception of KMeans and SVM, all benchmarks show that I-tile's performance drops more drastically than S-tile's when we shrink the D-cache size. S-tile's reduced demand on D-cache size also indicates that the same D-cache can host more thread contexts in S-tile than in I-tile.

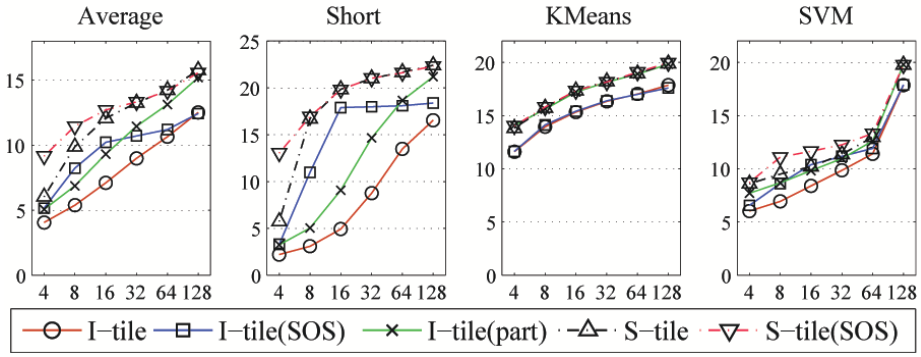
S-tile's improvement in storage utilization also leads to area efficiency and power savings. Several workloads show that S-tile's performance achieved at a D-cache size of 8 KB or 16 KB is similar to, or sometimes even better than, I-tile's with 128 KB D-caches. Using Cacti [89] to model the area cost, we show that a decrease in D-cache size from 128 KB to 8 KB saves 1.89 mm^2 for a system with 4 cores. On a system with 8 cores, this saving can account for another in-order core! In addition, several techniques provide opportunities to power down some cache segments [109, 110]. In fact, seven out of eight cache segments can be powered down with a 128 KB cache in S-tile to achieve the same performance as in

I-tile.

3.6.3 Scalability through Reduced L1 D-cache Footprints



(a) Speedup vs. D-cache associativity. The D-cache size is 16 KB.



(b) Normalized speedup vs. D-cache size (KB). The D-cache associativity is 16.

Figure 3.5: D-cache sensitivity study conducted over a 16-way CMT. Each core has a SIMT width of eight. In each configuration, the average speedup across all benchmarks is shown as `Average`.

S-tile achieves more scalable performance because it is able to accommodate a larger number of symbiotic threads with little increase in D-cache footprints. We breakdown D-cache misses into first misses (i.e. misses that allocate MSHRs and send data requests

to the lower level caches) and secondary misses (i.e. misses captured by MSHRs whose requests have already been sent to the lower level storage). We focus on the comparison of first misses since secondary misses are not part of the critical loop of memory accesses.

Figure 3.6 shows the number of D-cache first misses when the number of symbiotic threads increases for both I-tile and S-tile. With more threads per core, I-tile often experiences an explosion of D-cache first misses caused by contention, even though the D-cache associativity scales accordingly to the number of threads per core. This inevitably leads to cache thrashing. The number of D-cache misses are reduced by I-tile(part), however, it still keeps increasing with more threads.

On the other hand, the number of S-tile's D-cache misses remains relatively constant under the same scaling. In most benchmarks, the number of D-cache misses remains almost identical to that of single-threaded cores even with eight threads per core or more. It is also observed that S-tile alone is equally as effective as I-tile(SOS) in reducing the number of D-cache first misses, and this is achieved without decreasing the number of active threads. As a result, S-tile is able to scale parallel performance beyond other systems.

3.6.4 Sensitivity on Various Shared Cache Designs

Performance sensitivity on shared cache designs is also investigated. We scale the last level cache's (LLC's) associativity from 4 to 32 and show the resulting performance in Figure 3.7a. All systems show similar sensitivity to LLC associativity. S-tile does not improve LLC cache sharing by much because concurrent threads over different cores still operate on distinct tiles. Instead, SOS may be more effective in reducing LLC contention when the LLC's associativity is extremely small. Nevertheless, systems with S-tile always outperform their I-tile equivalent.

Both S-tile and I-tile suffer when the available LLC capacity is small, as illustrated in

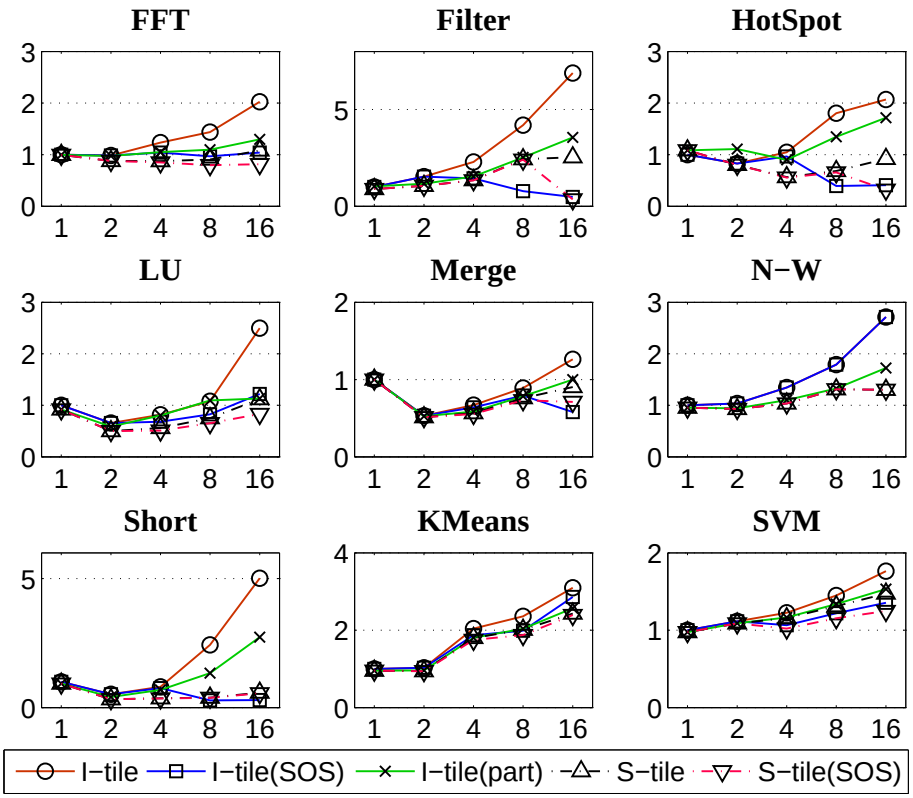
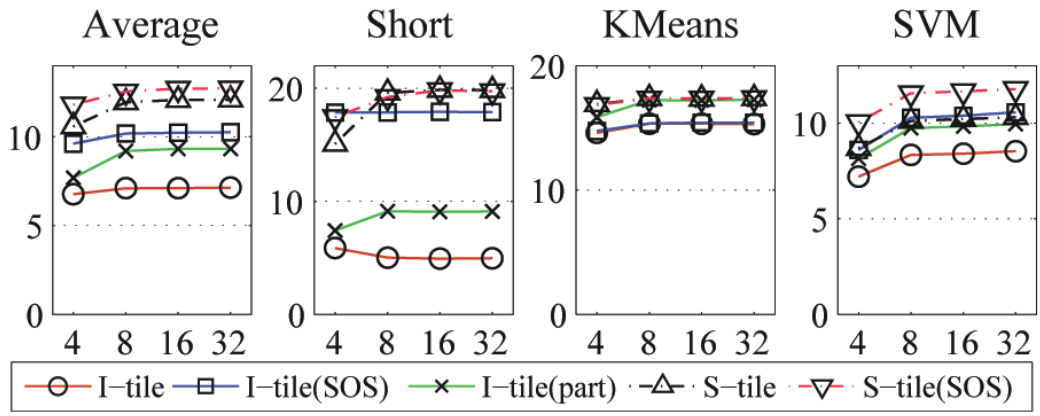
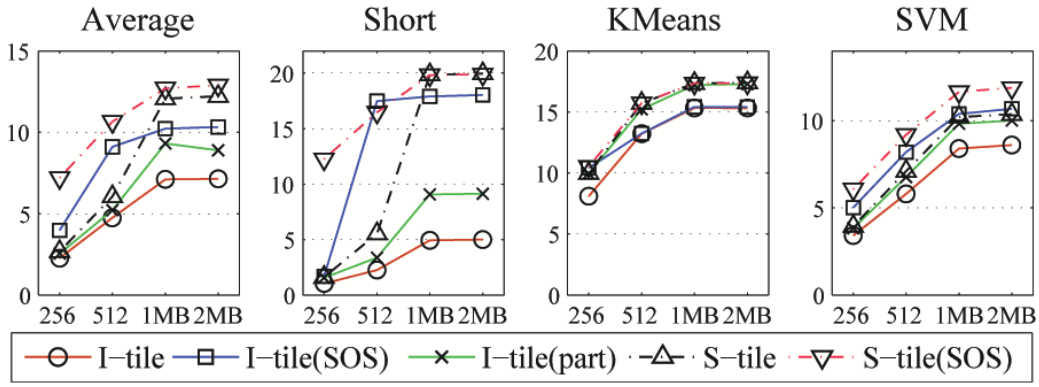


Figure 3.6: Number of D-cache first misses vs. Number of threads per core, measured with the same system configuration as Figure 3.3.

Figure 3.7b. In fact, S-tile's performance may even degrade to that of I-tile. In this scenario, both S-tile(SOS) and I-tile(SOS) perform significantly better than S-tile and I-tile, and S-tile(SOS) achieves the best speedup. With an LLC larger than 1024 KB, its capacity is no longer a scaling bottleneck, and performance starts to benefit more significantly from S-tile than from I-tile(SOS).



(a) Speedup vs. LLC associativity



(b) Speedup vs. LLC size (KB)

Figure 3.7: Sensitivity to the LLC cache design on a 16-way CMT. Each core has a SIMT width of eight. S-tile and I-tile respond similarly to different LLC cache designs. S-tile performs consistently better than I-tile. In each configuration, the average speedup across all benchmarks is shown as `Average`.

3.6.5 Applicability with Various Pipeline Configurations

To demonstrate the wide variety of multi-threaded cores S-tile can benefit, we vary the number of scalar pipelines per core, or the width of a thread group, from 1 to 16. The degree of multithreading, or the number of thread groups, is also varied from 1 to 16. Figure 3.8 illustrates the average speedup relative to the performance of 4 single-threaded cores over the memory system specified in Table 4.1, with D-cache associativities equal to

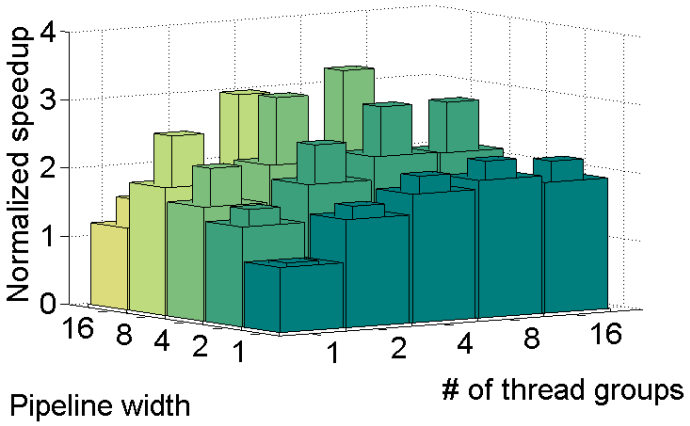


Figure 3.8: Performance comparison of S-tile and I-tile over various multi-threaded cores. Speedup is normalized to the execution time on four single-threaded cores. Narrower bars denote S-tile speedup and wider bars denote I-tile. Data is averaged across all benchmarks.

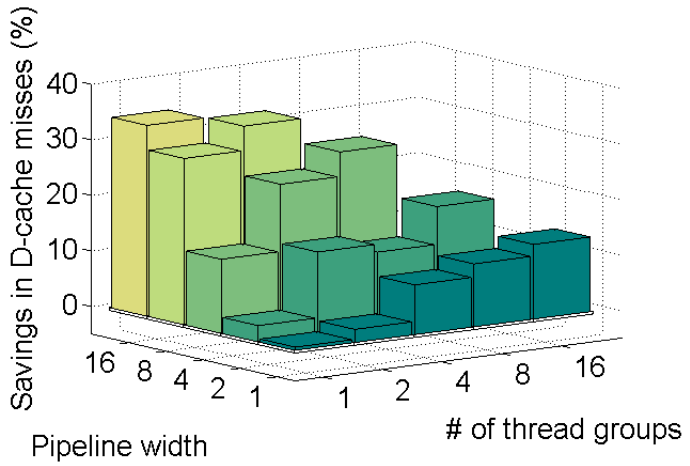


Figure 3.9: S-tile's average savings in D-cache misses compared to I-tile over various multithreaded cores. Data is averaged across all benchmarks.

the total number of threads per core. The maximum speedups for I-tile and S-tile are 2.39× and 3.54× respectively, and both are achieved with a moderate degree of multithreading

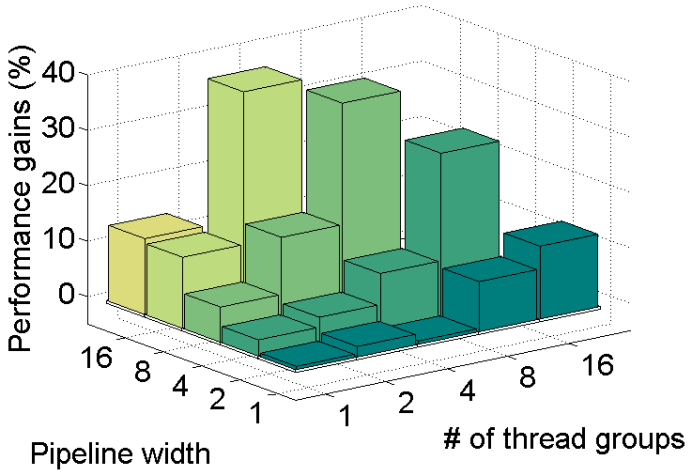


Figure 3.10: S-tile's average performance gains compared to I-tile(part). Data is averaged across all benchmarks.

and pipeline width; too few thread groups are not able to sufficiently hide memory latency, while too many thread groups may involve more threads than necessary and increase cache contention. On the other hand, since we only model pipelines that operate in SIMT lock-step, a wider pipeline is more likely to suffer from stalls due to cache misses by individual threads or from under-utilization caused by divergent branches.

The speedup of S-tile over I-tile under the same configuration is maximized with a moderate degree of multithreading and pipeline width as well. This phenomenon has to do with the interaction between D-cache contention and the effectiveness of latency hiding; the benefit of S-tile can be increased with more thread groups due to more D-cache contention, at the same time, it may also be obscured by the improved latency hiding. While I-tile's performance may get closer to S-tile with more effective latency hiding, Figure 3.9 demonstrates that S-tile's savings in D-cache misses always grows with the number of threads per core. On the other hand, S-tile does not always benefit more with a wider pipeline; if S-tile is not able to eliminate cache misses simultaneously for *all* threads within the same group,

a SIMT thread group has to stall anyway.

Compared to I-tile(part), S-tile's performance gains increase with a larger number of symbiotic threads, as shown in Figure 3.10. Similar to the comparison of I-tile, S-tile benefits most with a modest degree of multi-threading and pipeline width.

It is also important to note that S-tile does not only benefit cores with SIMT pipelines. Even for a single scalar pipeline that switches among multiple threads, S-tile can lead to 10% performance gains over I-tile as well as I-tile(part). This indicates that S-tile and latency hiding techniques are complementary.

3.6.6 Energy Savings

Since SAS promotes constructive cache sharing and significantly reduces the number of D-cache misses, it also significantly reduces the number of L2 cache lookups. As Figure 3.11 suggests, the energy budget on the L2 cache is reduced significantly using S-tile.

The performance gains also translate into lower leakage energy. Leakage accounts for a significant portion of total energy consumption at a technology node of 65nm or smaller. Therefore, execution time correlates closely with energy consumption. All benchmarks save energy with S-tile, with an average energy savings of 36% compared to I-tile and 17% compared to I-tile(SOS) and I-tile(part). We observe the same trend that programs with high-dimensional access patterns benefit more from symbiotic tiling in energy consumption.

3.7 Conclusions and Future Work

With emerging multithreaded cores, there has been some work on thread scheduling that reduces resource contention, but nothing that considers cache sharing among dozens or

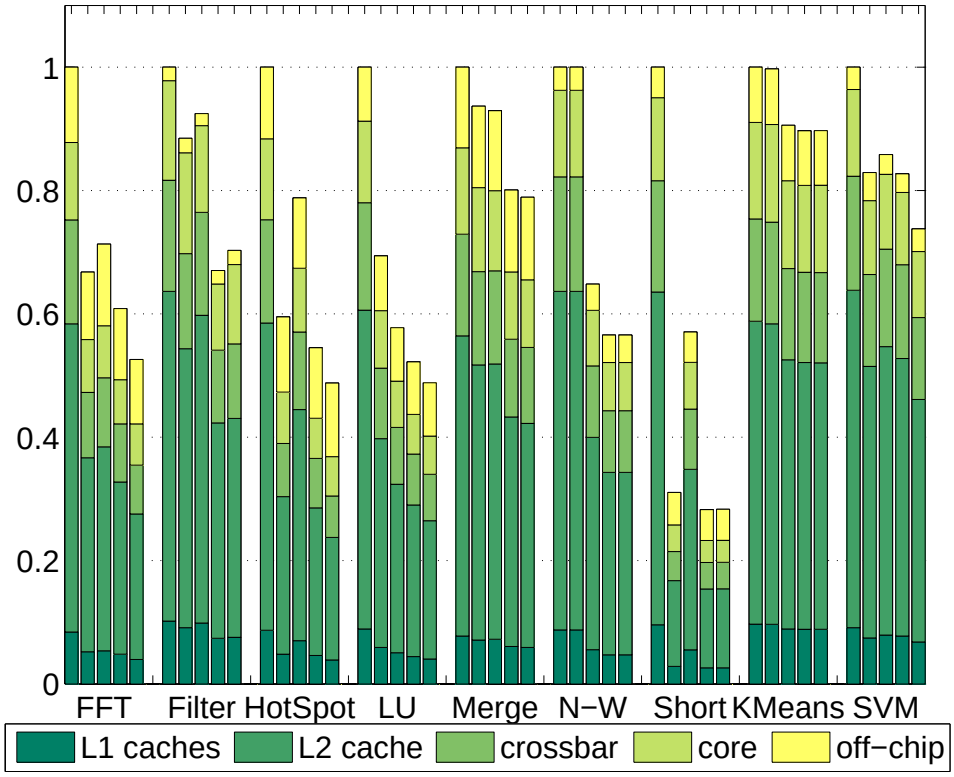


Figure 3.11: Energy consumption with four cores each with two SIMT groups of width eight. For each benchmark, five systems are shown and from left to right they are: I-tile, I-tile(SOS), I-tile(part), S-tile, and S-tile(SOS). Energy is normalized to each system's I-tile equivalent.

hundreds of concurrent threads sharing a common L1. Conventional data partitioning techniques can be adapted to subdivide data partitions among symbiotic threads for spatial locality; however, threads would still operate on disparate data and may suffer from conflict misses. The solution is to also exploit temporal locality among symbiotic threads. We propose symbiotic affinity scheduling (SAS) that partitions work according to the number of L1 caches, and then use all threads on that processing unit to work *simultaneously* on adjacent data in the same partition. This becomes particularly important for high-dimensional

data accesses because their strided accesses penalize data-locality. The scheduler demonstrates an average speedup of 1.69× and average energy savings of 33% on the data-parallel benchmarks we studied. Based on conventional cache-blocking techniques, the exploited temporal locality brings additional 30% performance gains. It also outperforms adaptive contention reduction techniques by 17%.

Combining SAS with temporal blocking or hierarchical tiling [21] is a natural extension for future work. In fact, in this case, SAS becomes *essential* for CMTs, otherwise independent threads sharing the same cache are not able to coordinate and work on potentially time-skewed tiles optimized for individual caches.

Future work also includes extending SAS to cover parallel operations over trees and graphs. It would also be useful to integrate SAS into existing parallel programming APIs such as OpenMP [101].

Chapter 4

Avoiding Cache Thrashing in Shared Cache

4.1 Introduction and Background

Processors optimized for throughput employ many small, multithreaded cores [95] and seem likely to scale up to large core and thread counts. They are sometimes referred to as chip multithreading (CMT). Examples include Sun's Niagara [8] and Intel's Larrabee [15]. At the same time, caches remain important for performance and they reduce off-chip traffic. Hardware cache coherence also remains important because it simplifies the task of writing parallel programs.

As the number of cores increases, a single broadcast medium cannot sustain the inter-processor communication bandwidth. This requires a shift to point-to-point on-chip networks (OCNs) as well as a directory protocol, because the broadcast operations required for snooping are not practical over a point-to-point OCN. The directory coherence, in turn, benefits from a shared, inclusive last level cache (LLC) for management of shared data.

This is because exclusive [111] or non-inclusive [54] caching both incur three-way communication (i.e. among data requester, directory, and owner) to locate shared data and they also introduce nontrivial design and verification complexity [112].

Inclusion poses a problem for large-scale, CMT organizations, because each LLC cache set is contended for by all threads. Multithreaded manycore chips risk excessive conflict misses over the shared, inclusive LLC. Inclusion requires that an eviction in the LLC also evict any copies in the L1s, so conflicts can lead to cache thrashing even for data in active use.

We have observed that one of the most important sources of thrashing is LLC contention by *private* data that need not reside in the LLC at all. In the rest of this chapter, we refer to data that are only accessed from one core as *private data*, and they are mostly composed of *thread-private* data such as stacks. All other data are referred to as *shared data*, which *may* be accessed from multiple cores. We find that current memory management practice typically distributes private data non-uniformly among the LLC cache sets, with wasteful and unnecessary LLC conflicts. In this chapter, we focus on the problem caused by conventional stack allocation mechanisms that tend to align stack bases to page boundaries, with the consequence that thread-private data are likely to concentrate on a few LLC cache sets.

To our knowledge, there has been no prior study that shows to what extent private data contend for the LLC and how they may affect the performance of a large scale CMT connected through a point-to-point OCN. While separate address spaces [113], locality-aware memory allocators [103, 114, 115] and task schedulers [42, 32, 39] can reduce coherence traffic, they do not address capacity or conflict misses in the LLC. Even based upon a locality-aware memory allocator, we show that with 16 eight-way multi-threaded cores in an inclusive organization, 5-10% of cache sets in a 16-way associative LLC are severely contended for by private data, raising conflict misses and unnecessary L1 evictions in an

inclusive organization. This may eventually lead to cache thrashing and jeopardize performance.

To reduce LLC conflicts and mitigate cache thrashing, we first propose a simple run-time stack allocation mechanism that randomizes the offset of the stack bases relative to page boundaries. Without any modification to the program nor the hardware, *stack randomization* distributes thread-private data more uniformly, and it alone improves performance by a factor of 2.7X on 32 cores compared to the best baseline configuration. This is different from address space randomization used for security reasons [116], which may randomize stack bases — not necessarily their offsets relative to page boundaries — to lower the chances that stack bases are easily predicted by an attacker.

We also compare different LLC replacement policies according to their ability to address the same issue without modifying the application or the run-time system. We observe that the LRU insertion policy (LIP) [117] barely mitigates the penalty of LLC conflicts in such scenario. We also investigate a modified LRU replacement policy, LRSU, that replaces shared data in a conflicting cache set before replacing any private data. This may seem counter-intuitive, but assumes that private data (such as stack variables) are likely to be used actively in the L1 caches. However, like LIP, LRSU also proves to be inferior.

Despite the potential benefit of stack randomization, it does not address the underlying hardware behavior: private data are used by a single core and if threads do not migrate, private data can be excluded from coherence as well as the inclusive hierarchy to make room for shared data in the LLC. Although researchers proposed dedicated storage for thread-private data [118, 119, 120, 121], such techniques run the risk of under- or over-utilization of the private storage. Using the MESI coherence protocol as a baseline design, we exclude private data—including both stacks and private heaps—from the inclusive coherence protocol. We refer to this as a *non-inclusive, semi-coherent (NISC)* cache organization. NISC allows private data to exist only in the L1 caches and private data evicted from LLC

need not invalidate their copies in the L1 cache. Results show NISC performs significantly better than cache replacement policies but stack randomization still performs the best for LLCs with sufficient capacity. However, NISC can be employed together with stack randomization, and we show that when the LLC does not have sufficient capacity to support inclusion, NISC's ability to reduce the number of lines contending for the LLC allows significant performance benefits relative to just stack randomization. The resulting ability to reduce the demand for LLC resources can be used to reduce silicon costs or to increase the number of cores, as well as accommodate more concurrent processes with large working sets. This work was published in [64].

4.2 Non-uniform Distribution of Private Data

We study a two-level memory hierarchy with private L1 caches and an LLC as shared, distributed L2 caches. Hsu et al. [47] underline the importance of shared LLC, and a great deal of work has explored ways to hide wire delay for distributed L2 caches [50, 52, 51]. However, they do not mitigate LLC contention caused by non-uniform distribution of data, which would arise regardless of whether the LLC is centralized or distributed. In the following discussion we focus on the non-uniform distribution of private data in the LLC, which is mainly caused by conventional stack allocation mechanisms that align stack bases to page boundaries.

As an example, consider an operating system (OS) with a page size of 8 KB and a cache hierarchy with a 1024 KB, 16-way associative LLC with 128 B cache lines. As Figure 4.1 illustrates, the LLC has 512 cache sets in total each with a 9-bit index, of which 6 bits are part of page offsets and only 3 bits are part of page numbers. Therefore, data located at page boundaries is clustered around cache sets with the indices of "xxx000000". As a result, private data compete for these cache sets more intensively than others. Specifically,

stack bases on page boundaries would compete for 8 cache sets or 128 cache lines; in other words, thrashing is likely to occur when the LLC is shared by 128 concurrent threads or more. We name this small subset of cache lines that lead to thrashing *critical lines*, and their number can be calculated as

$$\max(\frac{LLC\_size}{page\_size}, LLC\_associativity) \tag{4.1}$$

Usually, Equation 4.1 results in $\frac{LLC\_size}{page\_size}$ and it indicates that systems with larger page sizes or smaller LLCs have fewer critical lines and data distribution becomes less uniform. Even worse, critical lines also have to accommodate data other than stacks, and they are scattered in different cache sets which may not be evenly contended for. As a result, despite the increased number of critical lines, a larger LLC may only suffer less, but not avoid thrashing caused by private data. We further study this effect in Section 4.5.2.

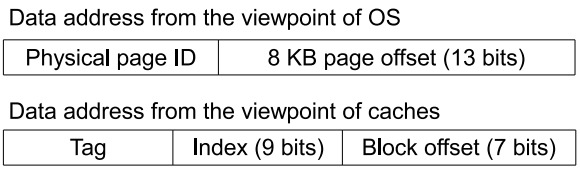
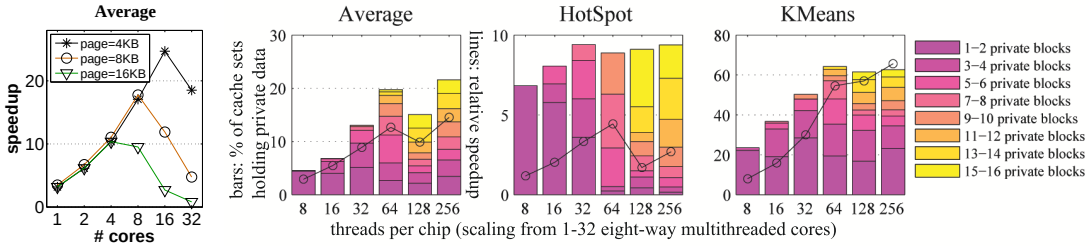


Figure 4.1: The OS segments the address space to pages while the hardware segments the address space into cache sets and blocks. Aligning stack bases to page boundaries introduces non-uniform distribution of private data. This is illustrated by OS with 8K pages and a 1024 KB, 16-way associative LLC with 128 B cache lines.

This issue of LLC contention caused by the non-uniform distribution of private data, to the best of our knowledge, has not been studied since the advent of the recent trend toward manycore CMT architectures. We characterize a spectrum of nine benchmarks (shown in Table 4.2 in Section 7.4.2) and simulate them on a multithreaded CMP that has 1–32 cores, with each providing 8 hardware thread contexts. For all benchmarks except



(a) (b)

Figure 4.2: (a) The average speedup across all benchmarks when we scale the number of cores from 1 to 32, each is 8-way threaded. The LLC is 1 MB and 16-way associative. Speedup is normalized to single-threaded execution. A larger page size leads to more performance degradation at a smaller number of cores. (b) Bars show the distribution of LLC cache sets with regard to private data occupancy, and lines illustrate the correlation with relative performance speedup. The LLC is 16-way associative, with a capacity of 1024 KB with 16 cores or fewer and 2048 KB with 32 cores. Each core is 8-way multi-threaded.

KMeans, private data consists of stack data only. KMeans also has private heaps that store partial sums during Map-Reduce computation. The effect of fewer critical lines is reflected in Figure 4.2a where a larger page size leads to performance degradation at smaller number of cores or threads.

With a page size of 8 KB, Figure 4.2b shows the occupancy of private data in a 1-2 MB shared, inclusive LLC that accommodates I- and D-caches of 16 KB each. With 16 cores and 128 thread contexts or more, 5-10% of the 16-way associative LLC cache sets have private data occupying more than half of their blocks. These cache sets are intensively contended and victims have to invalidate their D-cache copies as well. Victims in these cache sets are likely to be private data.

We further characterize our benchmarks and study the breakdown of their memory accesses using the infrastructure described in Section 4.3. For 16 cores with 128 thread contexts, private data account for less than 15% of the applications' working set, however, 51% of the memory accesses request private data. This reflects that private data are reused

much more often than shared data. On the other hand, 64% of D-cache misses occur to private data and *62% of these misses end up reloading an invalidated block*. This implies that evicting private data as victims of LLC conflicts introduces unnecessary invalidations and causes cache thrashing. Note that this phenomenon is only obvious when a large number of threads aggregate a large amount of private data that all compete for the inclusive LLC.

4.3 Experimental Setup

4.3.1 Simulation

Due to the unavailability of coherent, cache-based manycore products with large numbers of cores and threads, we simulate a future CMT processor using M5 [84], a cycle-accurate, event-driven simulator for networks of processors. We extend the simple, scalar core model to support fine-grained multithreading and add support for a 2D mesh OCN, directory-based coherence, and a shared, banked LLC. We name our large-scale CMT simulator *MV5*.

Table 4.1 summarizes the main parameters of the simulated manycore processor. We model throughput-oriented, multi-threaded in-order cores that are loosely inspired by Niagara-style cores [8]. Scalar threads are grouped into SIMD groups that share a common instruction sequencer as modern GPUs do [13]. Upon memory accesses, the SIMD group cannot proceed until all its threads' requests are fulfilled. In order to hide this memory latency, the core immediately switches to another SIMD group and continues execution with no extra latency by simply indexing into the appropriate register files. We acknowledge the limitation that our core model does not support non-blocking loads or other more aggressive techniques for exploiting memory-level parallelism (MLP), and that the interaction between MLP techniques and the phenomena we study is an interesting area for future

Table 4.1: Default System Configuration

Technology Node	65 nm
Cores	Alpha ISA, 2.0 GHz, 0.9V Vdd in order, 8-way multithreaded 2 SIMD thread groups each with 4 threads
L1 Caches	16 KB I-cache and 16 KB D-cache 16-way associative, 32 B line size, write-back physically indexed, physically tagged 4 banks, 16 MSHRs, 3 cycle hit latency
L2 Cache	$\geq$ 1024 KB, distributed 16 banks, up to 8 pending requests each 16-way associative, 128 B line size 32 cycle access latency, write-back physically indexed, physically tagged 64 MSHRs, each can hold up to 8 requests
Interconnect	2-D Mesh, wormhole routing 300 MHz, 57 Gbytes/s 1 cycle routing latency 1 cycle link latency per hop
Memory Bus	266 MHz, 16 GB/s
Main Memory	50 ns access latency

work.

The memory system is a two-level coherent cache hierarchy. Each core has a private I-cache and D-cache, which are connected to a shared LLC through a 2D mesh interconnect. The LLC is composed of distributed L2 cache banks, with the number of banks equal to the number of cores. L2 caches and cores are tiled in an interleaved way similar to a checkerboard. The total size of the LLC is 1024 KB with 16 cores or fewer and 2048 KB with 32 cores, note that LLC capacity is no smaller than twice the cumulative L1 cache capacity, as Hsu et al. [47] suggested. The LLC cache sets are statically partitioned across the L2 banks. L1 caches are kept coherent through an inclusive, directory-based MESI protocol.

We model cache latency according to Cacti [89] and work from Kim [122]. Pullini et al. provide the basis for our interconnect latency modeling [90]. For the in-order cores, instructions-per-cycle (IPC) is assumed to be one except for memory references, which are modeled faithfully through the memory hierarchy (although we do not model memory

controller effects).

4.3.2 Benchmarks

We select a set of emerging data-parallel applications ranging across image processing, scientific computing, physics simulation, machine learning and data mining. Our primary objective is to obtain representatives of different parallel program types. We choose the input size such that our benchmarks exhibit sufficient parallelism for evaluation while maintaining manageable simulation times. Table 4.2 summarizes the benchmarks. Benchmarks are cross-compiled to the Alpha ISA using GCC 4.1.0. Parallel `for` loops are annotated in OpenMP style and are interpreted by an emulated system call that later allocate memory spaces for stacks and spawn threads across cores. Data-parallel tasks are grouped into blocks according to the number of cores. Blocks are assigned to cores in a persistent order for the purpose of cache-affinity.

Our performance study uses a baseline that already employs a scalable locality-aware memory allocator that maintains per-core lists of private pages for private data [114, 115]. By doing so, we isolate our results from previous studies that aim to reduce coherence traffic.

4.4 Approaches to Reduce LLC Conflicts

From the aspects of both software and hardware, we propose and compare several solutions to reduce LLC conflicts and cache thrashing caused by non-uniform distribution of private data. We also evaluate existing techniques in their effectiveness to address the same issue.

Table 4.2: Simulated benchmarks with descriptions and input sizes.

	Benchmark Description
FFT	Fast Fourier Transform (Splash2 [97]) Spectral methods. Butterfly computation Input: a 1-D array of 32,768 (2^{15}) numbers
Filter	Edge Detection of an Input Image Convolution. Gathering a 3-by-3 neighborhood Input: a gray scale image of size 500×500
HotSpot	Thermal Simulation (Rodinia [20]) Iterative partial differential equation solver Input: a 300×300 2-D grid, 100 iterations
LU	LU Decomposition (Splash2 [97]). Dense linear algebra Alternating row-major and column-major computation Input: a 300×300 matrix
Merge	Merge Sort. Element aggregation and reordering Input: a 1-D array of 300,000 integers
N-W	Needleman-Wunsch DNA alignment. (Rodinia [20]) Dynamic programming by updating matrix with a diagonal wavefront Input is a 2-D array of size 512×512
Short	Winning Path Search for Chess. Dynamic programming. Neighborhood calculation based on the the previous row Input: 6 steps each with 150,000 choices
KMeans	Unsupervised Classification (MineBench [98]). Map-Reduce. Distance aggregation. Input: 10,000 points in a 20-D space
SVM	Supervised Learning (MineBench [98]) Support vector machine's kernel computation. Input: 1,024 vectors with a 20-D space

4.4.1 Randomly Offsetting Stack Bases

Since the non-uniform distribution of private data is mostly caused by stack bases that are usually aligned to page boundaries, we modify the thread library so that stack bases are now offset to random locations within a physical page. We still enforce the stack bases to be aligned to cache line boundaries in order to avoid misaligned data. In this way, stack may start with *any* cache lines and private data are likely to span evenly across all LLC cache sets. We do not modify the operating system's mechanism that maps virtual pages to physical pages.

No hardware modification is needed for this approach. However, this may introduce inefficient memory management in the OS since offsetting stack bases introduces fragmentation in the allocated physical pages. In the case where the stack base is offset to the tail

of a page, the fragment can be almost as large as an entire page. The issue of fragmentation warrants further investigation but space limitations preclude their study here.

4.4.2 LLC Replacement Policies

Randomizing the stack bases *avoids* conflict misses caused by private data in the LLC. Alternatively, the underlying architecture can be designed to *tolerate* the non-uniform distribution of private data with no need to modify existing stack allocation mechanisms. We extend the LLC's LRU replacement policy to evict shared data first. In this way, actively used private data in the L1s are not likely to be invalidated due to LLC conflicts. This policy is referred to as *LRSU* (least recently used shared data) replacement.

Alternatively, we also experiment LLCs with LIP (LRU Insertion Policy) [117]. Based on our observation that private data are reused more often than shared data (private data evicted from D-caches are likely to be reloaded), we postulate that these more valuable lines are more likely to reside closer to the MRU (most recently used) position. For shared data that are streamed into L1s for one-time access, their LLC cache blocks stay in the LRU position and get evicted during the next replacement, leaving private data intact in the LLC. However, in some circumstances, LIP may increase the chance of private data to thrash the LLC — for those private data that remain in the L1, it is possible that their LLC copies are not reused and remain in the LRU position; these private data are likely to subject to LLC replacement.

4.4.3 Non-Inclusive, Semi-Coherent Cache Hierarchy

Cache replacement policies, however, do not address the fundamental issue: private data compete for LLC space due to the enforcement of inclusion property, however they require no coherence and therefore do not benefit from inclusion property's savings of coherence

overhead. By excluding private data from the inclusive coherence protocol, private data can be well accommodated in D-caches and the LLC can host more shared data. We call this a Non-Inclusive, Semi-Coherent cache organization (NISC). As Figure 4.3 illustrates, NISC allows private data to exist only in the L1 cache—replaced private data in LLC do not have to invalidate their copies in the L1 cache. As a result, private data contend less for LLC cache sets, reducing LLC conflict misses. Moreover, the overall on-chip storage is utilized better because of the reduced data replicates, and this reduces capacity misses as well.

NISC is compatible with *any* inclusive coherence protocols, and we demonstrate how NISC works with a simple MESI coherence protocol [123] for a two-level write-back cache hierarchy. An additional *coherence toggle bit (CTB)* is added to each cache line's state. The CTB marks whether the cache line stores shared or private data so that coherence is toggled on and off for that cache line, respectively. The CTB can be determined in several ways:

- Option 1 (OS approach): Utilizing run-time scalable memory allocators which maintain a list of per-core private pages and shared pages [114, 115]. Private pages can mark their TLB entries as private using an additional bit per TLB entry, whose value can be inherited by the corresponding cache line's CTB.

- Option 2 (Compiler approach with extended ISA): Using implicit static analysis or explicit code annotations to identify private data. The compiler then uses a different set of memory access instructions specifically for accessing private data. For example, the ATI instruction set [124] and NVIDIA's parallel thread extension (PTX) [125] already use a different set of memory instructions for private data. These private-data oriented memory instructions set the corresponding cache lines' CTBs. For example, stack data can be assumed to be thread-private in most cases (although not guaranteed). On the other hand, private data may be easily identified and annotated by

programmers, as demonstrated in several parallel programming languages and APIs such as OpenMP [102] and CUDA [78].

Any access to those cache lines with their CTBs set bypasses the MESI coherence protocol entirely and is managed through a non-inclusive, non-coherent protocol. Accesses to other cache lines proceed with the original MESI protocol. In this way, NISC manages private data and shared data distinctly while balancing them in the same storage hierarchy.

Care has to be taken for NISC upon thread migration. Since coherence is disabled for cache lines with CTBs set, these lines have to be flushed to the shared cache and reloaded to the destination L1 cache that will host the thread after its migration. Since we are dealing with data-parallel workloads with abundant parallelism, we assume workload can be mostly balanced and thread migration need not take place frequently. Nevertheless, context switches may still occur occasionally due to page faults or interrupts, and flushing the cache serves as a solution in such circumstances.

4.5 Evaluation

The following configurations are compared and their sensitivity to the last level cache design is studied as well.

- *conv*: The conventional system with stack bases aligned to page boundaries and the memory system is maintained by an inclusive MESI coherence protocol with LRU replacement policy.
- *randStack*: The OS offsets the stack base to some random distance away from the page boundary. The underlying hardware is unmodified.
- *LRSU*: The OS is unmodified. The LLC employs LRSU replacement policy that attempts to hold as much private data as possible.

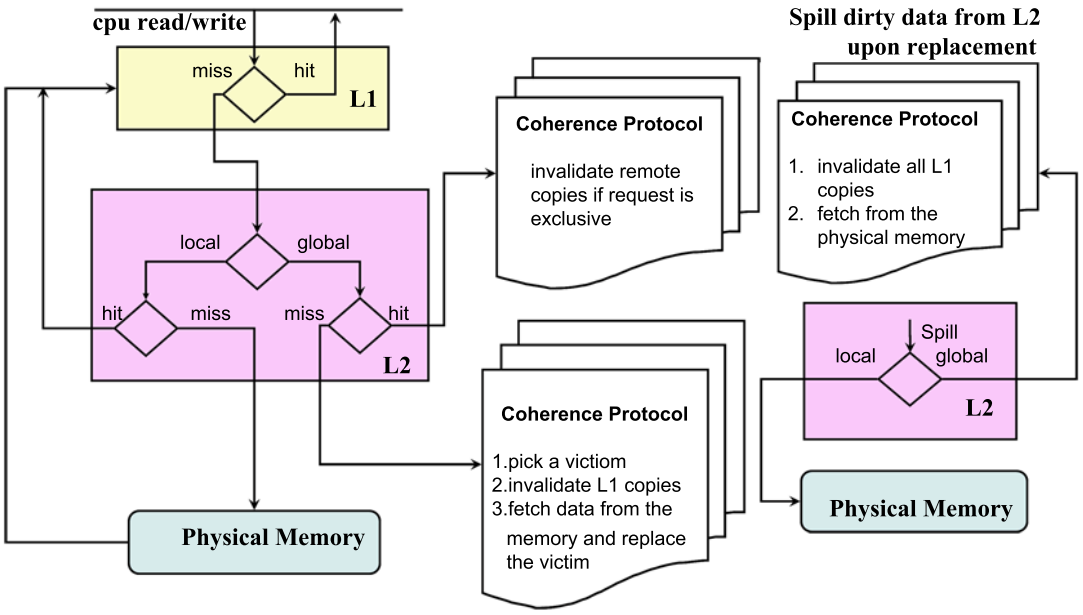


Figure 4.3: The NISC protocol based on MESI. Except for requests and spillings for private data, all other data movement exhibits the same behavior as in MESI. Note that blocks holding thread-private data should not have copies in remote L1s and therefore are always writable.

- *LIP*: The OS is unmodified. The LLC employs LIP that first replaces lines that are not reused.
- *NISC*: The OS is unmodified except it signals private data to the hardware which sets the CTB for corresponding cache lines and excludes them from the inclusive coherence.
- *NISC+randStack*: The OS randomizes stack bases as in *randStack* and it operates over an architecture with NISC cache organization.

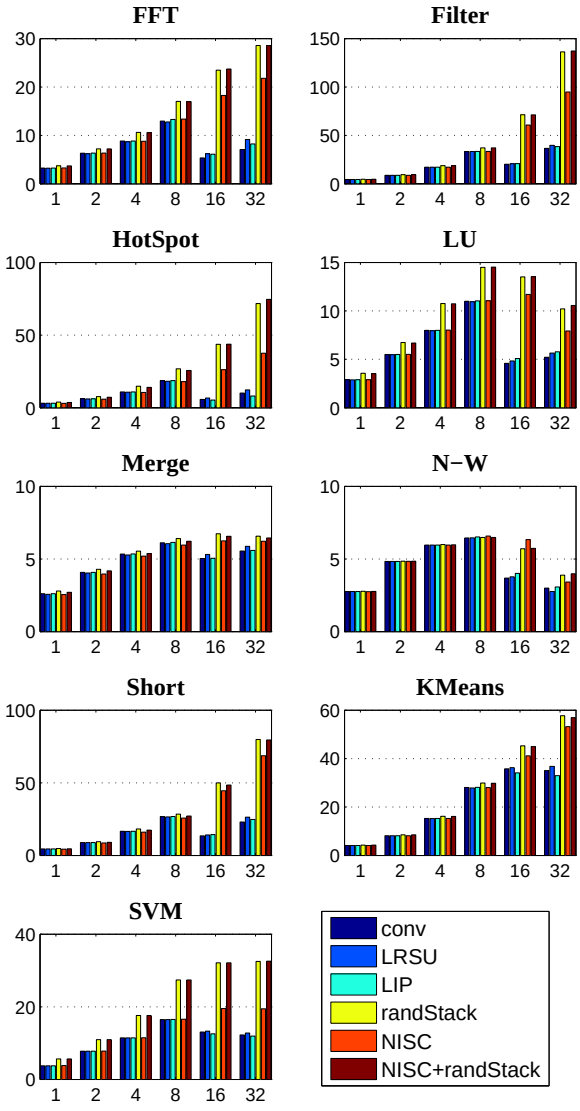


Figure 4.4: Speedup vs. Number of eight-way multithreaded cores. Speedup is normalized to single-threaded execution. LLC size is no smaller than twice the aggregate size of L1 caches, therefore it is 2048 KB in the case of 32 cores and 1024 KB otherwise.

4.5.1 Performance Scaling

We increase the number of cores from 1 to 32 and the total number of thread contexts from 8 to 256, and speedup is calculated based on single threaded execution. These results are

shown in Figure 4.4. Performances of *conv*, *LIP* and *LRSU* start to degrade beyond 8 cores. Note that we we double the LLC size to 2 MB with 32 cores to ensure the LLC size is no smaller than twice the aggregate size of L1 caches. However, performance with 32 cores is still much worse than 8 cores with *conv*. In order to eliminate partial accesses to LLC lines which may reduce the effectiveness of LIP, we also experiment an LLC with a line size equal to that of the L1 line size, however, performance scaling for LIP remains similar. *LIP* and *LRSU* fail to improve performance mainly because they do not reduce the amount of data that contend for the same LLC cache sets. We also observe that applications with more private data competing for the LLC degrades more severely, as we illustrate in Figure 4.2b.

On the other hand, performance of *randStack*, *NISC* and *NISC+randStack* continues to scale. Comparing to the peak performance of *conv* at 8 cores, the middleware approach, *randStack*, achieves average speedups of 1.2X at 8 cores, 1.8X at 16 cores, and 2.7X at 32 cores. The average speedups for the hardware approach, *NISC*, are 1.0X, 1.5X and 2.0X, respectively. Their combination, *NISC+randStack*, performs similarly to *randStack*. We also evaluate the combinations of *randStack* with *LIP* and *LRSU*, and they perform similarly to *randStack* as well. Therefore, provided with stack randomization and sufficient LLC capacity, there is no need to further employ hardware approaches to address the non-uniform distribution of private data.

4.5.2 LLC Sensitivity

If the LLC is simply too small, techniques such as stack randomization that only address conflicts will be insufficient. The LLC may not have sufficient capacity for programs with large working sets, or when trading off LLC area to include more cores within a given die budget. NISC is attractive here because it reduces the capacity requirement of the LLC by relaxing inclusion for private data. On the other hand, if this is merely a capacity issue,

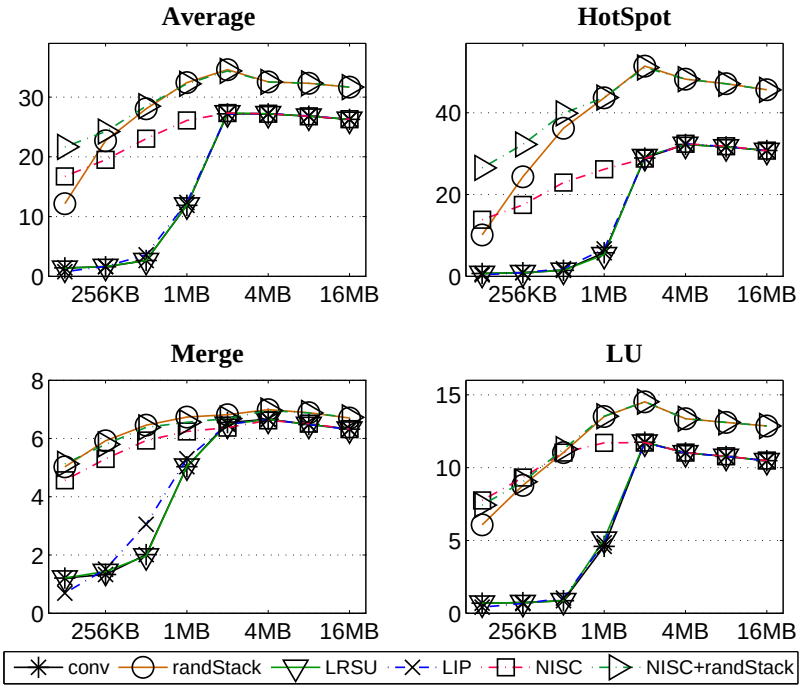


Figure 4.5: Speedup over single-threaded execution vs. LLC sizes. Speedup is measured on a system with 16 cores and 128 thread contexts, and is calculated relative to the performance of a single-threaded execution. LLC associativity is 16.

then randStack is not expected to benefit large enough LLC.

As we observe from Figure 4.5 when scaling the LLC size up to 16 MB over 16 cores, *the benefit of* randStack *is not limited to small LLC sizes*. With smaller LLCs, *conv*, *LIP* and *LRSU* degrade drastically, and *randStack* degrades more gracefully. With an LLC larger than 256 KB, *randStack*, as a technique that *avoids* LLC conflicts caused by private data, out-performs *NISC*, which merely attempts to *tolerate* the LLC conflicts. However, when the LLC is smaller than 256 KB, capacity misses dominate and this is only addressed by *NISC*. The best performance with a small LLC is achieved by combining these two approaches — in *NISC+randStack*, NISC further improves the performance of *randStack* by a factor of 1.8 with a small LLC capacity of 128 KB. This shows that the software

approach (stack randomization) and the hardware approach (NISC) are complementary. We also experiment with LIP and LRSU when combined with stack randomization. However, they do not further improve performance over *randStack*.

Similar benefit of *NISC+randStack* is observed with smaller LLC associativity, as shown in Figure 4.6. Besides, Equation 4.1 indicates that larger associativity is not likely to increase the number of critical lines, but only helps distributing private data more evenly across cache sets that host critical lines. Therefore, we observe that the conventional performance improves significantly, but still cannot match that of *randStack* with an LLC associativity of 512.

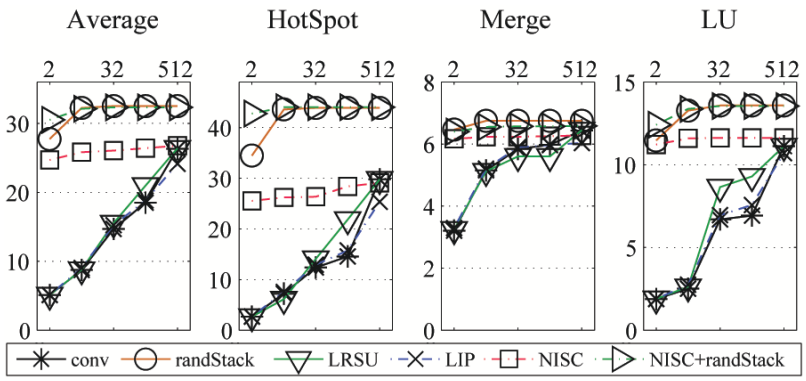


Figure 4.6: Speedup over single-threaded execution vs. LLC associativity. Speedup is measured on a system with 16 cores and 128 thread contexts, and is calculated relative to the performance of a single-threaded execution. The size of the LLC is 1024 KB.

4.6 Conclusions and Future Work

In the presence of a large number of concurrent threads, even a small amount of private data per core can swamp a shared inclusive LLC. Worse yet, LLC evictions due to this

contention then evict private data in the L1s. However, a shared, *inclusive* LLC is valuable in directory-coherence.

We study the performance impact of non-uniform distribution of private data caused by conventional stack allocation mechanisms. Several solutions are proposed. Our data-parallel benchmarks show that these limitations prevent scaling beyond 8 cores, while our techniques allow scaling to at least 32 cores for most benchmarks. At 8 cores, stack randomization provides a mean speedup of 1.2X, and stack randomization with 32 cores gives a speedup of 2.7X over the best baseline configuration. However, stack randomization may introduce fragmentation in the physical pages. We therefore investigate several hardware approaches including different LLC replacement policies and a non-inclusive, semi-coherent (NISC) cache organization that excludes private data from cache coherence. While LLC replacement policies fail to improve performance significantly, NISC alone scales performance up to at least 32 cores in most cases. Comparing to the best baseline configuration at 8 cores, NISC provides a mean speedup of 1.5X at 16 cores and 2.0X at 32 cores.

In cases where the LLC may have insufficient capacity for programs with large working sets, or when trading off LLC area to include more cores within a given die budget, combining NISC with stack randomization yields the best performance. With a limited LLC capacity of 128KB, NISC further improves the performance of *randStack* by a factor of 1.8. NISC results with small LLC suggest an interesting avenue for further research in reducing LLC area and allowing that to be used for other purposes.

Chapter 5

Replicate Computation to Reduce Communication

5.1 Introduction

Iterative stencil loops (ISL) [66] are widely used in image processing, data mining, and physical simulations. ISLs usually operate on multi-dimensional arrays, with each element computed as a function of some neighboring elements. These neighbors comprise the *stencil*. Multiple iterations across the array are usually required to achieve convergence and/or to simulate multiple time steps. Tiling [25][30] is often used to partition the stencil loops among multiple processing elements (PEs) for parallel execution, and we refer to a workload partition as a *tile* in this chapter. Similar tiling techniques also help localize computation to optimize cache hit rate for an individual processor [24].

Tiling across multiple PEs introduces a problem because stencils along the boundary

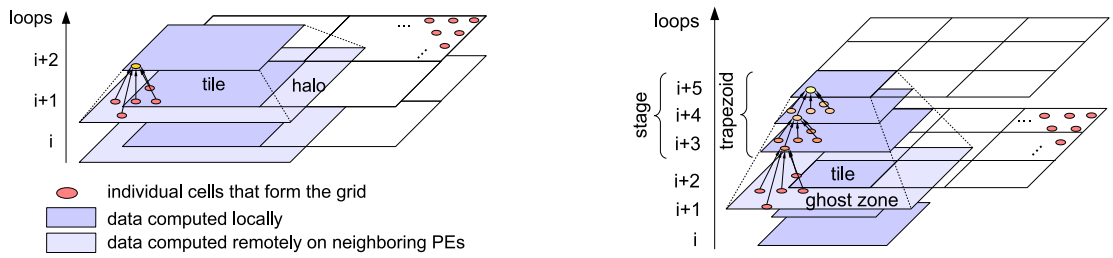


Figure 5.1: (a) Iterative stencil loops and halo regions. (b) Ghost zones help reduce inter-loop communication.

of a tile must obtain values that were computed remotely on other PEs, as shown in Figure 5.1a. This means that ISL algorithms may spend considerable time stalled due to inter-loop communication and synchronization delays to exchange these *halo* regions. Instead of incurring this overhead after every iteration, a tile can be enlarged to include a *ghost zone*. This ghost zone enlarges the tile with a perimeter overlapping neighboring tiles by multiple halo regions, as shown in Figure 5.1b. The overlap allows each PE to generate its halo regions locally [126] for a number of iterations proportional to the size of the ghost zone. As Figure 5.1b demonstrates, ghost zones group loops into *stages*, where each stage operates on overlapping stacks of tiles, which we refer to as *trapezoids*. Trapezoids still produce non-overlapping data at the end, and their height reflects the ghost zone size.

Ghost zones pose a tradeoff between the cost of redundant computation and the reduction in communication and synchronization among PEs. This tradeoff remains poorly understood. Despite the ghost zone's potential benefit, an improper selection of the ghost zone size may negatively impact the overall performance. Previously, optimization techniques for ghost zones have only been proposed in message-passing based distributed environments [126]. These techniques no longer fit for modern chip multiprocessors (CMPs) for two reasons. First, communication in CMPs is usually based on shared memory and its latency model is different from that of message-passing systems. Secondly, the optimal ghost

zone size is commonly one on distributed environments [127], allowing for a reasonably good initial guess for adaptive methods. However, this assumption may not hold on some shared memory CMPs, as demonstrated by our experimental results later in the chapter. As a result, the overhead of the adaptive selection method may even undermine performance. This chapter presents the first technique that we know of to automatically select the optimal ghost zone size for ISL applications executing on a shared-memory multicore chip multiprocessor (CMP). It is based on an analytical model for optimizing performance in the presence of this tradeoff between the costs of communication and synchronization versus the costs of redundant computation.

As a case study for exploring these tradeoffs in manycore CMPs, we base our analysis on the architectural considerations posed by NVIDIA's Tesla GPU architecture [128]. We choose GPUs because they contain so many cores and some extra complexity regarding the L1 storage and global synchronization. In particular, the use of larger ghost zones is especially valuable in the Tesla architecture because global synchronization is conventionally performed by restarting threads; unlike in the distributed environment where a tile's data may persist in its PE's local memory, data in all PEs' local memory has to be flushed to the globally shared memory and reloaded again after the inter-loop synchronization. Even though the memory fence is later introduced into GPU programming that allows global synchronization without restarting threads, its overhead increases along with the input size, as we will show in Section 5.4.5.

Our performance model and its optimizations are validated by four diverse CUDA applications consisting of dynamic programming, an ordinary differential equation (ODE) solver, a partial differential equation (PDE) solver, and a cellular automaton. The optimized ghost zone sizes are able to achieve a speedup no less than 95% of the optimal configuration. Our performance model for local-store based memory systems can be extended for cache hierarchies given appropriate memory modeling such as that proposed by

Kamil et al [129].

Our performance model can adapt to various ISL applications. In particular, we find that ghost zones benefit more for ISLs with narrower halo widths, lower computation/communication ratios, and stencils operating on lower-dimensional neighborhood. Moreover, although the Tesla architecture limits the size of thread blocks, our performance model predicts that the speedup from ghost zones tends to grow with larger tiles or thread blocks.

Finally, we propose a framework template to automate the implementation of the ghost zone technique for ISL programs in CUDA. It uses explicit code annotation and implicitly transforms the code to that with ghost zones. It then performs a one-time profiling for the target application with a small input size. The measurement is then used to estimate the optimal ghost zone configuration, which is valid across different input sizes.

In short, the main contributions of this work are an analytical method for deriving an ISL's performance as a function of its ghost zone, a gradient descent optimizer for optimizing the ghost zone size, and a method for the programmer to briefly annotate conventional ISL code to automate finding and implementing the optimal ghost zone. This work was published in [67].

5.2 Related Work

Tiling is a well-known technique to optimize ISL applications [25][130][106][30][131]. In some circumstances, compilers are able to tile stencil iterations to localize computation or/and exploit parallelism [132][24][133]. Some attempt to reduce redundant computation for specific stencil operations [134]. On the other hand, APIs such as OpenMP are able to tile stencil loops at run-time and execute the tiles in parallel [102]. Renganarayana et al. explored the best combination of tiling strategies that optimizes both cache locality and parallelism [135]. Researchers have also investigated automatic tuning for tiling stencil

computations [22][66]. Specifically, posynomials have been widely used in tile size selection [136]. However, these techniques do not consider the ghost zone technique that reduces the inter-tile communication. Although loop fusion [137] and time skewing [138] are able to generate tiles that can execute concurrently with improved locality, they cannot eliminate the communication between concurrent tiles if more than one stencil loops are fused into one tile. This enforces bulk-synchronous systems, such as NVIDIA's Tesla architecture, to frequently synchronizes computation among different PEs, which eventually penalizes performance.

Ghost zones are based on tiling and they reduce communication further by replicating computation, whose purpose is to replicate and distribute data to where it is consumed [139]. Krishnamoorthy et al. proposed overlapped tiling that employs ghost zones with time skewing and they studied its effect in reducing the communication volume [140]. However, their static analysis does not consider latencies at run-time and therefore the optimal ghost zone size cannot be determined to balance the benefit of reduced communication and the overhead of redundant computation.

Ripeanu et al. constructed a performance model that can predict the optimal ghost zone size [126], and they conclude the optimal ghost zone size is usually one in distributed environments. However, the performance model is based on message-passing and it does not model shared memory systems. Moreover, their technique is not able to make case-by-case optimizations — it predicts the time spent in parallel computation using time measurement of the sequential execution. This obscures the benefit of optimization — an even longer sequential execution is required for every different input size even it is the same application running on the same platform.

Alternatively, Allen et al. proposed adaptive selection of ghost zone sizes which sets the ghost zone size to be one initially and increases or decreases it dynamically according

to run-time performance measurement [127]. The technique works fine in distributed environments because the initial guess of the ghost zone size is usually correct or close to the optimal. However, our experiments on NVIDIA's Tesla architecture show that the optimal ghost zone size varies significantly for different applications or even different tile sizes. Therefore an inaccurate initial guess may lead to long adaptation overhead or even performance degradation, as demonstrated in Section 5.5.4. Moreover, the implementation of the adaptive technique is application-specific and it requires nontrivial programming effort.

The concept of computation replication involved in ghost zones is related to data replication and distribution in the context of distributed memory systems [141][142], which are used to wisely distribute *pre-existing* data across processor memories. Communication-free partitioning has been proposed for multiprocessors as a compiling technique based on hyperplane, however, it only covers a narrow class of stencil loops [143]. To study the performance of 3-D stencil computations on modern cache-based memory systems, another performance model is proposed by Kamil et al. [129] which is used to analyze the effect of cache blocking optimizations. Their model does not consider ghost zone optimizations.

An implementation of ghost zones in CUDA programming is described by Che et al. [144]. The same technique can be used in other existing CUDA applications ranging from fluid dynamics [145] to image processing [146].

Another automatic tuning framework for CUDA programs is CUDA-lite [147]. It uses code annotation to help programmers select what memory units to use and transfer data among different memory units. While CUDA-lite performs general optimizations and generates code with good performance, it does not consider the trapezoid technique which serves as an ISL-specific optimization.

5.3 Ghost zones on GPUs

We show an example of ISLs and illustrate how ghost zones are optimized in distributed environments. We then introduce CUDA programming and NVIDIA's Tesla architecture and show how ghost zones are implemented in a different system platform.

5.3.1 Ghost Zones in Distributed Environments

Listing 5.1: Simplified HotSpot code as an example of iterative stencil loops

```
/* A and B are two 2-D ROWS x COLS arrays      *
 * B points to the array with values produced *
 * previously, and A points to the array with *
 * values to be computed.                     */
float **A, **B;
/* iteratively update the array               */
for k = 0 : num_iterations
    /*  in each iteration, array elements are *
     *  updated with a stencil in parallel    */
    for_all i = 0 : ROWS-1 and j = 0 : COLS-1
        /* define indices of the stencil and  *
         * handle boundary conditions by      *
         * clamping overflown indices to      *
         * array boundaries                   */
        top = max(i-1, 0);
        bottom = min(i+1, ROWS-1);
        left = max(j-1, 0);
        right = min(j+1, COLS-1);
        /* compute the new value using the    *
         * stencil (neighborhood elements     *
         * produced in the previous iteration) */
        A[i][j] = B[i][j] + B[top][j]       \
                + B[bottom][j] + B[i][left]  \
                + B[i][right];
    swap(A, B);
```

Listing 5.1 shows a simple example of ISLs without ghost zones. A 2-D array is updated iteratively and in each loop, values are computed using stencils that include data elements in the upper, lower, left and right positions. Computing boundary data, however, may require values outside of the array range. In this case, the values' array indices are clamped to the

boundary of the array dimensions. When parallelized in a distributed environment, each tile has to exchange with its neighboring tiles the halo regions, which is comprised of one row or column in each direction. Using ghost zones, multiple rows and columns are fetched and they are used to compute halo regions locally for subsequent loops. For example, if we wish to compute an $N \times N$ tile for two consecutive loops without communicating among PEs, each PE should start with a $(N + 4) \times (N + 4)$ data cells that overlaps each neighbor by $2N$ cells. At the end of one iteration it will have computed locally (and redundantly) the halo region that would normally need to be fetched from its neighbors, and the outermost cells will be invalid. The remaining $(N + 2) \times (N + 2)$ valid cells are used for the next iteration, producing a result with $N \times N$ valid cells.

5.3.2 CUDA and the Tesla Architecture

To study the effect of ghost zones on large-scale shared memory CMPs, we program several ISL applications in CUDA, a new language and development environment from NVIDIA that allows execution of general purpose applications with thousands of data-parallel threads on NVIDIA's Tesla architecture. CUDA abstracts the GPU hardware using a few simple abstractions [128]. As Figure 5.2 shows, the hardware model is comprised of several streaming multiprocessors (SMs), all sharing the same device memory (the global memory on the GPU card). Each of these SMs consists of a set of scalar processing elements (SPs) operating in SIMD lockstep fashion as an array processor. In the Tesla architecture, each SM consists of 8 SPs, but CUDA treats the SIMD width or "warp size" as 32. Each warp of 32 threads is therefore quad-pumped onto the 8 SPs. Each SM is also deeply multithreaded, supporting at least 512 concurrent threads, with fine-grained, zero-cycle context-switching among warps to hide memory latency and other sources of stalls.

In CUDA programming, a *kernel function* implements a parallel loop by mapping the function across all points in the array. In general, a separate thread is created for each point, generating thousands or millions of fine-grained threads. The threads are further grouped into a grid of *thread blocks*, where each thread block consists of at most 512 threads, and each thread block is assigned to a single SM and executes without preemption. Because the number of thread blocks may exceed (often drastically) the number of SMs, thread blocks are mapped onto SMs as preceding thread blocks finish. This allows the same program and grid to run on GPUs of different sizes or generations.

The CUDA virtual machine specifies that the order of execution of thread blocks within a single kernel call is undefined. This means that communication between thread blocks is not allowed within a single kernel call. Communication among thread blocks can only occur through the device memory, and the relaxed memory consistency model means that a global synchronization is required to guarantee the completion of these memory operations. However, the threads within a single thread block are guaranteed to run on the same SM and share a 16 KB software controlled local store or scratchpad. This has gone by various names in the NVIDIA literature but the best name appears to be *per-block shared memory* or PBSM. Data must be explicitly loaded into the PBSM or stored to the device memory.

5.3.3 Implementing Ghost Zones in CUDA

Without ghost zones nor memory fences, a thread block in CUDA can only compute one stencil loop because gathering data produced by another thread block requires the kernel function to store the computed data to the device memory, restart itself, and reload the data again. Different from the case in distributed environments where each tile only has to fetch halo regions, all data in PBSM is flushed and all has to be reloaded again. Even after CUDA 2.2 introduced the memory fence that allows global synchronization without

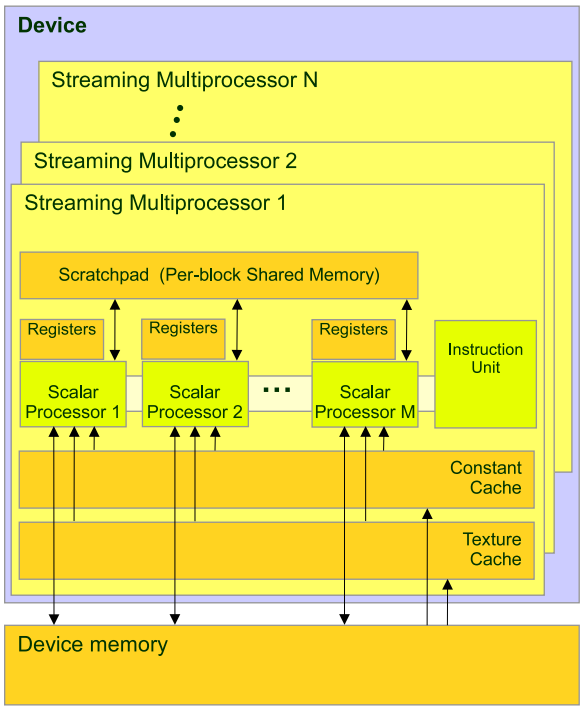


Figure 5.2: CUDA's shared memory architecture. Courtesy of NVIDIA.

restarting kernels, the overhead in such synchronization increases along with the input size, as we will show in Section 5.4.5. Moreover, thread blocks often contend for the device memory bandwidth. Therefore, the penalty of inter-loop communication is especially large. Using ghost zones, a thread block is able to compute an entire trapezoid that spans several loops without inter-loop communication. At least three alternative implementations are possible.

First, as the tile size decreases loop after loop in each trapezoid, only the stencil operations within the valid tile are performed. However, the boundary of the valid tile has to be calculated for each iteration, which increases the amount of computation and leads to more control-flow divergence within warps. It eventually undermines SIMD performance.

Alternatively, a trapezoid can be computed as if its tiles do not shrink along with the

loops. At the end, only those elements that fall within the boundary of the shrunk tile are committed to the device memory. This method avoids frequent boundary-testing at the expense of unnecessary stencil operations performed outside the shrunk tiles. Nevertheless, experiments show that this method performs best among all, and we base our study upon this method although we can model other methods equally well.

Finally, the Tesla architecture imposes a limit on the thread block size, which in turn limits the size of a tile if one thread operates on one data element. To allow larger tiles for larger trapezoids, a CUDA program can be coded in a way that one thread computes multiple data elements. However, this complicates the code significantly and experiments show that the increased number of instructions cancels out the benefit of ghost zones and this method performs the worst among all.

5.4 Modeling Methodology

We build a performance model in order to analyze the benefits and limitations of ghost zones used in CUDA. It is established as a series of a multivariate equations and it can demonstrate the sensitivity of different variables. The model is validated using four diverse ISL programs with various input data.

5.4.1 Performance Modeling for Trapezoids on CUDA

The performance modeling has to adapt to application-specific configurations including the shape of input data and halo regions. For stencil operations over a D-dimensional array, we denote its length in the i^{th} dimension as $DataLength_i$. The width of the halo region is defined as the number of neighborhood elements to gather along the i^{th} dimension of the stencil, and is denoted by $HaloWidth_i$, which is usually the length of the stencil minus

one. In the case of code in Listing 5.1, $HaloWidth$ in both dimensions are set to two. The halo width, together with the number of loops within each stage and the thread block size, determines the trapezoid's slope, height (h), and the size of the tile that it starts with, respectively. We simplify the model by assuming the common case where the thread block is chosen to be isotropic and its length is constant in all dimensions, denoted as blk_len. The width of the ghost zone is determined by the trapezoid height as $HaloWidth_i \times h$. We use the average cycles per loop (CPL) as the metric of ISL performance. Since a trapezoid spans across h loops which form a stage, the CPL can be calculated as the cycles per stage (CPS) divided by the trapezoid's height.

$$CPL = \frac{CPS}{h} \tag{5.1}$$

CPS is comprised of cycles spent in the computation of all trapezoids in one stage plus the global synchronization overhead ($GlbSync$). Trapezoids are executed in the form of thread blocks whose execution do not interfere with each other except for device memory accesses; when multiple memory requests are issued in bulks by multiple thread blocks, the requests are queued in the device memory and a thread block may have to wait for requests from other thread blocks to complete before it continues. Therefore, the latency in the device memory accesses ($MemAcc$) needs to consider the joint effect of memory requests from all thread blocks, rather than be regarded as part of the parallel execution. Let CPT (computing cycles per trapezoid) be the number of cycles for an SM to compute a single trapezoid assuming instantaneous device memory accesses, T be the the number of trapezoids in each stage, and M be the number of multiprocessors, we have:

$$CPS = GlbSync + MemAcc + CPT \times \frac{T}{M} \tag{5.2}$$

Where the number of trapezoids can be approximated by dividing the total number of

data elements with the size of non-overlapping tiles with which trapezoids end.

$$T = \frac{\prod_{i=0}^{D-1} DataLength_i}{\prod_{i=0}^{D-1} (blk\_len - HaloWidth_i \times h)} \tag{5.3}$$

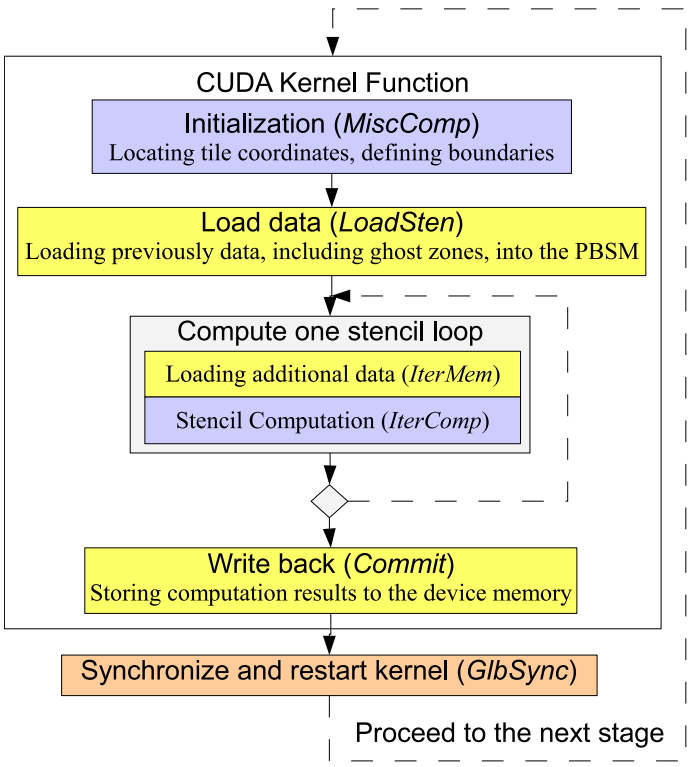


Figure 5.3: Abstraction of CUDA implementation for ISLs with ghost zones.

Furthermore, ISLs, when implemented in CUDA, usually take several steps described in Figure 5.3. By default, the global synchronization between stages are implemented by terminating and restarting the kernel. As it shows, latencies spent in device memory accesses ($MemAcc$) are additively composed of three parts namely:

- $LoadSten$: cycles for loading the ghost zone into the PBSM to compute the first tile

in the trapezoid.

- $IterMem$: cycles for other device memory accesses involved in each stencil operation for all the loops.

- $Commit$: cycles for storing data back to the device memory.

CPT, the cycles spent in a trapezoid's parallel computation, is additively comprised of two parts as well:

- $IterComp$: cycles for computation involved in a trapezoid's stencil loops.

- $MiscComp$: cycles for other computation that is performed only once in each thread. This mainly involves the computation for setting up the thread coordination and handling boundary conditions.

We now discuss the modeling of these components individually. The measuring of $GlbSync$ is described as well.

5.4.2 Memory Transfers

Due to the SIMD nature of the execution, threads from half a warp coalesce their memory accesses into memory requests of 16 words ($CoalesceDegree = 16$). Since concurrent thread blocks execute the same code, they tend to issue their memory requests in rapid succession and the requests are likely to arrive at the device memory in bulks. Therefore, our analytical model assumes that all memory requests from concurrent blocks are queued up. The device memory is optimized for bandwidth: for the GeForce GTX 280 model, it has eight 64-bit channels and can reach a peak bandwidth of 141.7 GBytes/sec [18]. The number of cycles to process one memory request with p bytes is

$$CyclesPerReq(p) = \frac{p}{MemBandwidth \div ClockRate} \tag{5.4}$$

Because the Tesla architecture allows each SM to have up to eight concurrent thread blocks ($BlksPerSM = 8$), the total number of concurrent thread blocks is

$$ConcurBlks = BlksPerSM \times M \tag{5.5}$$

With $ConcurBlks$ thread blocks executing in parallel, $\frac{T}{ConcurBlks}$ passes necessary for T thread blocks to complete their memory accesses. To estimate the number of cycles spent to access n data elements of x bytes in the device memory, we have:

$$\begin{aligned} passes &= \frac{T}{ConcurBlks} && (5.6) \\ MemCycles(n) &= passes \times [UncontendedLat + \alpha \times \\ &\quad \frac{n \times CyclesPerReq(x \times CoalesceDegree)}{passes \times CoalesceDegree}] \\ &= passes \times UncontendedLat + \alpha \times \frac{n \times x \times ClockRate}{MemBandwidth} && (5.7) \end{aligned}$$

where $UncontendedLat$ is the number of cycles needed for a memory request to travel to and from the device memory. It is assumed to be 300 cycles in this chapter, however, later studies show its value does not impact the predicted performance as significantly as the memory bandwidth does, given large data sets.

Besides the peak bandwidth, the memory overhead is also affected by bank conflicts, which depend on the execution of particular applications. Therefore, to compensate for the effect on the memory access overhead, we introduce an artificial factor, $\alpha = \sigma^{D-1}$, where σ is empirically set to 5.0 to best approximate the experimental results. We assume accessing higher dimensional arrays leads to more bank conflicts and lower memory throughput. Further experiments show that the overall performance trend predicted by our analytical performance model is not sensitive to the choice of σ.

For a trapezoid over a D-dimensional thread block with a size of blk_len^D, it typically loads blk_len^D data elements including the ghost zone. Usually, only one array is gathered for stencil operations ($NumStencilArrays = 1$), although in some rare cases multiple arrays are loaded. After loading the array(s), each tile processes
$\prod_{i=1}^{D}(blk\_len - HaloWidth_i)$ elements. Zero or more data elements
($NumElemPerOp \geq 0$) can be loaded from the device memory for each stencil operation, whose overhead is include in the model as $IterMem$. Because our implementation computes a trapezoid by performing the same number of stencil operations in each loop and only committing the valid values at the end (Section 5.3.3), the number of additional elements to load in each loop remains constant. Finally, the number of elements for each trapezoid to store to the device memory is $\prod_{i=0}^{D-1}(blk\_len - HaloWidth_i \times h)$. We summarize these components as:

$$LoadSten = NumStencilArrays \times MemCycles(T \times blk\_len^D) \quad (5.8)$$

$$Commit = MemCycles(T \times \prod_{i=0}^{D-1}(blk\_len - HaloWidth_i \times h)) \quad (5.9)$$

$$IterMem = NumElemPerOp \times h \times MemCycles(T \times \prod_{i=0}^{D-1}(blk\_len - HaloWidth_i)) \quad (5.10)$$

5.4.3 Computation

We estimate the number of instructions to predict the number of computation cycles that a thread block spends other than accessing the device memory. Because threads are executed in SIMD, instruction counts are based on warp execution (not thread execution!). We set the cycles-per-instruction (CPI) to four because in Tesla, a single instruction is executed by a warp of 32 threads distributed across eight SPs, each takes four pipeline stages to

complete the same instruction from four threads. Reads and writes to the PBSM are treated the same as other computation because accessing the PBSM usually takes the same time as accessing registers.

In addition, the performance model also has to take into account the effect of latency hiding and bank conflicts. Latency hiding overlaps computation with memory accesses; therefore the overall execution time is not the direct sum of cycles spent in memory accesses and computation. We define the effectiveness of latency hiding, β, as the number of instructions that overlap with memory accesses divided by the total number of instructions. Due to the lack of measurement tools, we approximate the effect of latency hiding by assuming half of the instructions overlaps with memory accesses; therefore β equals 0.5. With regard to the overall execution time, the effect of latency hiding is the same as that of reducing the number of warp instructions by a factor of β.

On the other hand, a bank conflict serializes SIMD memory instructions and it has the same latency effect as increasing the number of instructions by a factor of $BankConflictRate$, which equals the number of bank conflicts divided by the number of warp instructions, both can be obtained from the CUDA Profiler [148].

The overall effect of latency hiding and bank conflicts on computation cycles is as if the number of instructions is scaled with a factor of τ, which equals $(1 - \beta) \times (1 + BankConflictRate)$. After all, the computation cycles for a thread block can be determined as

$$CompCycles = StageInstsPerWarp \times \tau \times ActiveWarpsPerBlock \times CPI \quad (5.11)$$

where $StageInstsPerWarp$ is the number of instructions executed by an individual warp during a trapezoid stage, and $ActiveWarpsPerBlock$ is the number of warps that have one

or more running threads not suspended by branch divergence. While $ActiveWarpsPerBlock$ can be deduced from the tile size, $StageInstsPerWarp$ has to be synthesized for different trapezoid heights. We breakdown $StageInstsPerWarp$ into two additive parts: those that are performed once for each trapezoid ($StageInstsPerWarp_{MC}$), which correspond to $MiscComp$, and those that are performed iteratively in all stencil loops ($StageInstsPerWarp_{IC}$), which correspond to $IterComp$. $StageInstsPerWarp_{MC}$ and $StageInstsPerWarp_{IC}$ are estimated separately as:

$$StageInstsPerWarp_{MC} = IterInstsPerWarp_{MC} \tag{5.12}$$

$$StageInstsPerWarp_{IC} = h \times IterInstsPerWarp_{IC} \tag{5.13}$$

where $IterInstsPerWarp$ stands for the number of instructions executed by an individual warp during a conventional iteration. Its two components are $IterInstsPerWarp_{MC}$, which corresponds to $MiscComp$, and $IterInstsPerWarp_{IC}$, which corresponds to $IterComp$. With a trapezoid of height h, computation involved in $IterComp$ is repeated for h iterations, therefore the number of instructions per iteration is multiplied by h to estimate the number of instructions per stage in Equation 5.13. We then use the CUDA Profiler [148] to estimate $IterInstsPerWarp_{MC}$ and $IterInstsPerWarp_{IC}$ for each application using a small data set sized N with no ghost zones applied ($h = 1$). The recorded numbers, $InstsPerSM_{MC}$ and $InstsPerSM_{IC}$, respectively, are the numbers of instructions per iteration executed by all the warps on the same multiprocessor. Each type of $InstsPerSM$ relates to their corresponding $IterInstsPerWarp$ as:

$$IterInstsPerWarp = \frac{InstsPerSM}{WarpsPerSM} = InstsPerSM \div \frac{N}{WarpSize \times M} \tag{5.14}$$

$InstsPerSM_{MC}$ and $InstsPerSM_{IC}$ are measured separately by commenting out different sections of code and rerun the CUDA Profiler. Note that the resulting values of $IterInstsPerWarp_{MC}$ and $IterInstsPerWarp_{IC}$ are independent of the input size and the block size; once obtained, they can be used for arbitrary inputs and block sizes as long as the underlying architecture stays the same.

In Equation 5.11, $ActiveWarpsPerBlock$ is an integer value where a warp without all threads active should still be counted as one. We approximate it with floating point numbers so that it can be represented as the number of active threads divided by the the warp size. It is therefore estimated as $\frac{blk\_len^D}{WarpSize}$ for $MiscComp$ and $\frac{\prod_{j=0}^{D-1}(blk\_len - HaloWidth_j)}{WarpSize}$ for $IterComp$. Substituting these expressions into Equation 5.11, we obtain the computation cycles for a thread block as the sum of two terms:

$$MiscComp = \frac{InstsPerSM_{MC} \times M}{N} \times blk\_len^D \times CPI \tag{5.15}$$

$$IterComp = h \times \frac{InstsPerSM_{IC} \times M}{N} \times \prod_{j=0}^{D-1}(blk\_len - HaloWidth_j) \times CPI \tag{5.16}$$

5.4.4 Global Synchronization

To exchange the halo region, a thread block needs only synchronize with its neighbors. In CUDA, the conventional way to perform global synchronization is to terminate and restart the kernel function. Due to the lack of publicly available technical details, we assume that the exposed overhead of global synchronization is the time to terminate and restart concurrent blocks, whose quantity is calculated in Equation 5.5. The time spent for restarting other thread blocks is hidden by the computation of concurrent blocks.

In the case of GTX 280, there are 240 concurrent thread blocks. We therefore measured the time spent in restarting a kernel function with 240 empty thread blocks, which averages at 3350 cycles.

$$GlbSync_{KnlRestart} = 3350 \tag{5.17}$$

5.4.5 Extension: Modeling Memory Fence

While the global synchronization is traditionally implemented by restarting a CUDA kernel, the introduction of memory fences since CUDA 2.2 brings an alternative. With memory fences, a single CUDA kernel can persist across stages until all iterations are finished, as described in Figure 5.4. This leads to two changes in our performance model. First, $MiscComp$ — the time spent in setting up coordinates and handling boundary conditions — only needs to be counted once for all iterations. Given a large number of iterations, the cycles spent in $MiscComp$ can be negligible and it is no longer a component of CPT. Secondly, the modeling of the overheads in global synchronization has to be adjusted.

We apply the following method to implement global synchronization using memory fences. After synchronizing threads in the same thread block at the end of each stage and using memory fences to make sure their device memory accesses have committed, the first thread in each thread block atomically increments a counter in the device memory. The thread then uses a spinlock which repeatedly reads the counter's value until all thread blocks complete the memory fence. Eventually, it allows the belonging thread block to proceed to the next stage after the counter tells that all thread blocks have committed their partial results into the device memory. We use a different counter for *each* stage. Otherwise, if the same counter is reused across all stages, it has to be reset at the end of each stage after all thread blocks release their spinlock, which requires another round of atomic operations

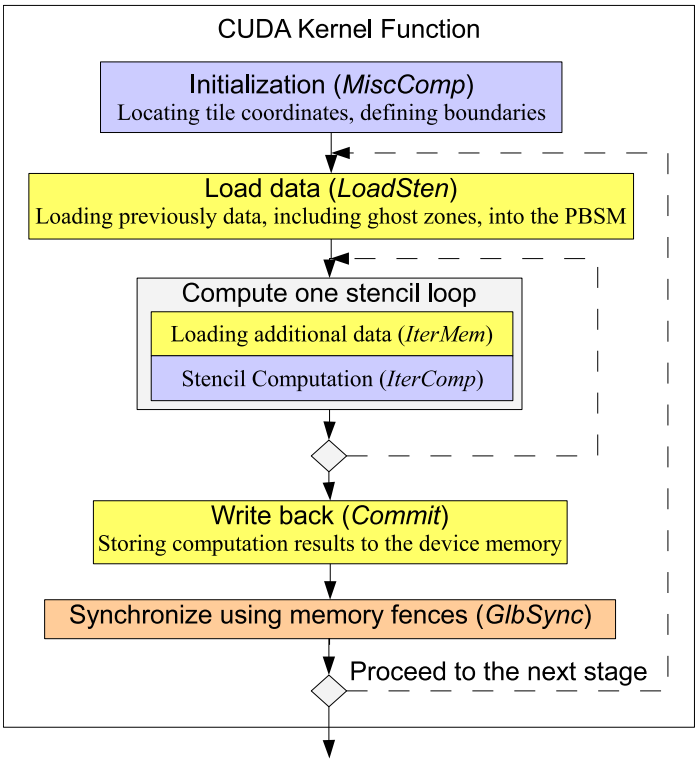


Figure 5.4: Abstraction of CUDA implementation for ISLs with ghost zones.

and memory fences.

Experiments show that the overhead in such synchronization correlates to the number of thread blocks. By measuring the execution time of a kernel function that performs nothing but global synchronization with memory fences, and by varying the number of thread blocks, we obtain the empirical function using linear curve fitting with 95% confidence: $GlbSync_{MemFence} = 210.3 \times \mathrm{T} + \delta$, where δ is negligible. In reality, the latency in global synchronization can overlap with computation from other thread blocks. To compensate for such effect, we introduce an artificial factor μ and assume about half of the time spent in global synchronization overlaps with computation ($\mu = 0.5$). Therefore, we have

$$GlbSync_{MemFence} = \mu \times 210.3 \times T \tag{5.18}$$

The number of kernel instructions used for implementing the memory fence is not included in the profiled instruction count of $InstsPerSM$. In fact, both $InstsPerSM_{IC}$ and $InstsPerSM_{MC}$ stay the same as that in the implementation without memory fences. Nevertheless, $InstsPerSM_{MC}$ is no longer needed because the overheads in $MiscComp$ is negligible in the implementation with memory fences.

5.4.6 Extensibility to Other Platforms

Our performance model focuses on shared memory CMPs with local store based memory system. Therefore, it can be extended to model the Cell Broadband Engine (CBE) [9]. Specifically in CBE, a tile needs not flush its data to the globally shared memory in order to communicate with others. Moreover, the model needs to consider the effect of latency hiding in the light of direct memory access (DMA) operations.

Given an appropriate memory latency model, the performance model can also be generalized to systems with cache hierarchies. Several additional factors need to be modeled including caching efficiency and the effect of prefetching. One candidate cache latency model for ISLs is Kamil et al.'s Stencil Probe [129] which does not take into account the effects of ghost zones yet.

5.5 Experiments

We validate our performance model using four distinct ISL applications that fall in the categories of dynamic programming, ODE and PDE solvers, and cellular automata. We compare their performance predicted by the model with their actual performance on the

GeForce GTX 280 graphics card, whose architectural parameters are summarized in Table 5.1 and are used in our performance model for optimizations. These parameters are retrieved using device query and some are obtained literally through the GTX 280 specifications [18]. We then use the performance model to select the optimal trapezoid height and evaluate its accuracy.

Parameter	Symbol	Value
GPU clock rate	$ClockRate$	1.3 GHz
(assumed) device memory access latency	$UncontendedLat$	300 cycles
coalesce width	-	16
concurrent blocks per SM	$BlksPerSM$	8
warp size	-	32
number of SPs per SM	-	8
number of SMs	M	30
average CPI	CPI	4
memory bandwidth	$MemBandwidth$	141.7 GBytes/sec
maximum number of threads per block	-	512
maximum memory pitch	-	262144 bytes

Table 5.1: Architectural parameters for GeForce GTX 280. Parameters not directly used in our performance model is marked by the symbol of "-". Although the warp size is not directly used in the equations, it is accounted for in the measurement of $InstPerSM$, which counts instructions based on warp execution; with constant workload, varying the warp size will also vary the profiled value of $InstPerSM$. The coalesce width is implicitly reflected as well in the memory bandwidth; a smaller coalesce width would undermine the effective memory bandwidth. Note that in GTX 280, memory requests from a half-warp are *always* coalesced.

While the major workload is carried out by the GPU, the benchmarks are launched on a host machine with an Intel Core2 Extreme CPU X9770 with a clock rate of 3.2 GHz. The CPU connects to the GPU through NVIDIA's MCP55 PCI bridge. The actual time measurement only includes the computation performed on the GPU.

5.5.1 Benchmarks

Our benchmark suite contains four distinct ISL applications programmed in CUDA. They have different dimensionality, computation intensities, and memory access intensities.

- *PathFinder* uses dynamic programming to find a path on a 2-D grid from the bottom row to the top row with the smallest accumulated weights, where each step of the path moves straight ahead or diagonally ahead. It iterates row by row, each node picks a neighboring node in the previous row that has the smallest accumulated weight, and adds its own weight to the sum.

- *HotSpot* [104] is a widely used tool to estimate processor temperature. A silicon die is partitioned into functional blocks based on a floorplan, and the simulation solves a ODE iteratively, where new temperature in each block is recalculated based on its neighborhood temperatures in the previous time step.

- *Poisson* numerically solves the poisson equation [149] which is a PDE widely used in electrostatics and fluid dynamics. The solver iterates until convergence, using stencils to calculate the Laplace operator over a grid.

- Cell is a cellular automaton used in Game of Life [150]. In each iteration, each cell, labeled as either live or dead, counts the number of live cells in its neighborhood, and determines whether it will be live or dead in the next time step.

The benchmark-specific parameters used in our performance model are listed in Table 7.2.

	PathFinder	HotSpot	Poisson	Cell
stencil dimensionality	1	2	2	3
stencil size	3	3×3	3×3	$3 \times 3 \times 3$
halo width	2	2×2	2×2	$2 \times 2 \times 2$
$NumStencilArrays$	1	2	1	1
$NumElemPerOp$	1	0	0	0
Profiling Input Size(N)	100,000	500×500	500×500	$60 \times 60 \times 60$
$InstsPerSM_{MC}$	1998	13488	12825	71603
$InstsPerSM_{IC}$	1859	16645	12474	220521
$BankConflictRate$0	0	0	1.1	

Table 5.2: Benchmark parameters used in our performance modeling.

5.5.2 Model Validation

We verify our performance model by varying the configurations and input sizes for each benchmark and comparing the experimental and theoretical results. The performance is measured in CPU cycles and is then normalized to GPU cycles. The measured execution time is then compared with the time predicted by the performance model.

Figure 5.5 compares the performance generated from actual experiments and that predicted from our analytical model. In this illustration, global synchronization is carried out by restarting kernel functions. We increase the size of the thread block size of PathFinder, HotSpot, Poisson and Cell from 64 to 512, 10×10 to 22×22, 10×10 to 22×22, and $6 \times 6 \times 6$ to $8 \times 8 \times 8$, respectively. The performance model captures the overall scaling trend for all benchmarks except for Cell, where performance is predicted to improve with a larger thread block size but it actually degrades with a block size of $8 \times 8 \times 8$. Further characterizations using the CUDA Profiler show this unexpected trend is due to a large number of bank conflicts with this particular thread block size. However, run-time dynamics of bank conflicts are not modeled faithfully in our analytical model.

In the experiment shown in Figure 5.6, we increase the input size of PathFinder, HotSpot, Poisson and Cell from $100,000$ to $1,000,000$, 500×500 to $2,000 \times 2,000$, 500×500 to $2,000 \times 2,000$, and $40 \times 40 \times 40$ to $100 \times 100 \times 100$, respectively. In this illustration, global

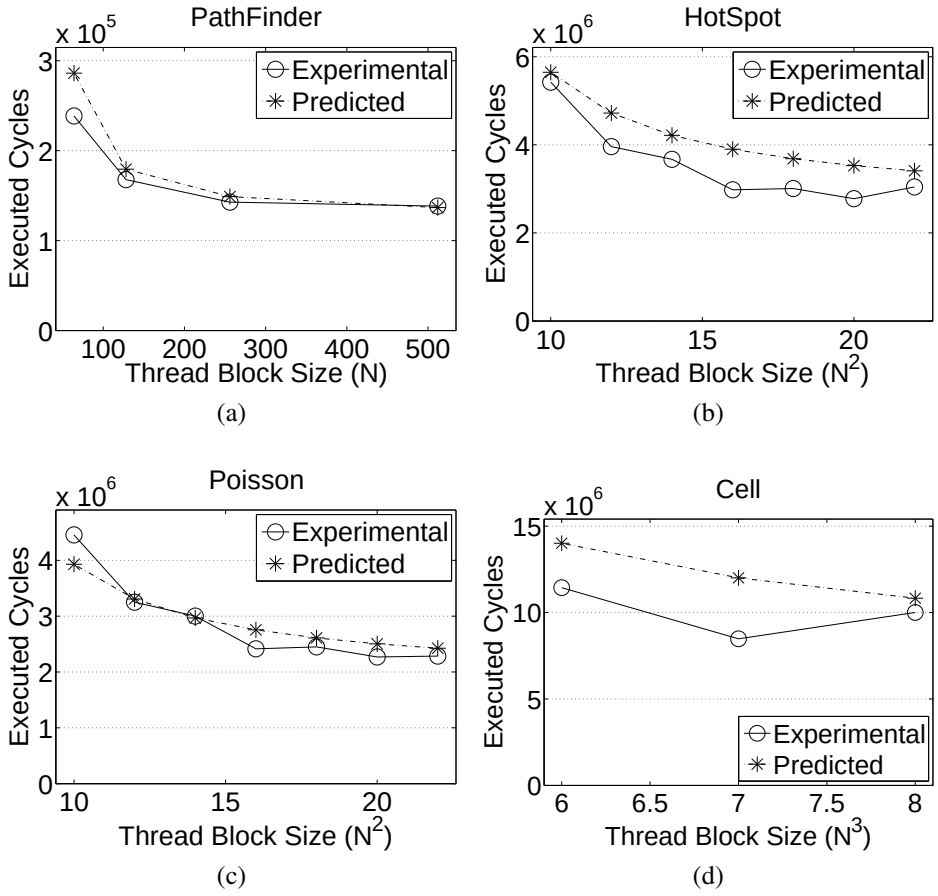


Figure 5.5: Model verification by scaling the thread block size of (a) PathFinder, (b) HotSpot, (c) Poisson, and (d) Cell. The trapezoid height is set to 16 for PathFinder, two for HotSpot and Poisson, and one for Cell. The input size is set to 1,000,000 for PathFinder, $2,000 \times 2,000$ for HotSpot and Poisson, and $100 \times 100 \times 100$ for Cell. The examples shown here perform global synchronization by restarting kernels; trends are similar with memory fences. The performance degradation in Cell with a block size of $8 \times 8 \times 8$ results from bank conflicts.

synchronization is performed using memory fences. The performance model captures the overall scaling trend for all benchmarks.

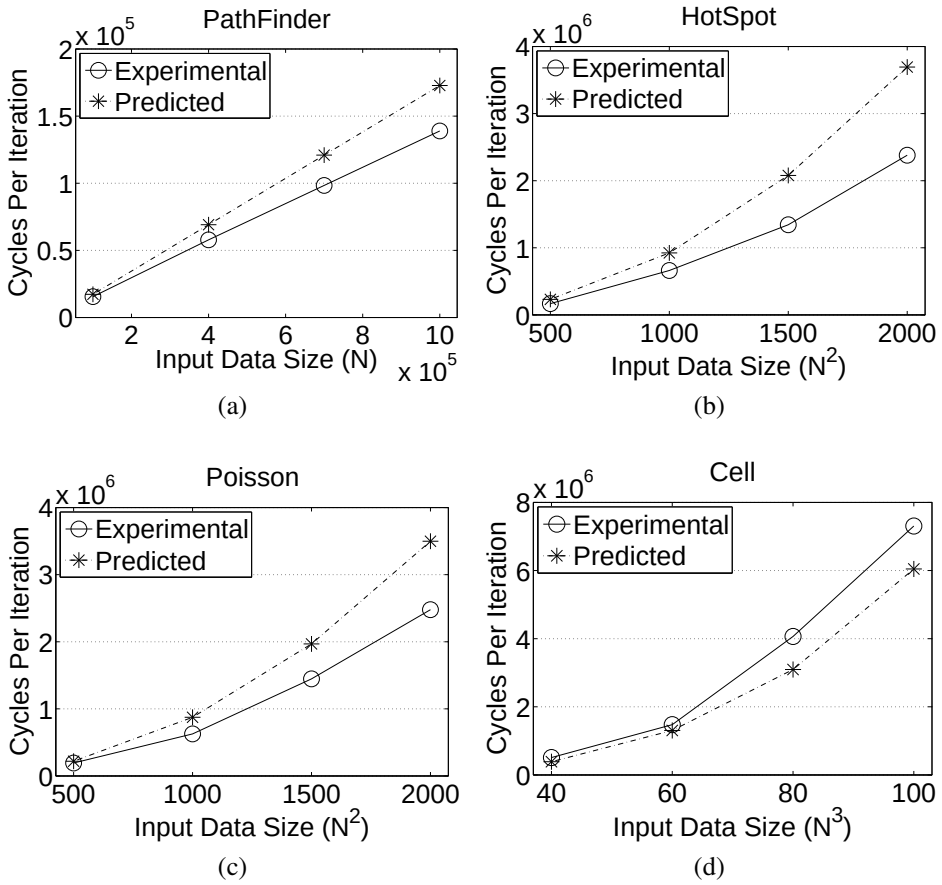


Figure 5.6: Model verification by scaling the input size of (a) PathFinder, (b) HotSpot, (c) Poisson, and (d) Cell. The trapezoid height is set to 16 for PathFinder, two for HotSpot and Poisson, and one for Cell. The thread block size is set to 256 for PathFinder, 20×20 for HotSpot, 16×16 for Poisson, and $8 \times 8 \times 8$ for Cell. The performance model captures the overall scaling trend for all benchmarks. The examples shown here perform global synchronization by memory fences; trends are similar with restarting kernels.

5.5.3 Sources of Inaccuracy

Our performance modeling may be subject to three sources of inaccuracy:

- *Unpredictable dynamic events.* This includes control-flow divergences, bank conflicts in both the PBSM and the device memory, and the degree of device memory

contention. Although the error introduced by control flow divergence is reduced by profiling the number of dynamic warp instructions which account for instructions in different branches, the other two types of errors are intrinsic in shared memory systems and they are difficult to prevent. The degree of bank conflicts in the device memory is estimated using arrays' dimensionality because higher dimensionality leads to strided accesses that are more likely to incur bank conflicts. Experiments also reveal that the frequency of bank conflicts also correlates to the thread block size in a nonlinear way. Moreover, due to the homogeneity of thread blocks, we assume thread blocks are likely to issue device memory requests close in time and therefore requests are queued and processed in bulks, maximizing the memory bandwidth. Finally, the effectiveness of latency hiding is also empirically approximated using an artificial factor.

- *Insufficient technical details.* Due to the lack of information regarding the launching of kernel functions, we are not able to accurately estimate the latency for global synchronization. Instead, we measure the latency of global synchronization based on a simplified, empirical model.

- *Approximation Error.* To maintain a continuous function for gradient based optimization, we have to calculate $ActiveWarpsPerBlock$ as floating point values rather than integers. This error is more significant for thread blocks narrow in length which lead to more branch divergence on the boundary, such is the case with Cell. In this scenario, a warp with most threads suspended may be counted as a small fraction, while in reality is should be counted as one.

Despite these sources of inaccuracy, we show that the performance modeling reflects the experimental performance scaling and it is sufficient to guide the selection of the optimal ghost zones or trapezoid configuration for ISL applications in CUDA.

5.5.4 Performance Optimization

Choosing Tile Size

We formulate the *cost-efficiency* of a trapezoid as the ratio between the size of the tile that it produces at the end and the size of the tile that it starts with. It is represented as

$$CostEffect = \frac{\prod_{i=0}^{D}(blk\_len - HaloWidth_i \times h)}{\prod_{i=0}^{D}(blk\_len - HaloWidth_i)} \tag{5.19}$$

For a given trapezoid height, a larger tile size always increases the cost-efficiency of trapezoid, reducing the percentage of stencil operations to replicate and achieving better performance, as we will demonstrate in Section 5.6.4. In our CUDA implementation, the tile size is determined by the thread block size which is set as large as possible within the architectural limit of 512.

Selecting Ghost Zone Size

The optimal ghost zone size is determined by the optimal trapezoid height, which in turn depends on the dynamics among computation and memory access. Our technique estimates the optimal trapezoid height in two steps. First, we comment out part of the application code in order to obtain $InstsPerSM_{MC}$ and $InstsPerSM_{IC}$ separately using the CUDA Profiler [148]. The run-time profiling is performed once for each ISL algorithm, and it only requires the computation of one stencil loop with a minimum input size big enough to occupy all the SMs. The resulting instruction counts are then used to compute the optimal trapezoid height in Equation 5.1. While Equation 5.1 is not a posynomial function, it is convex and we can obtain a unique solution using gradient-based constrained nonlinear optimization. We apply a constraint that sets the upper bound of the trapezoid height so that a tile produced at the end of a thread block has a positive area:

$$\forall i \in [0, D), blk\_len - HaloWidth_i \times h > 0 \tag{5.20}$$

The run-time profiling and ghost zone optimization needs to be performed only once for each ISL. However, they need to be recalculated if the same application is ported to systems with different settings (e.g. number of SMs, warp size, etc).

The performance model then predicts the optimal trapezoid height according to different applications and thread block sizes. In the example of Figure 5.7, we configure PathFinder with a thread block size of 256, HotSpot and Poisson with a thread block size of 20×20, and Cell with a thread block size of 8. The input size involved is $1,000,000$ for PathFinder, 2000×2000 for HotSpot, 2000×2000 for Poisson, and $100 \times 100 \times 100$ for Cell.

The predictions are compared to experimental results shown in Figure 5.7. Using the optimal trapezoid height obtained from actual experiments, the speedups compared to performance without ghost zones are 2.32X, 1.35X, 1.40X, 1.0X for PathFinder, HotSpot, Poisson and Cell, respectively; if memory fences are used, the speedups become 4.50X, 1.32X, 1.29X, 1.0X, respectively. The predicted optimal trapezoid heights turn out to exactly match the experimental results except for Poisson and PathFinder. In the case of Poisson implemented with memory fences, the optimal trapezoid height is predicted to be 2 but it is actually 3. In the case of PathFinder, the optimal trapezoid height is 16, and it is predicted to be 19 with memory fences and 12 without memory fences. Nevertheless, the performance at the predicted trapezoid height is no worse than 95% of the optimal performance.

Our estimation of the optimal trapezoid height can also serve as the initial guess for an adaptive selection method, assuming kernels are restarted after each stage to reset the trapezoid height. Suppose run-time profiling calculates the average execution time for

every two stages and decides whether to increment or decrement the trapezoid height, and there is a total of 30 iterations. With a more accurate initial guess, our adaptive technique achieves speedups of 2.29X, 1.35X, 1.40X, and 0.92X for PathFinder, HotSpot, Poisson and Cell, while an initial guess of one, as proposed in [127], results in speedups of 1.94X, 1.32X, 1.36X and 0.92X, respectively. The slowdown in Cell is because the performance does not benefit from ghost zones, however, the adaptive method still probes a suboptimal trapezoid height of two.

5.5.5 The Choice to Use Memory Fences

As described in Section 5.4.4 and Section 5.4.5, global synchronization can be implemented by either restarting kernel functions or using memory fences. The use of memory fences has two effects. First, thread coordinates and boundary conditions can persist through stages within a single kernel function call, and their corresponding computation, $MiscComp$, becomes negligible compared to the overall computation in all stages. Secondly, the overhead in global synchronization increases linearly with the number of thread blocks, which may eventually incur a latency longer than that resulted from restarting kernels. According to our performance model, the additional GPU cycles resulting from using memory fences can be estimated as:

$$\begin{aligned} Overhead_{MemFence} &= GlbSync_{MemFence} - \\ &\quad GlbSync_{KnlRestart} - MiscComp \times T \div M \qquad (5.21) \\ &= \mu \times 210.3 \times T - 3350 \\ &\quad -T \times \frac{InstsPerSM_{MC}}{N} \times blk\_len^{D} \times CPI \qquad (5.22) \\ &= T \times \delta - 3350 \qquad (5.23) \end{aligned}$$

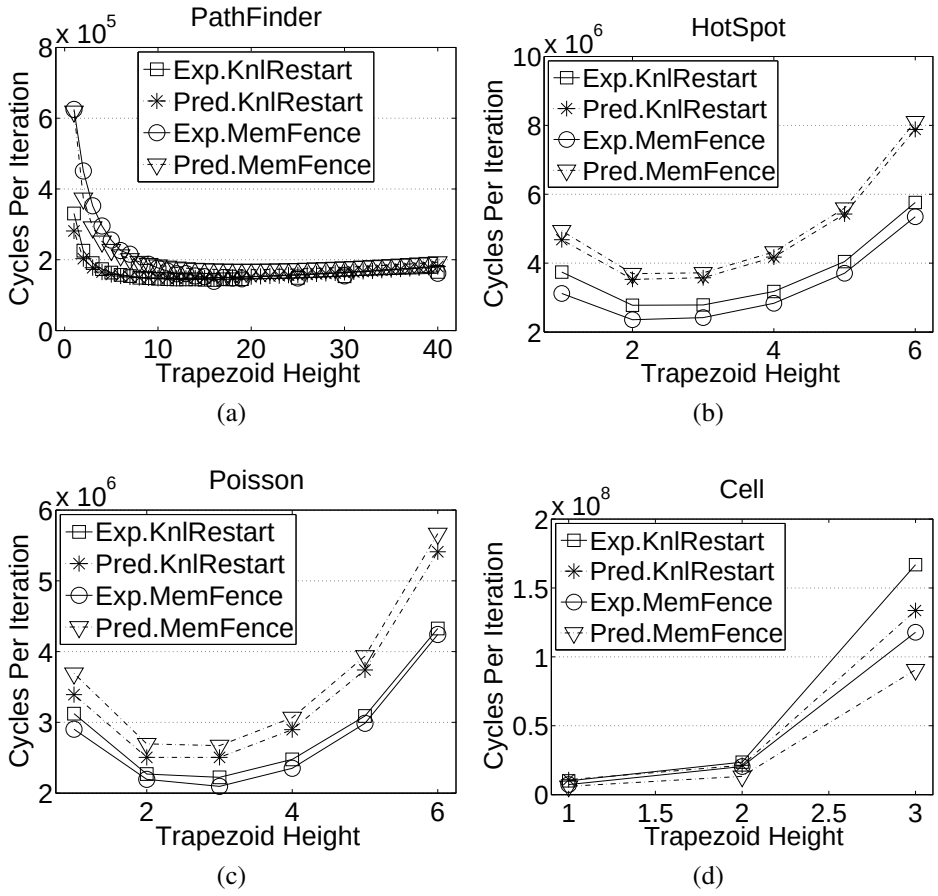


Figure 5.7: Evaluating the performance optimization by scaling the trapezoid height of (a) PathFinder, (b) HotSpot, (c) Poisson, and (d) Cell. "Exp" denotes actual performance measurement obtained from experiments, "Pred" denotes predicted performance obtained from the analytical model. Two types of implementations for global synchronization are evaluated; "KnlRestart" corresponds to restarting kernel functions, and "MemFence" corresponds to using the memory fences. The performance at the predicted trapezoid height is no worse than 95% of the optimal performance.

Given a particular architecture, a particular application, and a chosen block size, Equation 5.23 shows that the overhead of memory fences only correlates to the number of thread blocks, T. As T increases along with the input size or the trapezoid height, eventually the factor δ determines whether using memory fences is beneficial. A positive δ predicts that

using memory fences incurs overhead and thus restarting kernels is preferred. A negative δ predicts that using memory fences is beneficial. The absolute value of δ represents the confidence of the prediction. For the configurations used for Figure 5.7, the δ values are 87 for PathFinder, 21 for HotSpot, 25 for Poisson, and -464 for Cell. This prediction is reflected in our observation in Figure 5.7 where memory fences benefit the performance of Cell and hurts the performance of PathFinder. However, due to the approximate in modeling global synchronization, our model is not able to precisely predict whether using memory fences is beneficial to performance, especially when the confidence is relatively low. HotSpot and Poisson are examples. The model predicts that their performance will suffer from the memory fence, but they benefit instead. Nevertheless, such misprediction only occurs when the performance with memory fences is similar to that with restarting kernels.

5.6 Sensitivity Study

Using the validated modeling, we are able to further study the performance scaling brought by ghost zones on shared memory architectures. We show how it affects the execution time spent in computation, memory accesses, and global synchronization. We also investigate what applications and system platforms may benefit more from using ghost zones.

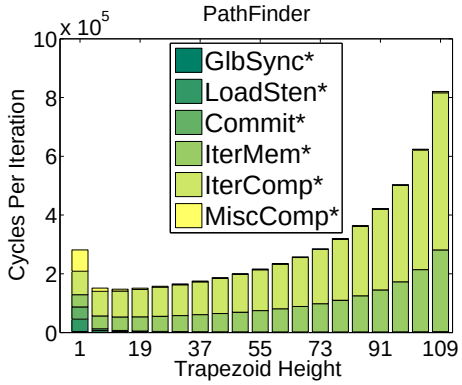
5.6.1 Component Analysis

We transform Equation 5.2 to the accumulation of six components, each term represents average cycles spent in one component within a stencil loop:

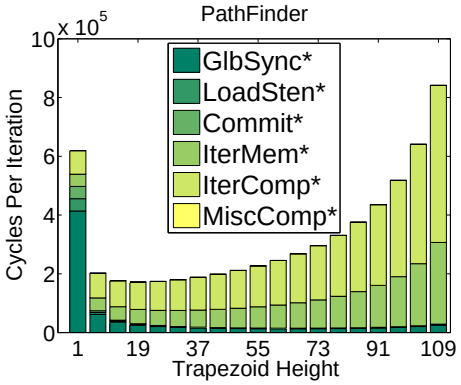
$$
\begin{aligned}
CPL &= GlbSync^* + LoadSten^* + Commit^* \\
&\quad + MiscComp^* + IterComp^* + IterMem^* && (5.24)\\
GlbSync^* &= \frac{GlbSync}{h} && (5.25)\\
LoadSten^* &= \frac{LoadSten}{h} && (5.26)\\
Commit^* &= \frac{Commit}{h} && (5.27)\\
MiscComp^* &= \frac{MiscComp \times T}{h \times M} && (5.28)\\
IterComp^* &= \frac{IterComp \times T}{h \times M} && (5.29)\\
IterMem^* &= \frac{IterMem}{h} && (5.30)
\end{aligned}
$$

We use PathFinder as an example to show the estimated performance breakdown obtained from our analytical model. As shown in Figure 5.8, all components contribute significantly to the overall performance except for $GlbSync^*$ in the case of restarting kernels and $MiscComp^*$ in the case of memory fences. As the trapezoid height increases, time spent in $LoadSten^*$ and $Commit^*$ drops while $IterMem^*$ and $IterComp^*$ increase gradually. Eventually, $IterMem^*$ and $IterComp^*$ dominate the execution time. Note that $IterMem^*$ only exists for those applications that access additional device memory objects during trapezoid computation.

Figure 5.9 illustrates in more detail about how each component reacts to the increased trapezoid height or ghost zone size. Each component's execution time is normalized to its time spent with a trapezoid height of one. The figure is drawn using a synthetic benchmark similar to HotSpot but with $NumElemPerOp$ set to one instead of zero to illustrate the scaling curve of $IterMem$.

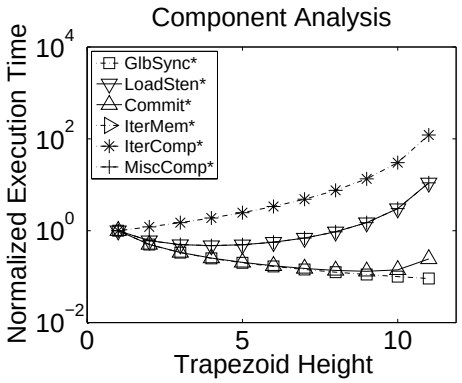


(a) Global synchronization by restarting kernels

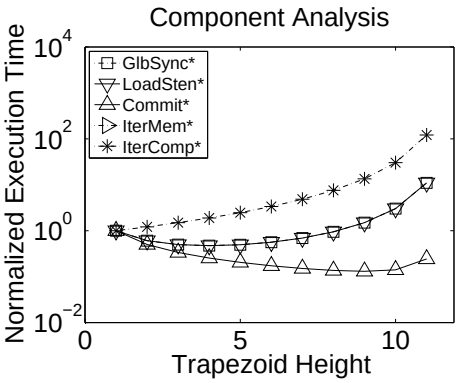


(b) Global synchronization with memory fences

Figure 5.8: The estimated performance breakdown for PathFinder obtained from our analytical model. Data is based on an input size of 1,000,000 and a block size of 256.



(a) Global synchronization by restarting kernels



(b) Global synchronization with memory fences

Figure 5.9: The effect of the trapezoid's height on each component with the execution time for each component normalized to the case where a trapezoid's height is one.

Time for $IterComp^*$ and $IterMem^*$ increases monotonically with taller trapezoids due to increased computation replication. However, time spent in $LoadSten^*$, $Commit^*$, $MiscComp^*$, and $GlbSync^*_{MemFence}$ decreases first with taller trapezoids due to reduced inter-loop communication and the number of stages. Nevertheless, they eventually increase

dramatically due to the exploding number of thread blocks resulted from taller trapezoids which end with tiny non-overlapping tiles.

Moreover, Equations 5.25 to 5.30 can be regarded as functions of h, and their derivatives are all monotonically increasing, as can be seen from Figure 5.9. As a result, the overall objective function is convex and it has a unique minimum.

5.6.2 Insensitive Factors

Although the uncontended latency ($UncontendedLat$) and the number of concurrent thread blocks per SM ($BlksPerSM$) is directly used in our performance model to calculate Equation 5.7, they are not sensitive to the overall performance. Under the architecture of GTX 280 and assuming a full block size ($blk\_len^D = 512$), Equation 5.7 can be re-written as

$$\begin{aligned} MemCycles(n) &= n \times \frac{UncontendedLat}{blk\_len^D \times M \times BlksPerSM} + \\ &\quad n \times \frac{\alpha \times x \times ClockRate}{MemBandwidth} \qquad (5.31) \\ &= 0.0024 \times n + \alpha \times 0.0367 \times n \qquad (5.32) \end{aligned}$$

with $\alpha \geq 1$. As a result, the first term involving $UncontendedLat$ and $BlksPerSM$ is at least an order of magnitude smaller than the second term. Therefore, the values of $UncontendedLat$ and $BlksPerSM$ hardly impact the overall performance. In fact, the two additive terms in Equation 5.32 each represents the memory access time attributed to latency and bandwidth, respectively. It reflects the fact that the GPU performance is usually bandwidth-bounded rather than latency-bounded.

The choice of the optimal trapezoid height also has little to do with the input size of the application. This is because in Equation 5.24, all the terms are linearly related to the input

size, n, except for $GlbSync^*_{KnlRestart}$ when global synchronization is implemented with restarting kernels. However, $GlbSync^*_{KnlRestart}$ becomes negligible compared to other terms when the input size is sufficiently large, as illustrated in Figure 5.8a. Overall, the optimal trapezoid height remains the same for different input sizes, as illustrated in Figure 5.10.

5.6.3 Application Sensitivity

According to the performance model, we summarize several characteristics that enable an application to benefit from ghost zones.

Stencils operating on lower-dimensional neighborhood. In fact, with the same trapezoid height and the same halo width in each dimension, the ratio of replicated operations grows exponentially with more dimensionality, as can be seen from Equation 5.8, 5.16, and 5.10. This phenomenon is observed in Section 5.5 where the 1-D PathFinder benefits the most, followed by 2-D HotSpot and Poisson, and the 3-D Cell does not benefit at all.

Narrower halo widths. A wider halo region increases the amount of operations to replicate, therefore it increases $IterComp^*$ which usually dominates the overall performance; the peak speedup becomes smaller and it tends to be reached with a shorter trapezoid (Figure 5.11).

Smaller computation/communication ratio. The penalty for replicating computation-intensive operations may be large enough to obscure ghost zone's savings in communication and synchronization. In this case, peak speedup is likely to be achieved with shorter trapezoids.

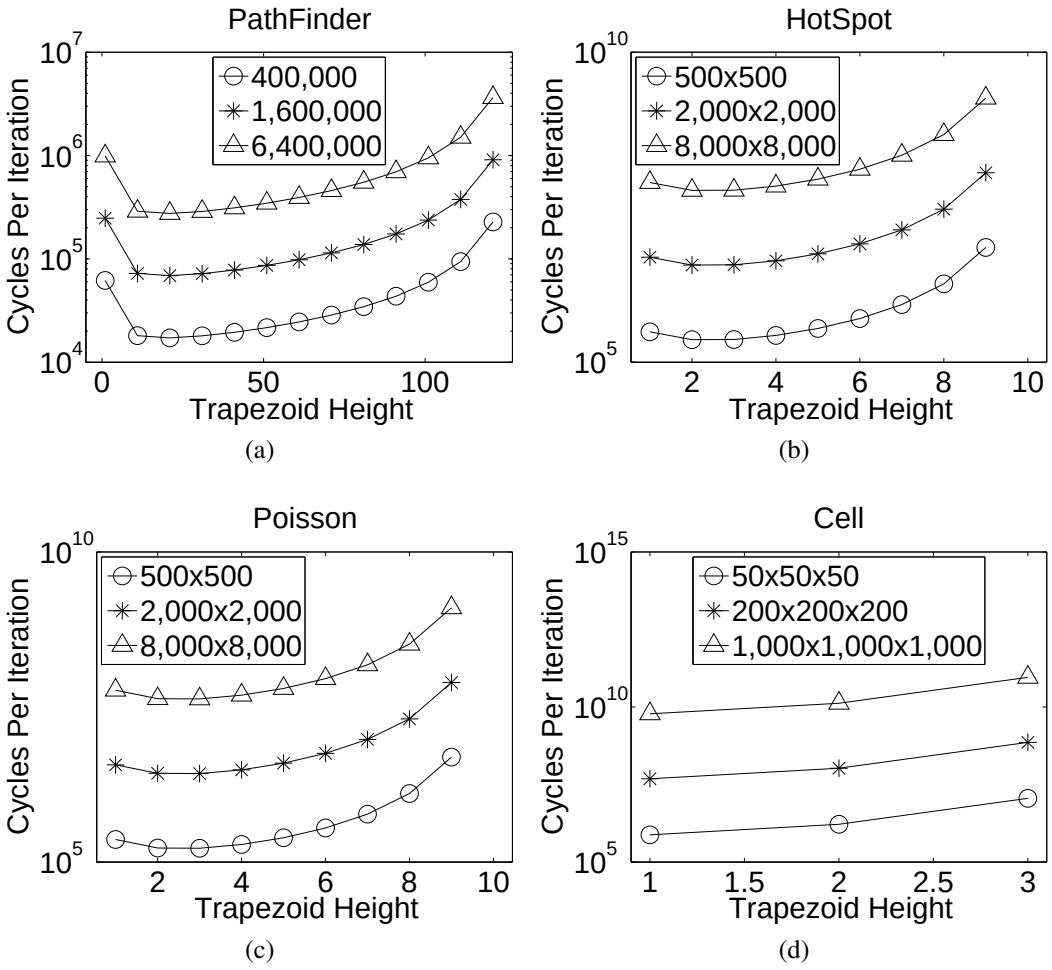


Figure 5.10: Compare the effect of various input sizes for (a) PathFinder, (b) HotSpot, (c) Poisson, and (d) Cell. The optimal trapezoid height is independent of the input size. Data is estimated using thread blocks sized 256 for PathFinder, 20×20 for HotSpot and Poisson, and $8 \times 8 \times 8$ for Cell. Global synchronization is implemented with memory fences but the overall trend is almost identical to that implemented by restarting kernels.

5.6.4 System Sensitivity

Although our performance model is based on the Tesla architecture, it can be easily extended to model other shared memory parallel systems. We investigate how adding ghost zones can benefit across different system parameters.

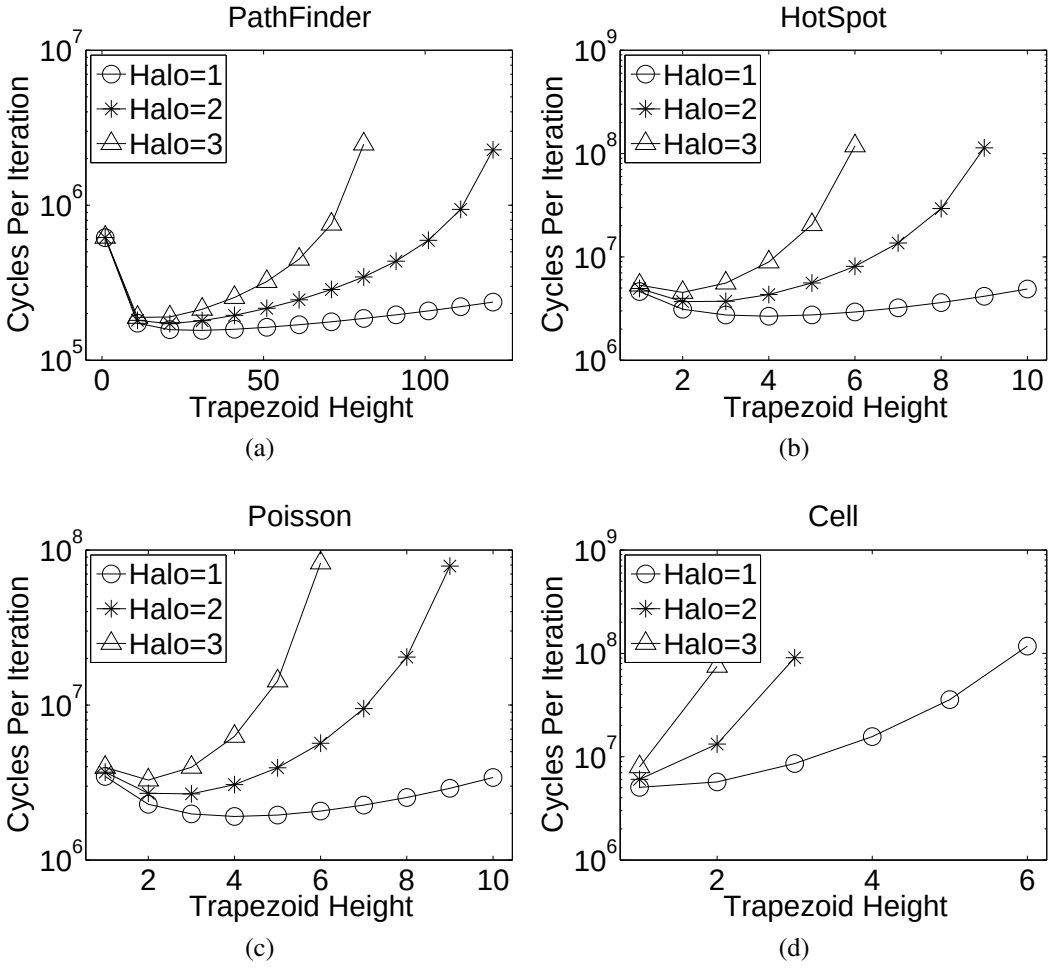


Figure 5.11: Comparing the effect of various halo widths for (a) PathFinder, (b) HotSpot, (c) Poisson, and (d) Cell. Global synchronization is implemented with memory fences but the overall trend is almost identical to that implemented by restarting kernels. Smaller stencils with smaller halo width can benefit more from the trapezoid technique, as demonstrated by our performance modeling by synthesizing four benchmarks with various halo width.

Larger tile size. As we discussed in Section 5.5.4, by using larger tiles, less computation needs to be replicated and performance can be improved. As Figure 5.12 shows, a larger thread block size increases the peak speedup and shifts the best configuration towards taller trapezoids. The benefit of ghost zones may be more significant for architectures that easily

allow for larger tile sizes, such as CBE. Note that the benchmark of Cell may benefit from the ghost zone technique if the architecture allows larger tile size or thread block size.

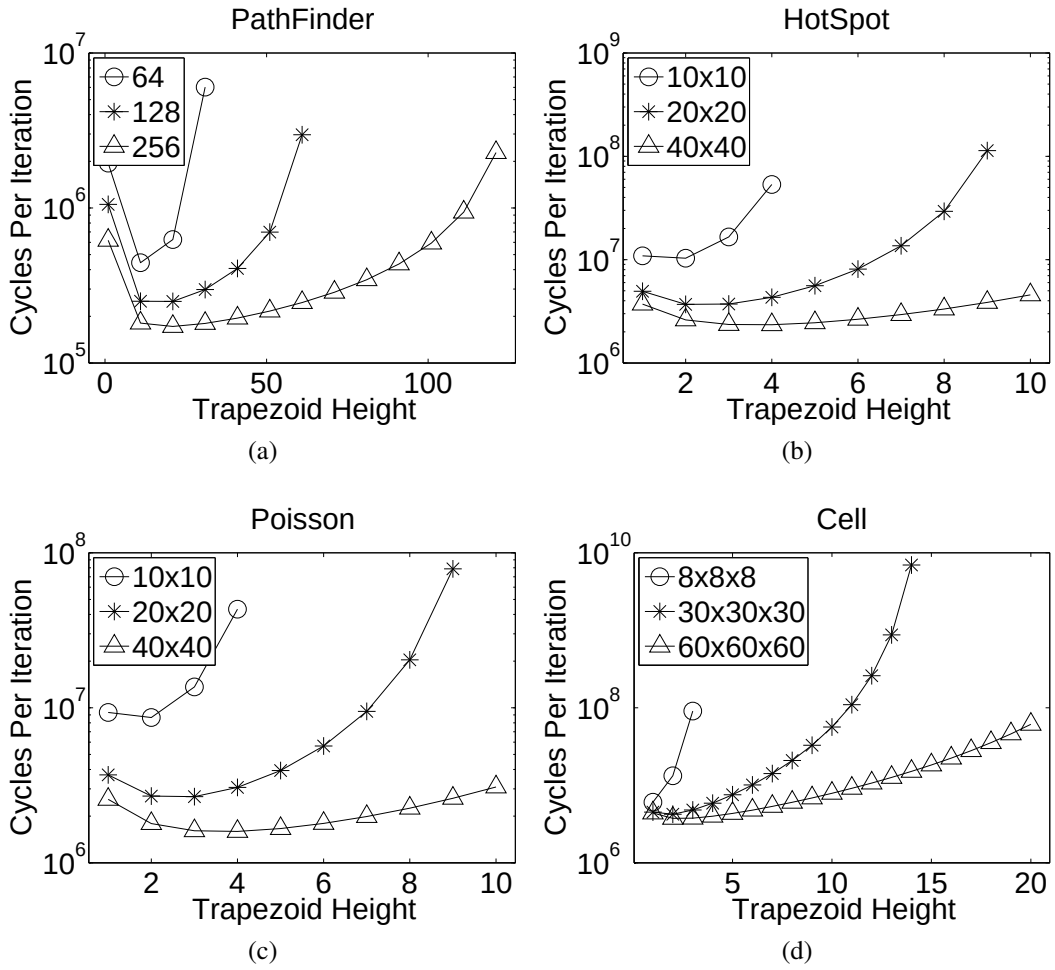


Figure 5.12: Compare the effect of various block sizes for (a) PathFinder, (b) HotSpot, (c) Poisson, and (d) Cell. Global synchronization is implemented with memory fences but the overall trend is almost identical to that implemented by restarting kernels. Code programmed with larger thread blocks can benefit more from ghost zones, as demonstrated using our performance modeling to scale the thread block size beyond the architectural limit.

Longer synchronization latency. One benefit of ghost zones is the elimination of inter-loop synchronization. As Figure 5.9 shows, $GlbSync^*$ always decreases with taller trapezoids. Nevertheless, latency in synchronization is not a major contributor to the overall performance on GTX 280.

Longer memory access latency. The savings in inter-loop communication is more evident with longer memory access latency. This can be caused by higher bandwidth demand, higher contention, or higher transferring overhead. For CBE whose memory bandwidth is not as optimized as Tesla, it is likely that ISL applications benefit more from ghost zones.

Smaller CPI. Technology is driving towards a smaller CPI — either by improving the instruction-level parallelism (ILP) or increasing the computation bandwidth (e.g. SIMDization). A smaller CPI puts less weight on computation and more on memory accesses, therefore programs can benefit from the ghost zones further.

5.6.5 Energy Consumption

Using ghost zones, the savings in energy consumption are approximately linear with the reduction in execution time. We measure the power consumption using the power meter WattsUp [151]. Because power is measured by plugging the machine's AC power into the WattsUp, the measurements are filtered by the behavior of the power supply. Therefore, the data collected from the WattsUp represents the total power that includes leakage and dynamic power spent by CPU, GPU, and buses. While the overall power increases around 84 watts on average after CUDA kernels are launched, we do not observe a noticeable change in power by using ghost zones. As a result, it appears that energy can be modeled as linear with the execution time, regardless of whether ghost zones are used. Since ghost zones do improve performance in most cases we studied (Section 5.5.4), they also improve energy efficiency.

Listing 5.2: Code annotation for automatic trapezoid optimization

```
float **A, **B;
for k = 0 : num_iterations with trapezoid.height=[H]
    for_all i = 0 : ROWS-1 and j = 0 : COLS-1
        /* define the array(s) to be loaded from */
        apply trapezoid.obj=[B]
        /* define the dimensionality of the array */
        apply trapezoid.dimension=2
        /* define the halo width in all dimensions
        apply trapezoid.gather[-1,+1][-1,+1]
        top = max(i-1, 0);
        bottom = min(i+1, ROWS-1);
        left = max(j-1, 0);
        right = min(j+1, COLS-1);
        A[i][j] = B[i][j] + B[top][j]          \
                + B[bottom][j] + B[i][left]    \
                + B[i][right];
    swap(A, B);
```

5.7 An Automated Framework Template for Trapezoid Optimization

Since the benefit of ghost zones depends on various factors related to both the application and the system platform, it is hard for programmers to find out the best configuration. Moreover, implementing ghost zones for ISL applications involves nontrivial programming efforts and it is often error-prone. We therefore propose a framework template that automatically transform ISL programs in CUDA to that equipped with the optimal trapezoid configuration. The framework is comprised of three parts: code annotation and transformation, one-time dynamic profiling and off-line optimization.

Listing 5.2 illustrate the proposed code annotation for the pseudocode in Listing 5.1. The programmer specifies the array to be loaded from and its dimensionality, as well as the halo width of the stencil operations. The framework is then able to transforms the code to that equipped with ghost zones. The code transformation also implicitly distinguishes computation involved in stencil loops from the rest for the purpose of profiling.

With the transformed code, the framework then counts the instructions and estimates the

computation intensity using the CUDA Profiler, as described in Section 5.4.3. Profiling is performed once implicitly and the results are used for calculation in the performance model. Finally, the performance model estimates the optimal trapezoid configuration based on the current system platform — as described in Section 5.5.4 — and generates appropriate code.

5.8 Conclusions and Future Work

We establish a performance model for ISL applications programmed in CUDA and study the benefits and limitations of ghost zones. The performance modeling based on the Tesla architecture is validated using four distinct ISL applications and it is able to estimate the optimal trapezoid configuration. The trapezoid height selected by our performance model is able to achieve a speedup no less than 95% of the optimal speedup for our benchmarks. Our performance model can be extended and generalized to other shared memory systems. Several application- and architecture- characteristics that can leverage the usage of ghost zones are identified. Finally, we propose a framework template that can automatically incorporate ghost zones to ISL applications in normal CUDA code and optimize it with the selection of trapezoid configurations. An immediate step in our future work will be to port our infrastructure to the OpenCL standard once suitable tools are available. Extensions can be implemented for CMPs with cache-based memory systems. It will also be interesting to study how the benefit from ghost zones is effected by cache prefetching and cache blocking optimizations.

Chapter 6

Dynamic Warp Subdivision for Latency Hiding Upon SIMD Divergence

6.1 Introduction

Single instruction, multiple data (SIMD) organizations use a single instruction sequencer to operate multiple datapaths or "lanes" in lockstep. SIMD is generally more efficient than *multiple instruction, multiple data* (MIMD) in exploiting data parallelism, because it allows greater throughput within a given area and power budget by amortizing the cost of the instruction sequencing over multiple datapaths. This observation is becoming important, both because data parallelism is common across a wide range of applications; and because data-parallel throughput is increasingly important for high performance as single-thread performance improvement slows.

SIMD lockstep operation of multiple datapaths can be implemented with vector units, where the SIMD operation is explicit in the instruction set and a single thread operates on

wide vector registers. SIMD lockstep can also be *implicit*, where each lane executes distinct threads that operate on scalar registers, but the threads progress in lockstep. The latter is also referred to by NVIDIA as *single instruction multiple threads* (SIMT). For purpose of generality in this chapter, we refer to the set of operations happening in lockstep as a *warp* and the application of an instruction sequence to a single lane as a *thread*. We refer to a set of hardware units under SIMD control as a *warp processing unit* or WPU.[1] SIMD organizations of both types are increasingly common in architectures for high throughput computing, exemplified today in STI's Cell Broadband Engine (CBE) [9], Clearspeed's CSX600 [10], Intel's Larrabee [15], Cray [11], and Tarantula [12]. Graphics processors (GPUs), including NVIDIA's Tesla [13] and Fermi [14] also employ SIMD organizations and are increasingly used for general-purpose computing. Academic researchers have also proposed stream architectures that employ SIMD organizations [152, 16], and in fact this history goes back to the Illiac project [153]. For both productivity and performance purposes, an increasing number of SIMD organizations support *gather and scatter*, where each lane can load from or store to unrelated addresses. (Vector architectures implement this by loading or storing a vector of data from or to a vector of addresses [96, 13, 15].) This introduces the possibility of "divergent" memory access latency, because a SIMD gather or scatter may access a set of data that is not fully in a particular level of the cache.

As in other throughput-oriented organizations that try to maximize thread concurrency and hence do not waste area on discovering instruction level parallelism, WPUs typically employ in-order pipelines that have limited ability to execute past L1 cache misses or other long latency events. To hide memory latencies, WPUs instead time-multiplex among multiple concurrent warps, each with their own PCs and registers. However, the multi-threading depth (i.e., number of warps) is limited because adding more warps multiplies the area

[1] We invent a new term here to distinguish it from "cores", "lanes" or "PEs," terms which are sometimes used to refer to individual scalar pipelines that constitutes the WPU.

overhead in register files, and may increase cache contention as well. As a result of this limited multi-threading depth, the WPU may run out of work. This can occur even when there are runnable threads that are stalled *only* due to SIMD lockstep restrictions. For example, some threads in a warp may be ready while others are still stalled on a cache miss.

This chapter observes that when a WPU does not have enough warps with all threads ready to execute, some individual threads may still be able to proceed. This occurs in two cases:

- Branch divergence: Branch divergence occurs when threads in the same warp take different paths upon a conditional branch. In current organizations, the WPU can only execute one path of the branch at a time for a given warp, with some threads masked off if they took the branch in the alternate direction. In array organizations, this is handled in hardware by a re-convergence stack [154] or conditional streams [155]; in vector organizations, this is handled in software by using the branch outcomes as a set of predicates [156, 157, 158]. In either case, allowing both paths to run creates problems in re-converging the warp.

- Memory latency divergence: Memory latency divergence occurs when threads from a single warp experience different memory-reference latencies caused by cache misses or accessing different DRAM banks. In current organizations, the entire warp must wait until the last thread has its reference satisfied. This occurs in both array and vector organizations (if the vector instruction set allows gather/scatter).

We propose *dynamic warp subdivision* (DWS) to utilize *both* thread categories. Warps are selectively subdivided into *warp-splits*. Each has fewer threads than the available SIMD width, but can be *individually regarded as an additional scheduling entity* to hide latency and can both be active:

- Upon branch divergence, a warp can be divided into two *active* warp-splits, each representing threads that fall into one of the branch paths. The WPU can then interleave the computation of different branch paths to hide memory latency.

- Upon memory latency divergence, a warp can be divided into two warp-splits as well: one represents

  threads whose memory operations have completed, the other represents threads that are still stalled (e.g., on a cache miss). The former warp-split need not suspend; it can run ahead non-speculatively and in the process touch and potentially prefetch cache lines that may also be needed by threads that fell behind.

In both cases, stall cycles are reduced, latency hiding is improved, and the ability to overlap more outgoing memory requests increases memory level parallelism (MLP). The challenge is to manage this process without reducing overall throughput: aggressive subdivision may result in performance degradation because it may lead to a large number of narrow warp-splits that only exploit a fraction of the SIMD computation resources. A dynamic mechanism is needed because the divergence pattern depends on run-time dynamics such as cache misses and may vary across applications, phases of execution, and even different inputs.

We evaluate several strategies for dynamic warp subdivision based upon eight distinct data-parallel benchmarks. Experiments are conducted by simulating WPUs operating over a two-level cache hierarchy that has private L1 caches sharing an inclusive, on-chip L2. The results show that our technique improves the average performance across a diverse set of parallel benchmarks by 1.7X. It is robust and shows no performance degradation on the benchmarks that were tested. We estimate that dynamic warp subdivision adds less than 1% area overhead to a WPU. This work was published in [68].

6.2 Impact of Memory Latency Divergence

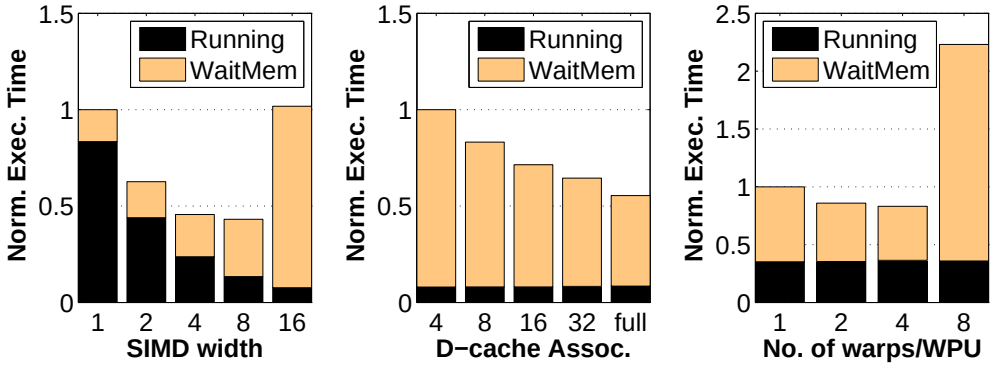


Figure 6.1: (a) A wider SIMD organization does not always improve performance due to increased time spent waiting for memory. The number of warps is fixed at four. (b) 16-wide WPUs with 4 warps even suffer from highly associative D-caches. (c) A few 8-wide warps can beneficially hide latency, but too many warps eventually exacerbates cache contention and increases the time spent waiting for memory. Results are harmonic means across the benchmarks listed in Table 7.2 on the system configuration in Table 7.1.

As SIMD width increases, the likelihood that at least one thread will stall the entire warp increases. However, *inter*-warp latency tolerance (*i.e.*, deeper multi-threading via more warps) is not sufficient to hide these latencies. The number of threads whose state fits in an L1 cache is limited. That is why *intra*-warp latency tolerance is needed. Intra-warp latency also provides opportunities for memory-level parallelism (MLP) that conventional SIMD organizations do not.

In order to illustrate the impact of memory latency divergence, the limitations on warp count, and the need for intra-warp latency tolerance, Figure 6.1a shows the performance scaling when varying the SIMD width from 1 to 16 with four warps sharing a 32 KB L1 D-cache. As the SIMD width grows, the overall execution time first improves (thanks to

greater throughput) but eventually starts to get worse even though this experiment is increasing the total computation resources; although the time spent in SIMD computation keeps decreasing, the time spent waiting for memory drastically increases and eventually dominates. The reasons for this trend are two-fold: first, a wider SIMD organization incorporates more threads, which increase D-cache contention; second, wider warps are more likely to incur memory divergence and suspend due to individual cache misses. The overall effect is fewer active warps but more latency to hide. Figure 6.1c shows that adding more warps does exacerbate L1 contention. This is a capacity limitation, not just an associativity problem, as shown in Figure 6.1b, where time waiting on memory is still significant even with full associativity.[2] These results are all averages across the benchmarks described in Section 7.4.2 and obtained with the configuration described in Section 7.4.

	FFT	Filter	HotSpot	LU	Merge	Short	KMeans	SVM
Avg. instruction count between branches	59	12	16	53	9	19	10	12
Percentage of divergent branches	3.4%	0%	1.4%	4.3%	13.1%	22.0%	2.0%	4.3%
Avg. instruction count between misses	7	27	7	5	45	6	47	11
Avg. instruction count between div. misses	10	30	10	6	75	8	57	17
Percentage of divergent memory accesses	69%	88%	77%	81%	60%	79%	83%	62%

Table 6.1: Characterization of the frequency of branch divergence and SIMD cache misses.

Intra-warp latency tolerance hides latencies without requiring extra threads. However, intra-warp latency tolerance is only beneficial when threads within the same warp exhibit divergent behavior. Table 6.1 shows that many benchmarks exhibit frequent memory divergence. A further advantage of intra-warp latency tolerance is that the same mechanisms

[2]It is true that current GPU architectures such as Tesla [13] and Fermi [14] have much higher warp counts because they reference global memory frequently (no L2 for Tesla and L2 is only a victim cache in Fermi), and have very long WPU latencies to issue back to back instructions from the same thread. If we increase L1 miss latency to 300 cycles (similar to having no L2), our architecture's optimal warp count also jumps to 8 or 16; the extra cache contention is now justified by the dramatic need for extra latency hiding. Intra-warp latency hiding is still beneficial in this case, of course.

also improve throughput in the presence of
branch divergence.

In summary, scaling SIMD width is not always beneficial in systems with limited on-chip storage capacity. In order to host wider SIMD organizations and further improve throughput, it is necessary to hide such latency without additional threads. Section 6.4 and Section 6.5 show how DWS can address both branch and memory divergence. DWS is the first technique that can address both sources of divergence using the same hardware mechanism.

6.3 Methodology

6.3.1 Overall Architecture

Branch and memory-latency divergence can affect a variety of architectures. In order to draw more general conclusions, this chapter models a general system that blends important aspects of modern CPU and GPU architectures: general-purpose ISAs (represented here by the Alpha ISA), SIMD organization, multi-threading, and a cache-coherent memory hierarchy. We model implicit, array-style SIMD instead of a vector architecture because this places fewer burdens on software. While there are simulators available for vector architectures, traditional CPUs, and GPUs, we are not aware of any that combine these aspects. We therefore developed MV5 [64] to simulate WPUs. MV5 is a cycle-accurate, event-driven simulator based on M5 [84]. Because existing thread schedulers do not directly support the management of SIMD threads, applications are simulated in system emulation mode with a set of primitives to create and schedule threads on SIMD resources.

The simulated applications are programmed in an
OpenMP-style API implemented in MV5, where data parallelism is expressed in `parallel`

`for` loops. Applications are cross-compiled to the Alpha ISA using G++ 4.1.0, with new instructions recognized by MV5 inserted to signal SIMD thread management to the simulator. The code for a thread is compiled as a scalar code, and the programming model then launches multiple copies for SIMD execution. Neighboring tasks are assigned to threads in the same warp in a way that exploits both spatial and temporal data locality [65].

	Benchmark Description
FFT	Fast Fourier Transform (Splash2 [97]). Spectral methods. Butterfly computation Input: a 1-D array of 65,536 (2^{16}) numbers
Filter	Edge Detection of an Input Image. Convolution. Gathering a 3-by-3 neighborhood Input: a gray scale image of size 500×500
HotSpot	Thermal Simulation (Rodinia [20]). Iterative partial differential equation solver Input: a 300×300 2-D grid, 100 iterations
LU	LU Decomposition (Splash2 [97]). Dense linear algebra. Alternating row-major and column-major computation Input: a 300×300 matrix
Merge	Merge Sort. Element aggregation and reordering Input: a 1-D array of 300,000 integers
Short	Winning Path Search for Chess. Dynamic programming. Neighborhood calculation based on the the previous row Input: 6 steps each with 150,000 choices
KMeans	Unsupervised Classification (MineBench [98]). Distance aggregation using Map-Reduce. Input: 10,000 points in a 20-D space
SVM	Supervised Learning (MineBench [98]). Support vector machine's kernel computation. Input: 100,000 vectors with a 20-D space

Table 6.2: Simulated benchmarks with descriptions and input sizes.

6.3.2 Benchmarks

We simulate a set of benchmarks shown in Table 7.2. The benchmarks are selected from several benchmark suites including MineBench [98], Splash2 [97], and Rodinia [20]. No consideration of SIMD divergence was present in selecting the benchmarks. The benchmarks are the same as those used to study thread scheduling when many threads share a

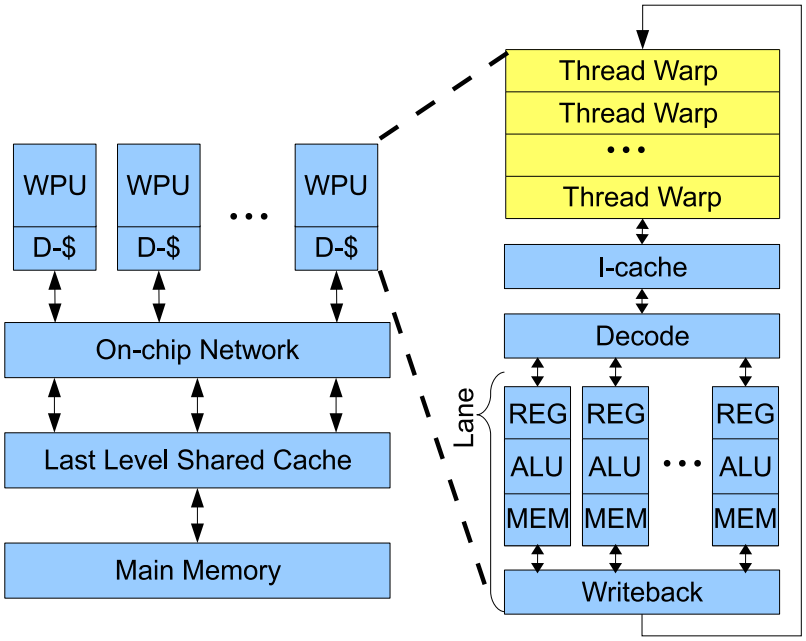


Figure 6.2: The baseline SIMD architecture groups scalar threads into warps and executes them using the same instruction sequencer. A thread operates over a scalar pipeline or lane that consists of register files, ALUs, and memory units.

cache [65]. These benchmarks are common, data-parallel kernels and applications. They cover the application domains of scientific computing, image processing, physics simulation, machine learning and data mining. They demonstrate varied data access and communication patterns. We increase the input sizes from the original benchmarks so that we have reasonable simulation times of within six hours. All means reported in this chapter are harmonic means.

6.3.3 Architecture Details

The baseline SIMD architecture used in this study is illustrated in Figure 7.3. The system has a two level coherent cache hierarchy. Each WPU has private I- and D-caches which

interact with an on-chip, shared, last-level cache (LLC). Only the LLC communicates with the off-chip memory. Examples of SIMD organizations that use cache hierarchies include Larrabee [15] and Fermi [14], both of which support general purpose computation. (Fermi's caches, however, are not cache-coherent). WPUs are simulated with up to 64 thread contexts and 16 lanes. The experiments in this chapter are limited to four WPUs because this provide reasonable simulation time.

A WPU groups scalar threads into warps. A fetched instruction is executed by threads within the same (active portion of a) warp simultaneously. Each thread's register state resides in a particular lane and the thread must execute in the corresponding lane. The register file is highly banked, with banks corresponding to lanes, so that multiple threads can access their operands at the same time without requiring deep multi-porting. Branch divergence is enabled by a re-convergence stack [154] — using a bit mask, threads that do not fall into the current control path are not executed. The mechanism is described in more detail in Section 6.4.1. Due to the lack of compiler support, we manually instrument the application code with post-dominators to signal control flow re-convergence after branches.

For the WPU lanes, we model in-order issue of one instruction per cycle. All instructions have latency of one except for memory references, which are modeled faithfully through the memory hierarchy (although we do not model memory controller reordering effects). The 4-way 16 KB L1 I-cache has a 1-cycle latency, while the baseline 8-way, 32 KB, L1 D-cache has a 3-cycle latency. Both caches use 128-byte lines. To hide latency on cache misses, a WPU switches to execute a different warp when the current warp accesses the cache. Switching warps takes no extra latency; the next warp is scheduled while the current warp issues.

This study does not model non-blocking loads whose benefits are limited with in-order issue. The hardware overhead is also considerable as the number of threads (SIMD width

$\times$ multi-threading depth) goes up. DWS would further increase the state for tracking non-blocking loads by the number of warp-splits allowed. These considerations are interesting areas for future work.

Each WPU has a private I-cache and D-cache. I-caches are not banked because only one instruction is fetched every cycle and distributed across all lanes. D-caches are always banked according to the number of lanes to cater to the high bandwidth demand of memory accesses. We assume there is a crossbar connecting the lanes with the D-cache banks. If bank conflicts occur, these memory requests are serialized and a small queuing overhead (one cycle) is charged. The queuing overhead can be smaller than the hit latency because we assume requests can be pipelined. Each cache bank performs its own tag match to support gather and scatter. All caches are physically indexed and physically tagged and use an LRU replacement policy. Memory coalescing is performed at the L1. All requests from a warp to the same cache line are coalesced in the MSHR. Otherwise, requests to the L2 are serialized. Each MSHR hosts a cache line and can track as many requests to that line as the SIMD width requires. Because the WPU supports gather/scatter, the TLB is multi-ported according to the number of lanes. The L1 caches connect to L2 banks through a crossbar. Coherence uses a directory-based MESI protocol.

Table 7.1 summarizes the main architectural parameters. Note that the aggregate L2 access latency is broken down into L1 lookup latency, crossbar latency, and the L2 tag and data lookup. The L2 then connects to the main memory through a 266 MHz memory bus with a bandwidth of 16 GB/s. The latency in accessing the main memory is assumed to be 100 cycles, and the memory controller is able to pipeline the requests.

Tech. Node	65 nm
WPU	1 GHz, 0.9 V Vdd, Alpha ISA, in-order 64 hardware thread contexts 4 warps with a SIMD width of 16 SIMD threads translate addresses simultaneously
I-Cache	16 KB, 4-way associative, 128 B line size 1 cycle hit latency, 4 MSHRs, LRU, write-back physically tagged, physically indexed
D-Cache	32 KB, 8-way associative, 128 B line size MESI directory-based coherence 3 cycle hit latency, LRU, write-back 32 MSHRs each hosts up to 8 requests physically tagged, physically indexed
L2 Cache	4096 KB, 16-way associative, 128 B line size 30 cycle hit latency, LRU, write-back 256 MSHRs each hosts up to 8 requests
Crossbar	300 MHz, 57 Gbytes/s
Memory	100 cycles access latency

Table 6.3: Hardware parameters used in studying Dynamic Warp Subdivision.

6.4 Dynamic Warp Subdivision Upon Branch Divergence

To hide more latency in the case of insufficient warps, we identify two categories of threads that may be unnecessarily suspended and can actually continue to execute. In this section, we discuss threads suspended due to branch divergence. Section 6.5 discusses threads suspended due to memory divergence.

6.4.1 Branch Divergence, Memory Latency and MLP

Figure 6.3 illustrates the conventional mechanism presented by Fung *et al.* [154] to handle divergent branches. Assume that a warp with a SIMD width of eight encounters a conditional branch at the end of code block A. Six of the threads branch to code block B (indicated by bit mask 11111001 in Figure 6.3), while the other two threads branch to code block C (indicated by bit mask 00000110 in Figure 6.3). The WPU first chooses to execute code block B. It then pushes the warp's re-convergence stack with the above two sets of bit masks, with the one that corresponds to code block B on the top of the stack

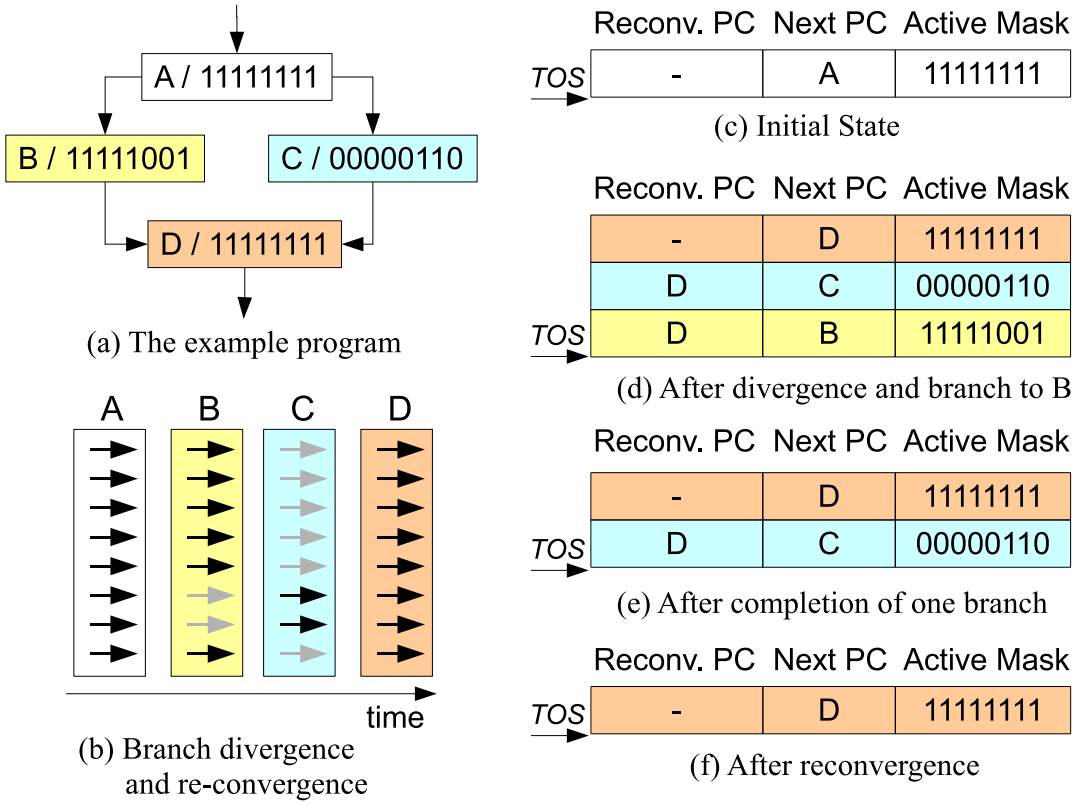


Figure 6.3: Conventional mechanism to handle branch divergence and re-convergence for a SIMD organization. (a) An example program; (b) the execution trace of diverged threads; (c)-(f) the state of the re-convergence stack. This is adapted from a figure by Fung *et al.* [154].

(TOS). The first PC in code block D becomes the *immediate post-dominator* of B. Not until this post-dominator is reached can the re-convergence stack pop and switch to activate threads executing the alternate path (code block C) — in this way, the re-convergence stack is able to handle potentially nested branches within code block B. Eventually code block C re-converges at the same post-dominator into code block D.

The fact that the re-convergence stack can only activate one branch path at a time may limit a WPU's ability to hide latency and leverage MLP. Figure 6.4 illustrates this scenario.

If threads executing code block B miss the cache and all other warps are waiting for memory as well, the WPU has to stall even though the threads that branched into code block C do not suffer from cache misses; these threads could actually continue to execute if it were not for the re-convergence mechanism.

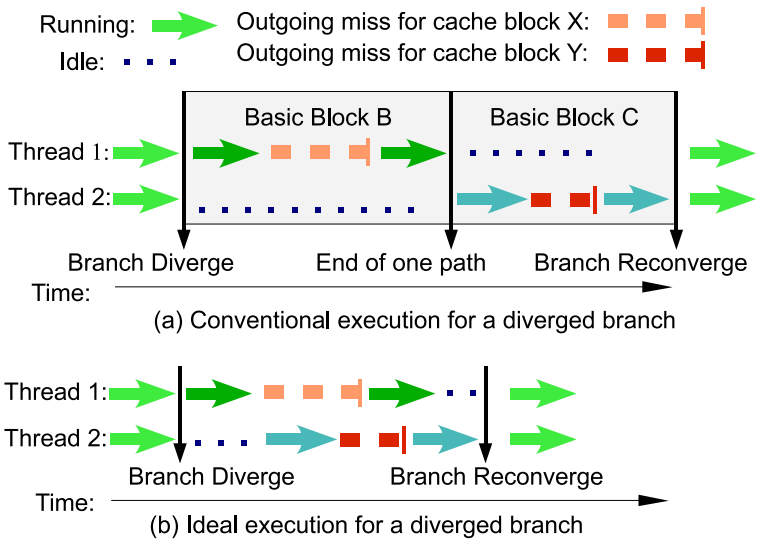


Figure 6.4: (a) The conventional mechanism serializes the execution of different branch paths. (b) By allowing threads that take different paths to interleave their execution, more latency can be hidden and memory level parallelism can be improved.

The post-dominator based re-convergence may also undermine MLP when threads that have reached their post-dominator have to wait for those that have not. As shown in Figure 6.5, if threads executing code block C miss the cache and all other warps are waiting for memory as well, the WPU has to stall even though the threads that finished code block B do not suffer from cache misses. If re-convergence can be relaxed, these threads can make progress themselves, getting their own memory requests issued earlier, as well as prefetching for the others.

It is important to note that DWS upon branch divergence can hide latency and improve

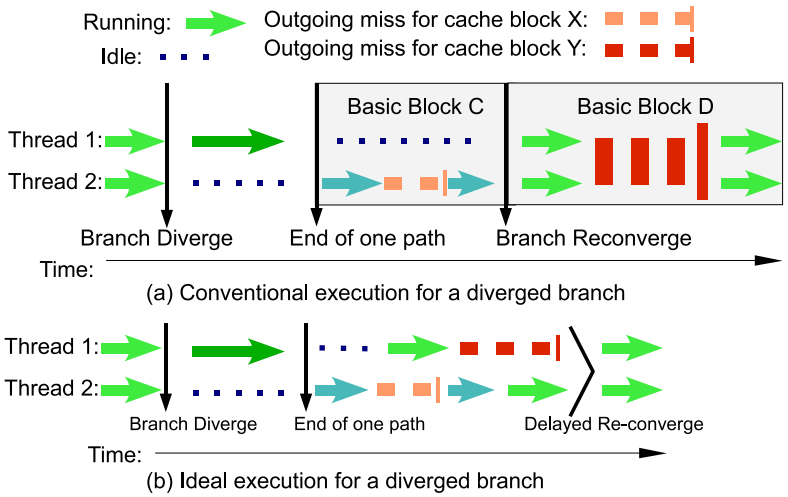


Figure 6.5: (a) The conventional mechanism forces diverged threads to re-converge at the post-dominator. (b) By allowing threads that reach the post-dominator first to run ahead, they can issue more outgoing requests to leverage memory level parallelism.

MLP *with or without the presence of memory divergence*. As we will discuss in Section 6.5, DWS upon branch divergence and DWS upon memory divergence are independent, complementary techniques, but DWS allows them to be integrated using the same hardware mechanisms and re-convergence policies. In order to isolate the role of branch divergence, results in this section disable the ability to perform DWS on memory divergence (i.e., warps cannot proceed until all threads' memory operations have completed).

6.4.2 Relaxing the Re-convergence Stack for Memory Throughput

We propose to subdivide a warp into two warp-splits upon branch divergence. Warp-splits are *independent scheduling entities* and are treated equally as warps by the scheduler. A warp-split can conceivably be recursively subdivided in future divergence events until it consists of only one thread. A full warp can be viewed as a *root warp-split* with full SIMD width. We use the term *SIMD groups* to refer to both full warps and warp-splits in the rest

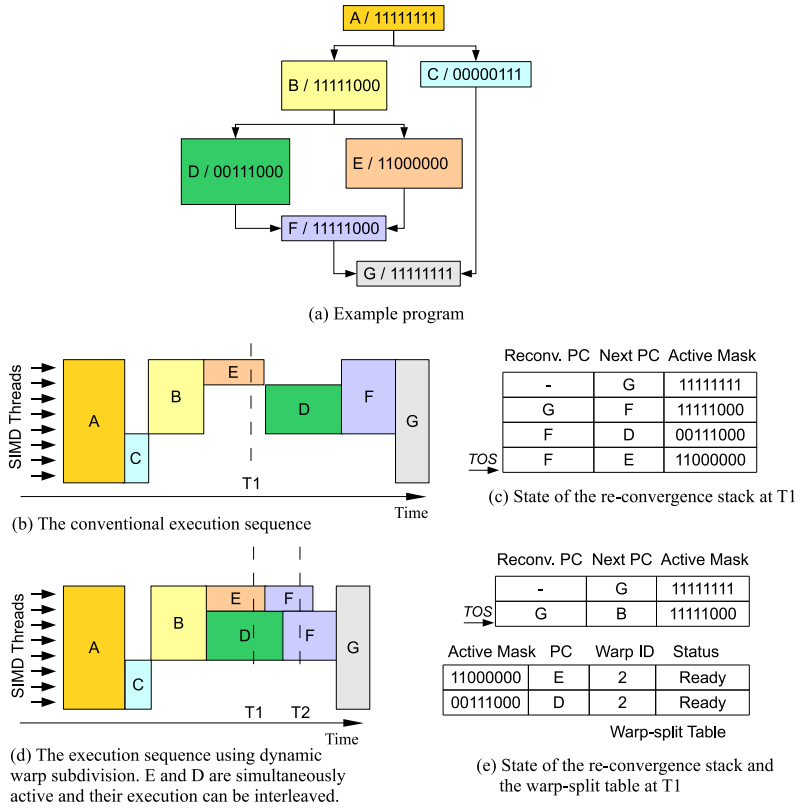


Figure 6.6: An example of dynamic warp subdivision upon branch divergence. Conventionally, different branch paths in the example program (a) are pushed to different layers in the re-convergence stack (c) and the execution of different paths is serialized (b). With dynamic warp subdivision, branch paths create entries in the warp-split table instead (e), denoting SIMD thread groups that can be simultaneously active. Their immediate post-dominator is ignored as well and no longer enforces re-convergence. As a result, diverged threads can interleave their execution (d), and they no longer have to re-converge at the beginning of code block F. Therefore, F is executed twice by different warp-splits; this may improve MLP but it also risks pipeline under-utilization.

of the chapter.

After a divergent branch subdivides a warp, the resulting warp-splits can interleave their computation to hide latency and improve memory level parallelism. This is demonstrated by an example in Figure 6.6. After a divergent branch in code block B, the conventional

approach uses the re-convergence stack to serialize different branch paths (Figure 6.6b). Alternatively, dynamic warp subdivision uses a warp-split table described in Section 6.4.4 so that a warp can progress like a binary tree with simultaneously active warp-splits (Figure 6.6d).

6.4.3 Over-subdivision

Warps may not benefit from subdivision upon *every* divergent branch — such aggressive subdivision may lead to a large number of narrow warp-splits, which can otherwise run altogether in a wider SIMD group. This phenomenon is referred to in the rest of the chapter as *over-subdivision*. Due to the potential for over-subdivision, we subdivide warps conditionally upon selected branches.

A static approach is used to select which branches are allowed to subdivide warps. As a side effect of subdivision, SIMD resources may be under-utilized if warp-splits are not re-converged in time. This occurs when two warp-splits have both passed their common post-dominator, but are not able to re-converge. This is illustrated in Figure 6.6 when the execution reaches time T2. We therefore use the heuristic that warps only subdivide upon branches whose associate post-dominator is followed up by a short basic block (F in the example of Figure 6.6) of no more than 50 instructions. This value was chosen because it takes roughly the same time to execute that many instructions as to handle an L1 miss, so run-ahead threads resulting from subdivision can hide some latency and initiate some further memory requests without running too far ahead and potentially inhibiting re-convergence. We manually instrumented the code to identify branches that subdivide warps, but in practice this process would be automated by the compiler.

6.4.4 Unrelenting Subdivision

When warp-splits independently execute the same instruction sequence while there is not much latency to hide, SIMD resources are under-utilized for little benefit. This scenario is referred to in the rest of the chapter as *unrelenting subdivision*. Under such circumstances, it may be better to re-converge the warp-splits as soon as possible so they can execute together as a wider SIMD group. Therefore it is important *both* to create warp-splits and to re-converge them appropriately.

To meet such requirement, a warp-split table (WST) is introduced *in addition to* the re-convergence stack to keep track of subdivision while preserving the re-convergence mechanism. Each entry in the WST corresponds to a warp-split. It records the warp-split's parent warp, the current PC, and the active mask that indicates the associated threads. Figure 6.6 illustrates an example of warp subdivision. Upon a divergent branch at the end of code block B, the warp can choose to push the re-convergence stack to serialize code blocks E and D (Figure 6.6b, c), or it can choose to be subdivided according to the branch outcome so that code blocks E and D can be interleaved (Figure 6.6d, e). If a warp is subdivided, two additional entries are created in the warp-split table. The re-convergence stack remains intact to avoid imposing a particular execution order of the paths; it can therefore no longer keep track of this and future branches nested within the branch registered on its TOS. As a result, future nested branches as well as their post-dominators are ignored by the re-convergence stack; warp-splits continue executing asynchronously and keep being subdivided upon future divergent branches until they reach the post-dominator associated with the top of the re-convergence stack. At this point, the warp-split stalls waiting to be re-united with other warp-splits once they reach the same post-dominator. The re-convergence is performed by the re-convergence stack in the conventional manner. We name this scheme *stack-based re-convergence*.

6.4.5 PC-based Re-convergence

Using stack-based re-convergence, warp-splits can eventually re-converge; however, they can re-converge earlier when they happen to execute the same instruction. For example, at time T2 in Figure 6.6d, both warp-splits are asynchronously executing the same code block and they *might* arrive at the same PC. In such cases, the two warp-splits can be re-united naturally without stalling any of them.

By recording the PC of each warp-split in the warp-split table, the WPU can identify and re-unite warp-splits belonging to the *same* warp that are *ready* to execute and share the same PC. We name this scheme *PC-based re-convergence*. Note that stack-based re-convergence is still used if PC-based re-convergence does not occur. A similar principle has been used in Fung et al.'s dynamic warp formation, although it was used to group *any* threads from *different* warps that execute the same branch path [154].

While comparing multiple PCs may take three to four cycles, such PC-based re-convergence does not have to be checked every cycle, and the latency can usually be hidden. Because the scheduler only switches SIMD groups upon cache accesses, resumed warp-splits from the ready queue always start with memory instructions. As a result, PCs need only be compared when the running warp-split executes a memory instruction. Furthermore, because re-convergence takes place only when there are one or more SIMD groups in the ready queue, the WPU can immediately switch to execute another SIMD group, preferably from a different warp, to hide the latency PC comparisons.

6.4.6 Results

Dynamic warp subdivision upon branch divergence is evaluated with both stack-based re-convergence and PC-based re-convergence. Figure 6.7 compares their speedups over a conventional architecture with no DWS. While stack-based re-convergence demonstrates

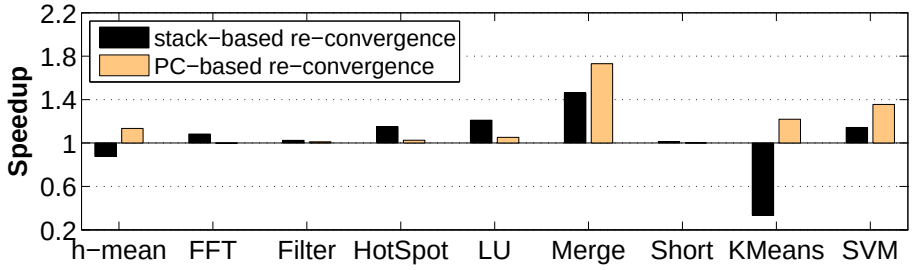


Figure 6.7: Performance gained by dynamic warp subdivision upon branch divergence. Speedups are normalized to that of the equivalent WPUs without DWS. Compared to stack-based re-convergence, PC-based re-convergence reduces unrelenting subdivision and improves performance significantly without ever making performance worse.

performance gains for some applications, it penalizes the performance of KMeans significantly due to over-subdivision and unrelenting subdivision — the average SIMD width for executed instructions are reduced to 4 for 16-wide WPUs. By introducing PC-based re-convergence, the average SIMD width for executed instructions increases to 9. This drastically reduces the under-utilization of SIMD resources. However, without memory divergence handling, warp-splits re-converged upon the same PC are not able to split again upon cache misses to hide intra-warp latency. As a result, for some applications, stack-based re-convergence performs slightly better than PC-based re-convergence. Overall, PC-based re-convergence outperforms stack-based re-convergence and it generates an average speedup of 1.13X.

Not all benchmarks are sensitive to branch divergence. Table 6.1 shows that the benchmarks benefiting from DWS frequently encounter conditional branches and the branch outcomes exhibit a significant fraction of divergence. Besides the occurrence of divergent branches, the benefit of DWS is also affected by other run-time dynamics. In the case of KMeans and Merge, memory divergence occurs frequently and there are usually ample instructions for latency hiding between cache misses. Therefore, they benefit significantly

from having more warp-splits to hide latency.

6.5 Dynamic Warp Subdivision Upon Memory Divergence

In conventional SIMD implementations, memory divergence stalls an entire warp. We propose to dynamically subdivide warps upon memory divergence so threads that hit can continue execution to hide latency for the threads that missed. This also offers the possibility for run-ahead threads to bring in cache lines that the fall-behind threads will also need. Warp-splits can be subdivided recursively upon future memory divergence.

6.5.1 Improve Performance upon Memory Divergence

Figures 6.8 and 6.9 compare conventional SIMD execution with dynamic warp subdivision upon memory divergence. Consider a warp with two memory instructions (not necessarily consecutive). Assuming all other warps have been suspended already, DWS can reduce pipeline stalls and leverage MLP in two scenarios:

- Threads that miss upon the former instruction hit the cache with the latter instruction, while threads that hit first later miss the cache. In this case, DWS allows run-ahead threads to initiate their misses earlier (Figure 6.8).

- Some threads hit and some miss on the first instruction, but all threads miss on the same cache block with the latter instruction. In this case, the run-ahead warp-splits in DWS not only initiate their misses earlier; they also prefetch data for the fall-behind warp-split (Figure 6.9). In contrast to speculative precomputation [63] or run-ahead simultaneous threads [62], the run-ahead warp-split always performs *useful* computation and threads' states are saved right away, requiring no ROB or dependency analysis that would otherwise complicate the design of the simple, in-order WPU.

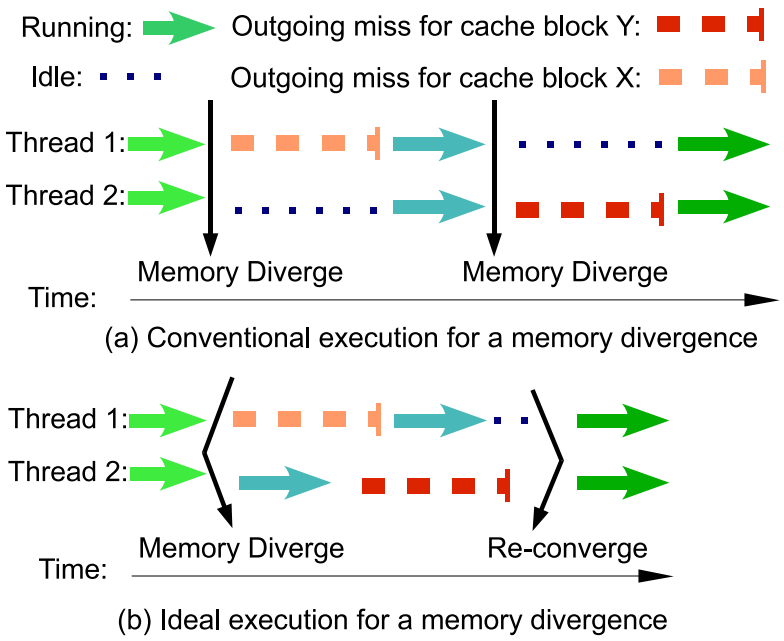


Figure 6.8: Dynamic warp subdivision upon memory divergence can reduce pipeline stalls and improve MLP. For illustration purposes, the SIMD width is shown as two but similar scenarios exist for wider SIMD organizations as well.

The above scenario is applicable to both array and vector organizations if they support gather loads or scatter stores. Using the same principle as DWS in array organizations, a WPU can use a set of bit masks to mark out vector components that hit the cache so that they can continue their execution and issue more memory requests. The PC upon which the memory divergence occurs can be stored in a table so that those vector components that miss can resume later at the recorded PC afterwards.

However, DWS may not improve performance if the same subset of threads keep missing the cache and requesting disparate addresses. If other SIMD groups are not able to hide sufficient memory latency, the computation *and* the memory latency incurred by those fall-behind threads become the critical path of the execution. As a result, the overall execution time would stay the same even if DWS allows other threads to run ahead.

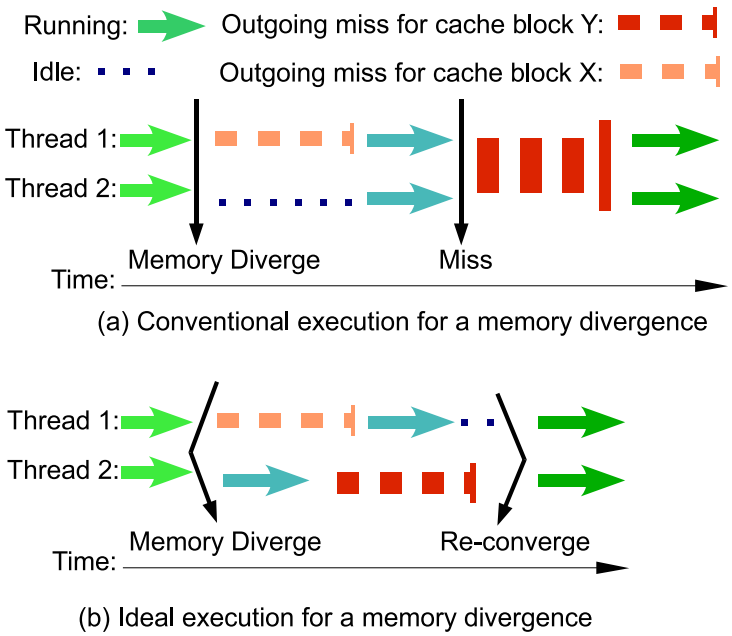


Figure 6.9: Dynamic warp subdivision upon memory divergence allows run-ahead threads to prefetch cache blocks for the fall-behind.

Similar to branch divergence, unconstrained warp subdivision may lead to over-subdivision. Therefore, we investigate several methods in Section 6.5.2 to selectively subdivide warps upon memory divergence. On the other hand, to reduce unrelenting subdivision, we exploit different mechanisms in Section 6.5.3 to re-converge warp-splits. Finally, it is important to handle branch divergence correctly even with the introduction of warp subdivision upon memory divergence. This is ensured by re-convergence techniques described in Section 6.5.3.

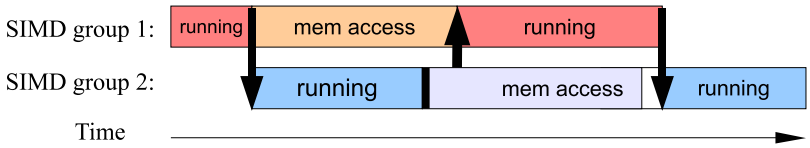
6.5.2 Preventing Over-subdivision

Warp subdivision is not likely to provide benefits when there are already sufficient active warps to hide latency. Aggressively subdividing warps upon every memory divergence regardless of the existence of other active SIMD groups may unnecessarily generate many narrow warp-splits which under-utilize SIMD resources. This brute force subdivision scheme is referred to in the rest of the chapter as *AggressSplit*.

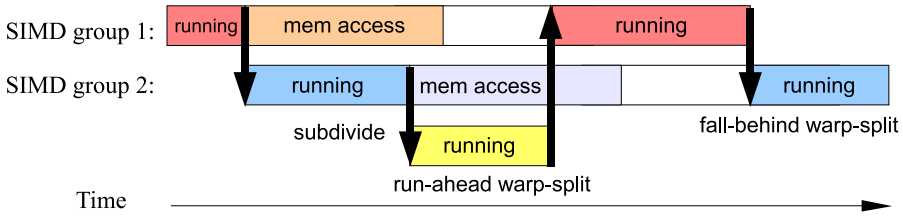
To lower the chance of unnecessary warp subdivision, the WPU can create more warp-splits only when there are no other SIMD groups to hide latency. Upon memory divergence, the WPU checks whether all other SIMD groups are waiting for memory. If so, the WPU subdivides the current active warp or warp-split to allow threads that hit to run ahead. This subdivision scheme is referred to as *LazySplit*.

However, LazySplit risks the inability to effectively subdivide warps. If the last active SIMD group has all its threads miss the cache, LazySplit has to stall the WPU's pipeline even though there may be other SIMD groups that have previously incurred memory divergence but were not subdivided and are waiting for memory. We therefore extend LazySplit by allowing it to look for those suspended SIMD groups that are eligible for subdivision when the pipeline is stalled. To avoid over-subdivision, only one SIMD group is subdivided at a time. We name this subdivision scheme *ReviveSplit*.

While ReviveSplit can leverage all SIMD groups that can be subdivided upon memory divergence, its performance may still be suboptimal. Since ReviveSplit attempts to subdivide warps whenever the pipeline is stalled, performance may degrade if the resulting run-ahead warp-split is not able to issue subsequent long latency memory requests in time (*i.e.*, before a suspended SIMD group is resumed by a completed request), as illustrated in Figure 6.10. In this case, the run-ahead warp-split may occupy the pipeline, keeping those resumed warp-splits from making progress while exploiting no more MLP. Afterwards, the



(a) Example case for conventional SIMT execution



(b) Subdivision upon misses may lead to performance degradation

Figure 6.10: Performance may degrade if the run-ahead warp-split does not generate long latency memory requests before any outgoing request completes. Such case may take place with LazySplit and ReviveSplit. We demonstrate the case with two warps with arrows pointing to the warp-split that the WPU executes.

same instruction stream will be executed again by the fall-behind warp-split, increasing the number of executed cycles. However, to make an optimal decision to subdivide warps, foreknowledge or speculation is required to estimate how soon a run-ahead warp-split will encounter another cache miss and how soon outgoing requests will complete and resume awaiting SIMD groups. Compiler analysis, dynamic optimization, or prediction hardware may be able to provide such information, and this is an interesting area for future work.

6.5.3 To Re-converge or To Run Ahead

If warp-splits are not re-converged, they will individually execute the same instruction sequence that could otherwise be run together in a wider warp. If such overhead outweighs the benefits of latency hiding and MLP brought by warp-slits, warp subdivision may penalize performance. On the other hand, if re-convergence is enforced too early, the run-ahead

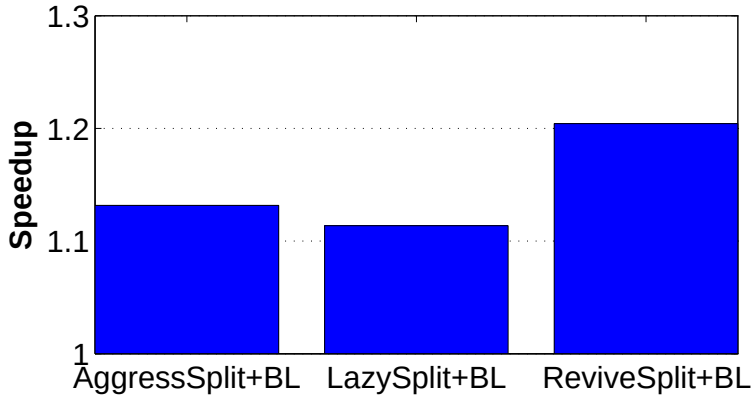


Figure 6.11: Harmonic means of speedups across all benchmarks. Dynamic warp subdivision upon memory divergence alone has limited benefits due to frequent branches. The BranchLimited re-convergence (specified by *BL*) results in little performance gains for all subdivision schemes, including AggressSplit, LazySplit, and ReviveSplit.

warp-split is likely to stall waiting for the fall-behind warp-split before it can beneficially issue another memory request to overlap outgoing requests. Nevertheless, PC-based re-convergence introduced in Section 6.4.5 allows multiple warp-splits to run together as a wider SIMD group without the cost of stalling any warp-split. Therefore it is always used in DWS upon memory divergence. However, since PC-based re-convergence does not force a run-ahead warp-split to wait for the fall-behind, it alone may still lead to unrelenting subdivision.

Similar to the process of subdivision, determining the optimal timing to enforce re-converge requires foreknowledge about whether future cache misses can occur in time to overlap with outgoing requests. In this chapter, we exploit several heuristics to effectively address re-convergence; some natural places to force warp-splits to re-converge are branches and post-dominators, as discussed below.

Branch Handling after Memory Divergence

Branches may interfere with the re-convergence process of warp-splits that divided upon memory divergence. Upon branches or post-dominators, the re-convergence stack acts implicitly as a barrier for the subdivided warp-splits to re-converge; when warp-splits are synchronized and re-
converged, the re-convergence stack can be pushed or popped in a conventional manner. Afterwards, the warp can be subdivided again upon future memory divergence. In other words, only those threads masked on the TOS are subdivided into warp-splits. Since this re-convergence scheme limits a warp-split's lifespan between branches and post-dominators, we name it *BranchLimited re-convergence*.

Although BranchLimited re-convergence allows DWS upon memory divergence while preserving the structure of the re-convergence stack, characterizations show it is likely to limit the benefit in latency hiding and MLP. As shown in Table 6.1, most benchmarks experience frequent branch instructions with only tens of instructions in between. Given such small basic blocks, a run-ahead warp-split is likely to reach the end of the basic block immediately and stall waiting for the fall-behind warp-split before beneficially issuing further memory requests. As a result, the performance gains from BranchLimited re-convergence are limited, as shown in Figure 6.11.

Run Ahead Beyond Branches and Loops

If run-ahead warp-splits can proceed beyond branches, they would have a better chance of issuing and overlapping further memory requests with the stall experienced by the fall-behind threads. Fortunately, we have already discussed in Section 6.4.2 how to use DWS to relax the re-convergence stack for branch divergence. Upon a future divergent branch, the run-ahead warp-split is subdivided into two warp-splits. As a result, the status of the

re-convergence stack remains intact while more entries are added to the warp-split table. These warp-splits re-converge when their PCs met. Otherwise, they are forced to wait for others to re-converge once they hit the post-dominator denoted by the re-convergence stack's TOS. We name this re-convergence scheme *BranchBypass*. Note that with Branch-Bypass, warps can be subdivided upon both memory divergence *and* branch divergence.

By allowing run-ahead warp-splits to proceed beyond
branches, it may appear harder for the fall-behind warp-split to catch up with the run-ahead. However, it is important to note that the ability to bypass branches naturally entitles the run-ahead warp-split to bypass loop boundaries into the next iteration. In such cases, the fall-behind warp-split may not have to execute the same number of instructions as the run-ahead to get re-united with it, especially if the loop body is short (*i.e.*, contains only a few instructions). In such a scenario, the run-ahead warp-split may frequently revisit the PC at which the fall-behind warp-split stopped. Once this PC is revisited and the run-ahead warp-split finds the fall-behind split is ready to execute, it re-unites with the fall-behind split immediately, despite potentially being a few iterations ahead of the fall-behind. In contrast to adaptive slip [159], which exploits this same opportunity, DWS warp-splits do not rely on short loops to re-converge: the fall-behind warp-split always resumes itself upon completion of data requests so that it can catch up with the run-ahead split in the case of a large loop body.

6.5.4 Implementation

Dynamic warp subdivision upon memory divergence is handled *in the same way* as DWS for branch divergence, except that the branch outcome that is used to divide threads is replaced by a *hit mask* that marks threads that hit the cache. Figure 6.12 illustrates the process. After memory divergence, two warp-splits are created: one with threads that hit

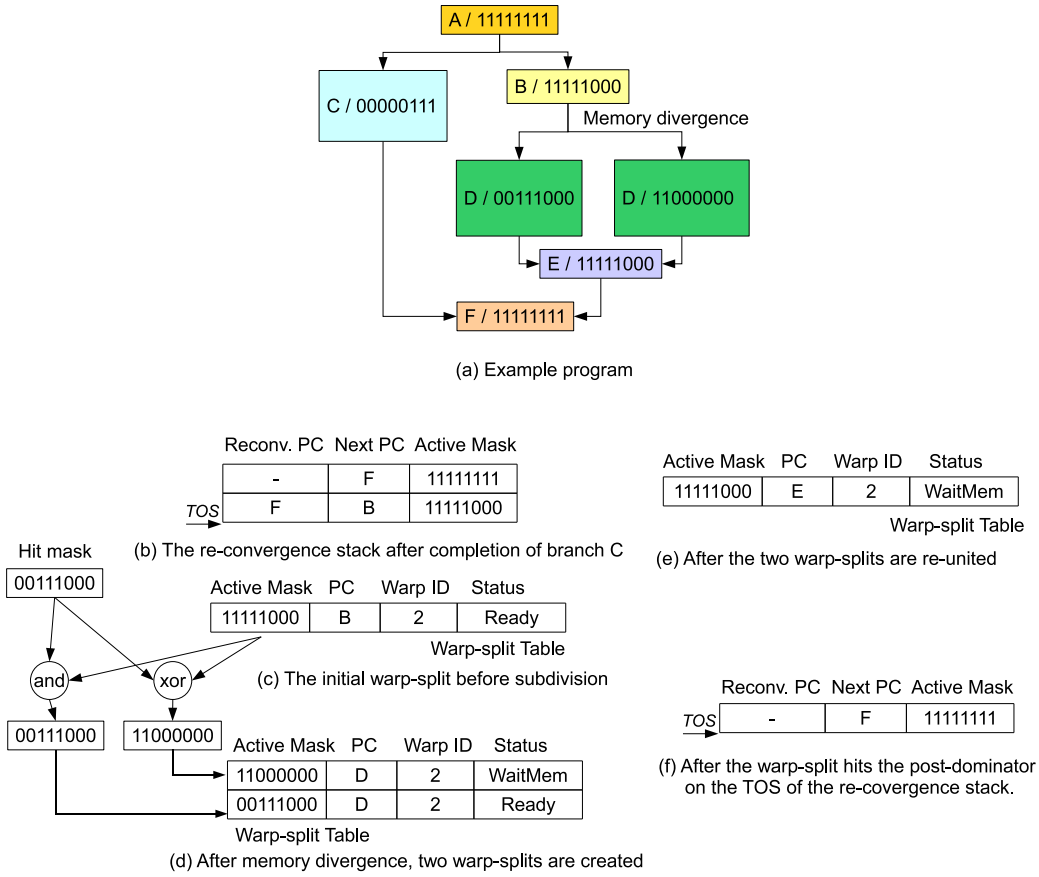


Figure 6.12: An example of dynamic warp subdivision upon memory divergence. While divergent branches can still be handled using the re-convergence stack (b, f), the warp-split table can be used to create warp-splits using the *hit mask* that marks threads that hit the cache (c-e).

and are ready to execute; another with threads that miss and are waiting for memory. Note that after subdivision, the entry of the obsolete parent warp-split is overwritten by one of the resulting child warp-splits so that *the WST only records existing warp-splits.* Moreover, creating a new scheduling entity does not require any other new state. Warp-splits still share the same register file resources, etc. Once the two warp-splits re-unite, their WST entries are merged into one by taking an "or" operation on their active masks. The hardware

overhead is discussed in more detail in Section 7.5.4. There is no need to differentiate warp-splits resulting from memory divergence with those resulting from branch divergence. *Any* warp-split can be further subdivided upon either branch or memory divergence.

Divergence does not break synchronization or communication semantics if programming models do not implicitly *guarantee* threads in a warp to operate in lockstep. Otherwise, DWS is not compatible because it allows warp-splits to proceed asynchronously for variable periods of time. With DWS, inter-thread synchronization or communication is only guaranteed through the use of explicit synchronization primitives, upon which warp-splits simply re-converge. Of course, frequent synchronization will limit intra-warp latency tolerance.

Compared to the baseline architecture, warp-splits merely change the ordering of execution for threads within the same warp. This does not affect memory exceptions. It only affects consistency for threads within the same warp. However, most SIMD programming models do not impose such sequential consistency. Finally, precise traps can still be handled for each individual warp-split.

6.5.5 Results

In Figure 6.13, we compare variations of dynamic warp subdivision and characterize the benefits of individual optimizations. DWS upon branch divergence (*DWS.BranchOnly*) alone reaches a speedup of 1.13X. DWS upon memory divergence alone using ReviveSplit (*DWS.ReviveSplit.MemOnly*) achieves a speedup of 1.20X. While DWS for branch and memory divergence are complementary and can be integrated, the overall benefit of DWS is sensitive to the combination of specific subdivision and re-convergence schemes. For example, *DWS.AggressSplit* and *DWS.LazySplit* combine BranchBypass with AggressSplit and LazySplit respectively, and they both lead to performance degradation. Nevertheless,

the combination of the best subdivision scheme (ReviveSplit) and re-convergence scheme (BranchBypass) does not harm performance in any case and it achieves an overall speedup of 1.71X (*DWS.ReviveSplit*). On average, *DWS.ReviveSplit* reduces the percentage of time in which WPUs stall waiting for memory from 76% to 36%; in the mean time, the average SIMD width per instruction drops from 14 to 4.

Different benchmarks exhibit different responses to dynamic warp subdivision. Merge benefits mainly from DWS upon branch divergence. KMeans, HotSpot, LU, and Filter benefit mainly from DWS upon memory divergence. FFT and SVM, on the other hand, have many cache misses but a large portion of these misses are not divergent, therefore warp subdivision occurs less often compared to other benchmarks. It is also observed that memory divergence may not be uniformly distributed across SIMD threads due to run-time dynamics. In fact, the pattern varies across benchmarks or even phases of execution. It is therefore difficult to statically pinpoint threads or lanes for warp subdivision.

6.5.6 Hardware Overhead

Because DWS does not increase the demand for tags and TLB ports, the hardware cost only lies in the WST and the scheduler. For a given WPU, the maximum number of warp-splits may become as large as the number of threads if each warp-split consists of a single thread. With a large number of threads per WPU, this may drastically increase the complexity of the hardware scheduler (*e.g.*, a priority encoder) in order to identify the next ready SIMD group in one cycle. We therefore model a scheduler that only doubles the number of entries in a conventional setting. This approximately doubles the cost of any scheduling structure, but can accommodate more scheduling entities resulting from DWS. In case there are more warp-splits than scheduling slots, the extra warp-splits sit idle until a scheduling slot is available. We also limit the number of WST entries to 16 to reduce its storage cost; warps

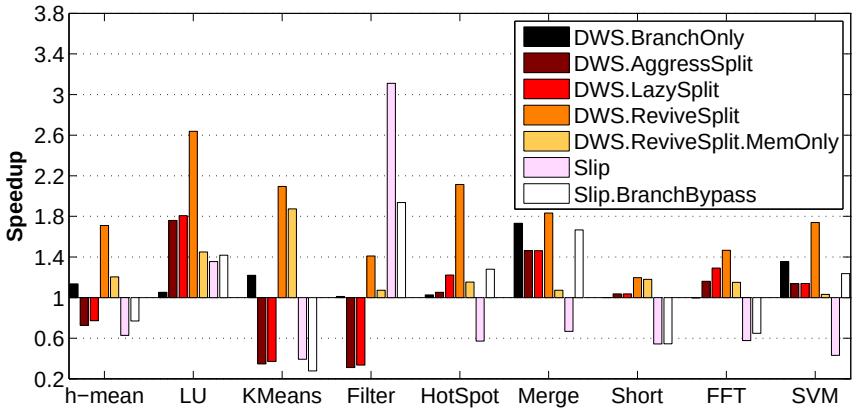


Figure 6.13: Comparing various DWS schemes. Speedups are normalized to that of equivalent baseline WPUs without DWS. Modest speedups are achieved when DWS applies to branch divergence alone (*DWS.BranchOnly*) or memory divergence alone (*DWS.ReviveSplit.MemOnly*). However, the combination of the two achieves a speedup of 1.71X (*DWS.ReviveSplit*). Slip only outperforms DWS in Filter and it is often subject to performance degradation, even after being modified to bypass branches (*Slip.BranchBypass*). The harmonic mean of the speedups for all benchmarks is shown as `h-mean`.

are not able to be subdivided when the WST is already full. Sensitivity study shows that the above limitations only cost less than 1% of the overall performance.

To estimate area overhead, we measure the realistic sizes for the different units of a core according to a publicly available die photo of the AMD Opteron processor in 130nm technology. We scale the functional unit areas to 65nm, assuming a 0.7 scaling factor per generation. We assume each SIMD lane has a 32 bit data path (adjacent lanes are combined if 64 bit results are needed). We also measure the cache area per 1 KB of capacity and scale that according to the cache capacity. If the WPU has four warps with a SIMD width of 16, each entry needs 16 bits for the active mask, 2 bits for the warp ID, 64 bits for the PC, and 2 bits for the warp status. The resulting size for each WST entry is then 84 bits or 11B. With up to 16 WST entries, the WST state consumes less than 1% of the storage area in a

WPU, assuming the WPU has a 32 KB D-cache and a 16 KB I-cache.

6.5.7 Comparison with Adaptive Slip

Tarjan *et al.* [159] proposed *adaptive slip*, which allows a subset of threads to continue while other threads in the same warp are waiting for memory. Those threads that wait for memory stay suspended until the run-ahead threads in that warp revisit the same memory instruction again, presuming memory divergence mainly occurs within iterative short loops. Their approach presumes aggressive predication so that run-ahead threads can always "slip" into the next iteration regardless of branch divergence.

We compare the performance of DWS to that of adaptive slip without aggressive branch predication, which we refer to in Figure 6.13 as *Slip*. To adaptively find out the maximum allowed thread divergence for adaptive slip, an interval of 100,000 cycles is used for dynamic profiling. The maximum allowed thread divergence is incremented if the WPU spends more than 70% of the time waiting for memory, and it is decremented if the pipeline actively executes for more than 50% of the time. These thresholds are obtained by experimenting with various combinations and selecting the combination that yields the best performance.

While adaptive slip leads to significant speedup for Filter, it results in performance degradation for many other benchmarks. The reasons are three-fold:

- Without aggressive predication, the run-ahead threads have to stall waiting for the fall-behind upon a conditional branch. This is because the re-convergence stack needs to generate the branch outcome for all threads masked by the TOS. As a result, adaptive slip is hardly effective when the main computation involves frequent conditional branches inside loops. Such is the case with HotSpot, Merge and Short.

- A diverged warp may not be re-united in time. This can be caused by long loops

(*i.e.*, loops with a large section of code in each iteration) where slipping into the next iteration may take a long time (*e.g.*, FFT). The same situation may also occur in the case of nested loops where memory divergence takes place inside the outer loop but outside of the inner loop; the fall-behind threads cannot be resumed until the run-ahead threads jump out of the inner loop (*e.g.*, KMeans).

- The appropriate thresholds to increase or decrease the maximum allowed thread divergence vary across benchmarks and hardware configurations. The same thresholds may work well for Filter while penalizing the performances of SVM.

We also attempt to combine adaptive slip with DWS upon branch divergence so that run-ahead threads can continue beyond branches to "slip" into subsequent iterations. This scheme is referred to in Figure 6.13 as *Slip.BranchBypass*. While this scheme significantly improves performance for many benchmarks, it does not address all the issues with adaptive slip and still harms performance for KMeans, Short, and FFT. Moreover, if memory divergence occurs within a branch path that is rarely visited and the run-ahead threads proceed to the next iteration, it may take a long time before the run-ahead can branch into the same path and re-converge with the fall-behind threads. In this case, this scheme can even perform worse than adaptive slip, as can be observed from KMeans.

6.6 Sensitivity Analysis

The benefit of dynamic warp subdivision in latency hiding and memory level parallelism depends on various factors: the frequency of branch and memory divergence, the length of memory latencies, and the WPU's ability to hide latency with existing warps. We study its sensitivity to various architectural factors. We show that DWS can bring performance gains in a wide range of configurations. In this section, we use the configuration *DWS.ReviveSplit*

to represent the performance of DWS which can subdivide warps upon both branch and memory divergence. We refer to the conventional architecture without DWS as *Conv*.

6.6.1 Cache Misses and Memory Divergence

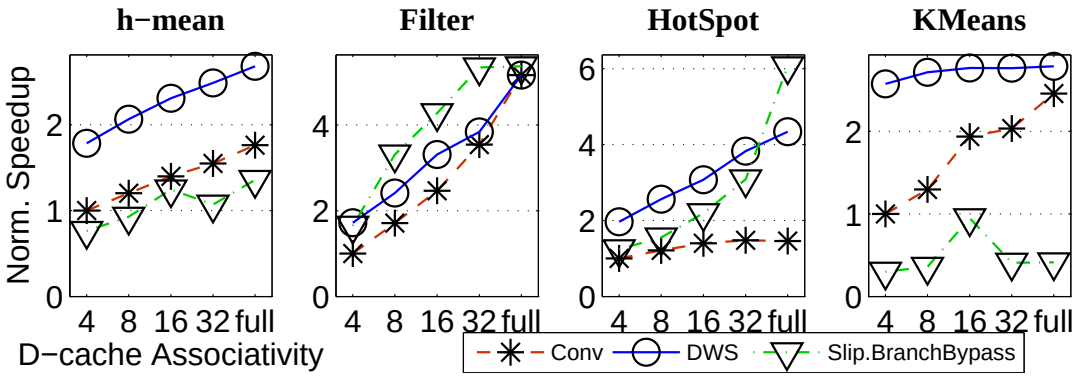


Figure 6.14: Speedup vs. D-cache associativity. DWS refers to *DWS.ReviveSplit* in Figure 6.13. Systems are configured according to Table 7.1. Performance is normalized to each benchmark's execution time under *Conv*. The harmonic mean for the normalized speedup of all benchmarks is shown as `h-mean`. Several individual applications with diverse behavior are also shown.

With larger D-cache associativity, both cache miss rate and memory divergence tend to decrease, and hence there is less latency to hide and less memory level parallelism to exploit. As a result, DWS may occur less often and its benefit may decrease. Figure 6.14 demonstrates this trend when the D-cache associativity is varied from four to fully-associative.

It can also be observed from Figure 6.14 that the benefit from DWS does not always increase with smaller D-cache associativity. Because small D-cache associativity leads to more cache misses, threads in a SIMD group are more likely to miss the cache simultaneously, and therefore the the occurrence of divergent memory accesses may decrease. As a

result, DWS upon memory divergence is less likely to take effect — in benchmarks such HotSpot, performance of DWS drops more than that of the conventional architecture when the D-cache associativity decreases to four.

6.6.2 Miss Latency

The amount of latency to hide also affects the effectiveness of DWS. The D-cache miss latency is largely dependent on the memory hierarchy. While most of the evaluation in this chapter is based upon a shared L2 cache as the last-level cache, there are various alternatives. Some architectures use private L2 caches while others do not have L2 caches at all and L1 misses are sent directly to the main memory [13]. The L1 miss latency can therefore vary from a few cycles to hundreds of cycles.

We study the sensitivity of the L1 miss latency by varying the L2 lookup latency from 10 cycles to 300 cycles. Results in Figure 6.15 shows that while DWS also suffers from longer miss latency, its speedup compared to the equivalent conventional architecture *increases*. This is not surprising because more SIMD groups are needed to hide longer latency; DWS achieves this by generating more warp-splits on demand.

6.7 Related Work

Fung *et al.* [154] addressed the under-utilization of the SIMD resources due to branch divergence. They proposed dynamic warp formation (DWF) in which diverged threads that happen to arrive at the same PC, *even though they belong to different warps*, can be grouped and run together as a wider SIMD group. However, if a subset of threads in a warp miss the cache and stall, there is still no way to allow the threads that hit to continue: this behavior is not compatible with the active mask on top of the re-convergence stack.

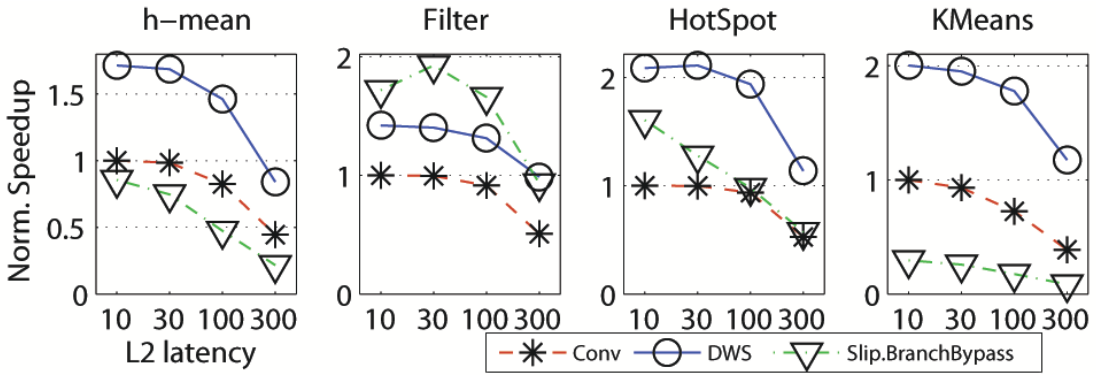


Figure 6.15: Speedup vs. L2 lookup latency. DWS refers to *DWS.ReviveSplit* in Figure 6.13. The speedup of DWS compared to *Conv* increases with longer L2 latency. Systems are configured according to Table 7.1. Performance is normalized to each benchmark's execution time under *Conv* with an L2 lookup latency of 10 cycles. The harmonic mean for the normalized speedup of all benchmarks is shown as `h-mean`.

As Table 6.1 has shown, divergent memory accesses can be more common than divergent branches in some applications. Furthermore, dynamic warp formation only occurs when multiple warps arrive at the same PC; if such condition is not met, the conventional reconvergence stack still prevents threads in the same warp from hiding each other's latency upon branch divergence. Our approach instead creates separate scheduling entities so that each warp-split can execute independently. This means that even after a divergent branch, threads on both paths are able to make progress and hide each others' latency. By creating new scheduling entities after each branch or miss divergence, our approach fully integrates branch and miss handling.

Similar to dynamic warp formation, stream processors such as Imagine can split streams conditionally according to branch outcomes [155], and each resulting stream follows the same control flow. Although conditional streams reduce unnecessarily predicated computation and communication, they do not address latency hiding. In fact, latency hiding is not necessary for stream processors; data is always available through the stream register file.

For this reason, memory divergence never occurs as well.

Our technique is complementary to Vector Lane Threading (VLT) [160] and the Vector-Thread (VT) Architecture [161]. VLT assigns groups of lanes to different user-level threads; lanes belonging to the same user-level thread execute in SIMD, but they do not need to execute in lockstep with lanes in other groups. However, lanes belonging to the same SIMD group may still stall unnecessarily due to branch and memory divergence. VT is a MIMD implementation of vector semantics, where each lane has the autonomy and overhead of its own fetch/decode/scoreboard logic. It is therefore not a strict SIMD organization and not subject to the divergence phenomena addressed in this chapter. Nevertheless, the principles of DWS may also apply to VT when it executes data-parallel applications.

Adaptive slip [159] addresses memory divergence to improve latency hiding for SIMD organizations. It employs a mechanism that is similar to a re-convergence stack, by simply using the cache access outcome instead of branch outcome so that threads that hit can continue. Similar to using the re-convergence stack, threads exhibiting divergent behavior cannot interleave their execution to hide latency, which limits benefits. It also assumes aggressive branch predication to avoid forced re-convergence at conditional branches within a loop body. Section 6.5.7 compares adaptive slip with DWS in more detail.

Existing techniques that address long latency memory accesses in the context of simultaneous multi-threading (SMT) do not help in the case of SIMD because SMT threads share a single datapath. With SMT, the problem is resource contention and reduced instruction level parallelism (ILP), since a stalled thread occupies expensive issue queues, rename-registers, and reorder-buffer entries, which limits ILP discovery for the other thread. Techniques for SMT resource distribution [162, 163] do not apply to in-order SIMD organizations, since SIMD threads operating on different lanes do not compete for pipeline resources. Instead, the main problem raised by long latency memory accesses is the risk of stall cycles due to inadequate latency hiding. Pre-computation using speculative threads [63]

or runahead threads [62] improves MLP. However, these speculative approaches require run-time dependency analysis among instructions as well as the ability for out-of-order execution and commit. These requirements are usually not met with simple, in-order SIMD hardware. We provide a solution designed specifically for SIMD hardware that allows it to exploit more MLP without speculative execution.

We have not considered the effect of prefetching or non-blocking loads. Nevertheless, DWS is complementary to both. It helps even in cases where the access pattern is not easily prefetched. Moreover, prefetching may increase contention for cache and bandwidth, especially when contention among many threads is severe. On the other hand, non-blocking loads employed by Tesla [13] and Fermi [14] are still in-order issue so they improve MLP but provide minimal latency hiding benefit. In both cases, memory divergence still occurs if threads in a warp are not all runnable at the same time.

Dynamic warp subdivision is also complementary to MLP-aware cache replacement [164]. While DWS creates additional scheduling entities to hide latency and issue more overlapping requests, MLP-aware cache replacement
smartly prioritizes these outgoing requests to reduce their aggregate latency.

Chapter 7

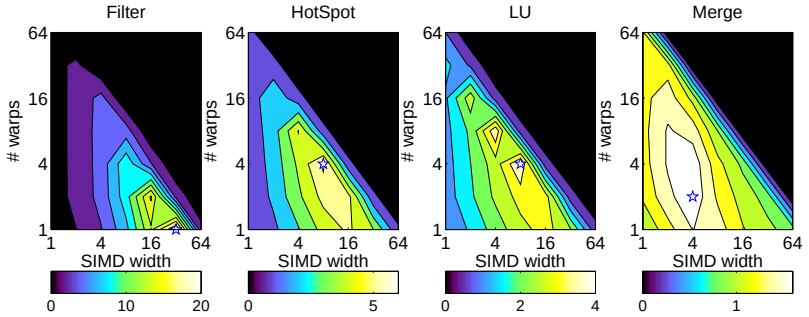
Robust SIMD: Dynamically Adapted SIMD Width and Multi-Threading Depth

7.1 Introduction

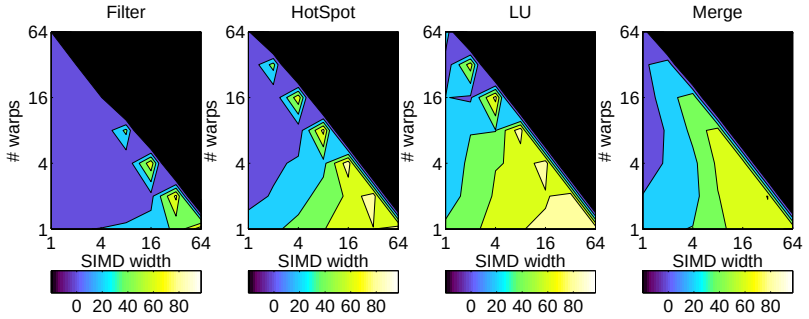
Single instruction, multiple data (SIMD) organizations are extensively used for computing data-parallel workloads including scientific computation, media processing, signal analysis, and data mining [20, 98, 97]. General purpose SIMD architectures include the Cell Broadband Engine (CBE) [9], Clearspeed [10], and Larrabee [15]. Graphics processors (GPUs), including NVIDIA's Tesla [13], Fermi [14], and ATI's recent architectures [165], also employ SIMD organizations and are increasingly used for general purpose computing. Academic researchers have also proposed stream architectures that employ SIMD organizations [166].

By using a single instruction sequencer to control multiple datapaths, SIMD organizations amortize the cost of instruction fetch, decode, and issue to save area and power. SIMD can operate multiple datapaths in the form of a vector or in the form of an array with a set of scalar datapaths. The latter is referred to by NVIDIA as *single instruction, multiple threads* (SIMT). Our work is applicable to both organizations. For the purpose of generality in this chapter, an instruction sequence associated with either a single element in a vector or a scalar in an array is referred to as a *thread*. The set of threads that progress in lockstep is referred to as a *warp*. The number of threads in a warp is referred as *SIMD width*, or *width* for short in this chapter. The banked register files and execute units are divided into *lanes*; each form a datapath. Each thread has its register file residing in one of the lanes. The set of hardware units under SIMD control is referred to as a *warp processing unit* or WPU. Unlike out-of-order processors, WPU processing elements are usually in-order to optimize throughput given a fixed area budget. Because threads have limited ability to execute beyond a cache miss, a WPU usually time-multiplexes several warps to overlap memory accesses of one warp with computation of another warp. The number of warps within a WPU is referred to as *multi-threading depth*, or *depth* for short in this chapter.

Conventionally, SIMD organizations come with a fixed width and depth. Such a fixed SIMD setup, however, cannot cater to the needs of diverse applications or execution phases. This is because the choice of preferred SIMD width and depth depends on various runtime factors including degree of data parallelism, demand for latency hiding, and intensity of cache contention, as well as frequency of branch divergence (*i.e.* threads in the same warp take different branch paths) and memory latency divergence (*i.e.* when a warp executes a memory instruction, some threads miss the cache and some hit). Although wider SIMD generally exploits more data parallelism and deeper multi-threading generally hides more latency, both increase thread count, which may in turn exacerbate L1 contention. It may



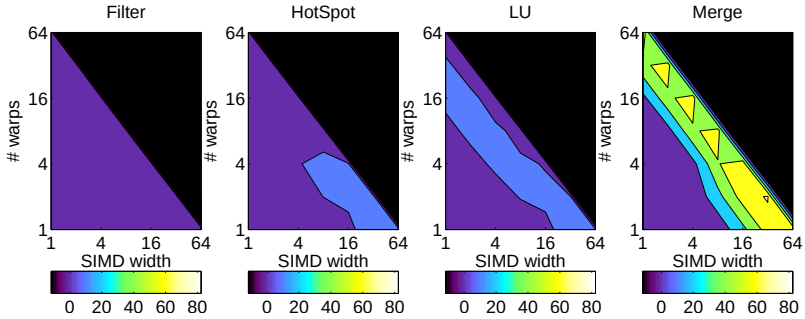
(a) Speedup



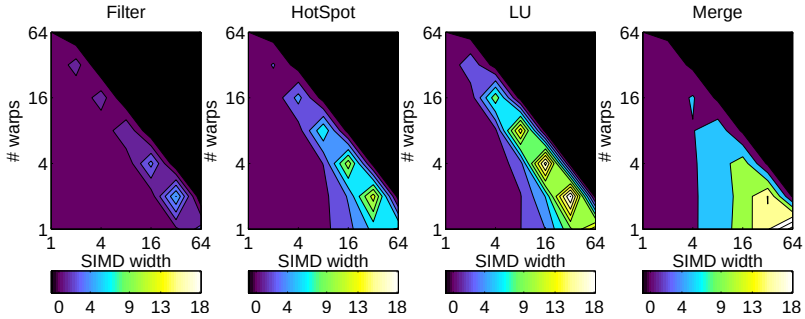
(b) Percentage of time that WPUs stall waiting for memory

Figure 7.1: Space exploration of various SIMD widths and multi-threading depths. We experiment with SIMD widths and depths at powers of two. Each WPU has a private I-cache and D-cache and all L1 caches share a last level cache. More details are described in Section 7.4. We chose four benchmarks that demonstrate diverse behavior.

appear that multi-threading can hide any extra latency caused by cache contention; however, our experiments show that deep multi-threading easily leads to cache thrashing, and the resulting cache latency grows beyond the threshold that can be hidden by available depth; this same phenomenon is also observed by Guz *et al.* [167]. Finally, wider SIMD is also likely to waste cycles due to pipeline stalls caused by branch and memory latency divergences [68]. In particular, memory latency divergence becomes increasingly common when SIMD organizations now support *gather loads* (*i.e.* load a vector from a vector of arbitrary addresses) or *scatter stores* (*i.e.* store a vector to a vector of arbitrary addresses)



(a) D-cache miss rate (%)



(b) Avg. number of branch and memory latency divergences per 100 instructions

Figure 7.2: Space exploration of various SIMD widths and multi-threading depths. We experiment with SIMD widths and depths at powers of two. Each WPU has a private I-cache and D-cache and all L1 caches share a last level cache. More details are described in Section 7.4. We chose four benchmarks that demonstrate diverse behavior.

over cache hierarchies [14, 96, 13, 15]. Figure 7.2b illustrates such trend. Because all above factors vary across different applications, the applications may prefer different SIMD width and depth, as shown in Figure 7.1.

Performance may also fall when SIMD organizations with fixed width and depth operate over emerging resizable memory systems. In energy-aware designs, researchers have proposed to reduce L1 and L2 cache capacity to save power. These techniques either power down cache segments [168, 169] or replicate cached data to endure errors at low voltages [170]. In the latter case, up to 50% of cache capacity can be sacrificed. However, the

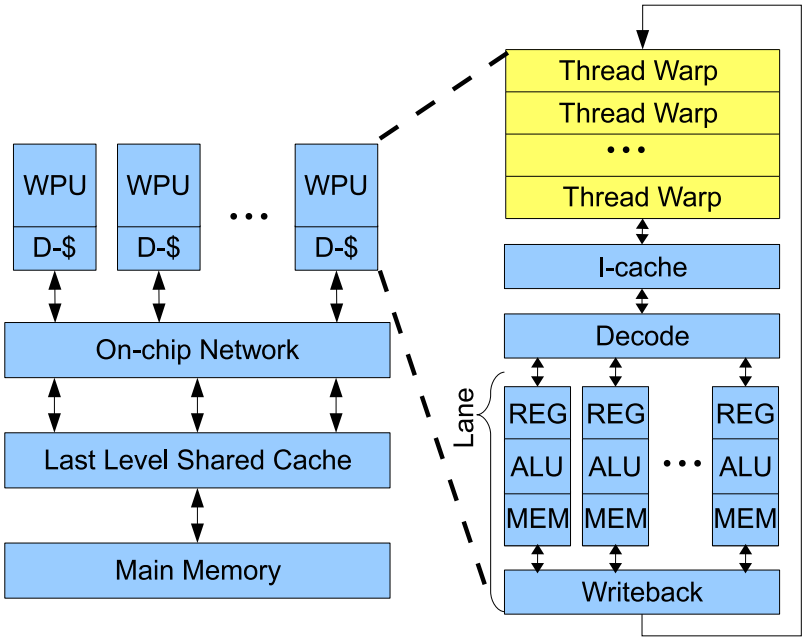


Figure 7.3: The baseline SIMD architecture groups scalar threads into warps and executes them using the same instruction sequencer. A thread operates over a scalar pipeline or lane that consists of register files, ALUs, and memory units.

reduction in cache capacity may significantly increase cache contention for the predetermined SIMD width and depth.

The key is to improve SIMD *performance robustness* so that the same SIMD organization works uniformly well for diverse applications and memory systems. This chapter proposes *Robust SIMD*, which dynamically optimizes and re-scales SIMD width and depth. With Robust SIMD, we propose the building of wide SIMD organizations in the first place. During execution, the WPU can appropriately reduce SIMD width and depth, or transition from wider SIMD to deeper multi-threading by splitting a wider warp into multiple narrower *warp-slices*. We explore several adaptation strategies and compare their effectiveness. The technique is evaluated by simulating eight diverse, data-parallel benchmarks

running on a chip multi-processor (CMP) with four WPUs over a two-level cache hierarchy. The system is illustrated in Figure 7.3. The simulation is intended to capture the trends of future architectures where CPUs are moving towards throughput-oriented cores and GPUs are moving towards implicitly managed memory systems (*e.g.* Intel's Larrabee [15] and NVIDIA's Fermi [14] both have SIMD over cache hierarchies). Compared to the performance resulted from running every benchmark on its individually preferred SIMD organization, the *same* Robust SIMD organization achieves similar, or sometimes even better performance due to phase adaptation. It outperforms the best fixed SIMD organization by 17%. It is also shown that even with dynamic warp subdivision (DWS), a fixed SIMD organization that performs best with 32 KB D-caches is only able to yield less than 50% of the optimal performance when D-cache capacities decreases to 16 KB. After combining Robust SIMD and DWS, we achieve 99% of the optimal performance persistently. In fact, the resulting performance is 97% of that achievable by a conventional organization with D-caches that double in size.

Overall, the benefits of Robust SIMD are fourfold:

1. It achieves near-optimal performance for diverse applications using the same SIMD organization.

2. It may adapt to non-standard SIMD widths and depths (*i.e.* not a power of 2) whose performance surpasses any conventional organizations.

3. It enables underlying caches to reduce their capacity and save power with minimal performance loss.

4. It reduces design complexity by reducing the SIMD design spaces to explore, advocating adaptive, wide SIMD organizations that appropriately scale their widths and depth at runtime.

This work was submitted to MICRO-43 [69].

7.2 Related Work

Several techniques have been proposed to adjust a vector organization according to traits in the workloads. Vector lane threading (VLT) [160] assigns groups of lanes to different user-level threads; lanes belonging to the same user-level thread execute in lockstep. The length of the vector is determined statically; short vectors are only used when the application has limited data parallelism. The vector-thread (VT) architecture [161], on the other hand, is a MIMD implementation of vector semantics, where each lane has the autonomy and the overhead of its own fetch, decode, and scoreboard logic. VT switches between SIMD and MIMD execution according to application parallelism. While VLT and VT exploit both data-level and thread-level parallelism, their static approaches predetermine the SIMD width for a given application. In fact, if there is ample data parallelism, SIMD width is set as large as the hardware can support, and both VLT and VT execute just like conventional SIMD organizations. Neither SIMD width nor multi-threading depth is adjusted according to runtime dynamics. Therefore, our technique is complementary to both VLT and VT.

Fung *et al.* [154] addressed under-utilization of SIMD resources due to branch divergence. They proposed dynamic warp formation (DWF), in which divergent threads that happen to arrive at the same PC, even though they belong to different warps, can be grouped and run as a wider warp; however, DWF does not adapt overall SIMD width or multi-threading depth. For applications with no branch divergence, DWF would execute in the conventional way, despite that adjusting SIMD width and multi-threading depth may improve latency hiding or reduce cache contention.

Memory latency divergence for SIMD architectures are addressed by adaptive slipping [159] and dynamic warp subdivision (DWS) [68]. These techniques allow threads

that hit the cache to run ahead while those that miss are suspended. The run-ahead threads can then prefetch data for the fall-behind and hide latency. DWS also allows warps to split upon divergent conditional branches. These techniques differ from our approach in several aspects. First, all threads are actively executed until they terminate, and there is no way to turn off some lanes or deactivate a few warps to reduce cache contention. Second, if divergence is rare and cache misses usually occur for all threads in the same warp, DWS would be unable to adjust SIMD width or depth. Finally, these techniques apply heuristics such as splitting warps according to cache hits or misses, which may risk performance degradation due to over-subdivision [68]. In contrast, Robust SIMD reconfigures width and depth experimentally based on the end performance results, rather than heuristically based on individual cache accesses or branches. We compare Robust SIMD with DWS in Section 7.6.2 and show that when cache capacity reduces, the SIMD organization that DWS originally anchored upon would suffer from dramatic performance degradation, while the same Robust SIMD organization would still generate near-optimal performance.

Symbiotic affinity scheduling (SAS) [65] was proposed to improve inter-thread temporal locality for multi-threaded cores by dispatching data-sharing tasks to concurrent threads. Using SAS, symbiotic tiling (S-tile) was introduced for data-parallel applications and it can be applied to SIMT organizations. SAS improves cache sharing to some extent but is not able to reduce thread count when too many threads thrash the cache. Neither SIMD width and depth nor adapted in SAS, therefore Robust SIMD is complementary to SAS. In all our experiments, we use SAS to schedule data-parallel tasks.

There also exist many scheduling techniques to reduce cache contention for non-SIMD chip multi-processors (CMPs). These techniques typically select a few threads from a pool of threads whose joint working set minimizes cache contention [40, 41, 42]. These techniques exploit the heterogeneity in threads' working sets and they do not consider data-parallel, homogeneous tasks. Moreover, they aim to find *which* threads to execute

concurrently, without considering *how* threads are mapped to hardware thread contexts. SIMD organizations raise the issue of how to organize concurrent thread contexts into warps, requiring specifications of both SIMD width and multi-threading depth, which are not addressed by previous techniques.

7.3 An Adaptive SIMD Architecture

We describe the software and hardware systems that enable Robust SIMD to reconfigure its SIMD width and multi-threading depth at runtime. Figure 7.4 illustrates the process of a WPU adapting to a data-parallel code section. It takes two phases to execute data-parallel tasks within a parallel `for` loop. First, in the *adaptation phase*, a WPU executes a few tasks during one sampling interval, measures the performance within this period, and feeds this measurement back to an *Adaptation State Unit (ASU)* which evaluates the previous configuration and suggests a new one. The WPU then reconfigures itself according to the suggested SIMD width and multi-threading depth, and repeats this process until the ASU converges on a preferred configuration. Second, in the *execution phase*, the WPU maintains the converged configuration until all tasks in the current parallel section are completed. We assume that data-parallel tasks within a parallel `for` loop exhibit similar behavior so that the SIMD configuration preferred by the sampled tasks will also be preferred later by other tasks. Section 7.5.1 compares several adaptation strategies that can be employed by the ASU. The rest of this section focuses on the software and hardware modifications to allow both SIMD width and multi-threading depth to be adjusted at runtime.

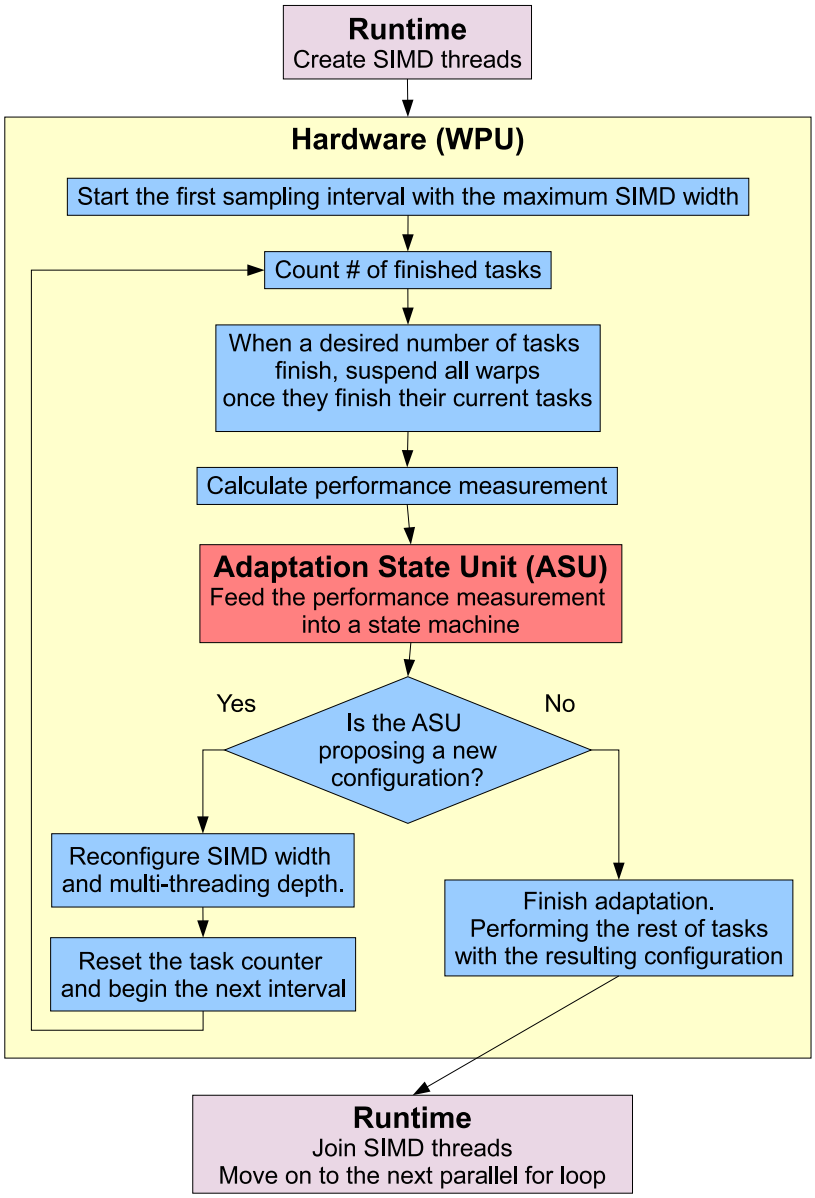


Figure 7.4: The process of adapting a WPU's SIMD width and multi-threading depth to a particular data-parallel section of code.

7.3.1 Software Support for Reconfigurable SIMD

To allow SIMD reconfiguration, the programming model should not expose a fixed SIMD width to the programmer. In fact, many existing programming models already express

data-parallel tasks as scalar operations without specifying the vector length. Examples are Cg [77], OpenMP [102], OpenCL [171], and CUDA [78]. With SSE extensions in GCC, the appropriate SIMD width can also be inferred from non-vectorized parallel `for` loops in general C code. Because the preferred SIMD width may vary dynamically, it cannot be determined by compilers either. Nevertheless, data-parallel tasks can be compiled as scalar operations. It is then up to the runtime system to determine how to group them into warps in the form of vector or array.

The runtime can group scalar operations into SIMD groups by simply assigning scalar threads to available hardware thread contexts that operate in lockstep. Usually, data-parallel tasks belonging to the same parallel `for` loop start at the same PC, share the same arguments, and differ only in their loop indices. The runtime can pass different loop indices to individual thread contexts through specialized registers in the hardware.

7.3.2 Hardware Support for An Adaptive SIMD Architecture

The SIMD width and multi-threading depth are only adjusted at the end of a sampling interval. The number of lanes in each WPU sets the upper bound of its SIMD width. Within this limit, an adaptive WPU supports the following operations:

- Lanes can be turned off and on to adjust SIMD width.
- Warps can be suspended or resumed to adjust multi-threading depth.
- SIMD width can be traded for multi-threading depth by breaking a wide warp into multiple narrower *warp-slices* that can interleave their execution to hide each other's latency. These warp-slices can be re-united in future intervals as a wider warp. An original warp with full SIMD width can be regarded as a warp-slice as well.

In this section, we discuss the hardware units that enable Robust SIMD to perform the above operations. We also describe methods to measure performance at each sampling interval. We illustrate with a SIMT or array organization where branch divergence is handled by a *re-convergence stack* [154]: when threads in the same warp branch to different control paths, the WPU first executes threads falling into one path and suspends others; a bit mask representing those active threads are pushed on to the re-convergence stack and popped when the corresponding threads finish their current branch path; the WPU can then switch to execute threads falling into the alternate path. Our technique also applies to vector organizations. In fact, as we discusse in Section 7.3.2, the implementation would be simpler in vector organizations which have no re-convergence stacks.

Trading SIMD Width for Multi-Threading Depth

Threads in the same warp can be split into multiple sets of threads; each set forms a warp-slice whose threads operate in lockstep. The process is illustrated in Figure 7.5. A warp-slice table (WST) is used to keep track of warp-slices. Each warp-slice occupies an entry in the WST and uses *active masks* to mark its associated threads. Warp-slices are treated as independent scheduling entities similar to warps. In one cycle, only one warp or warp-slice can execute. Warp-slices belonging to the same warp can be merged again. Creating a warp-slice would also require duplicating its re-convergence stack for SIMT or array organizations (for vector organizations where branches are predicated, the WST alone would suffice). Physically, the duplicated re-convergence stacks can be combined, as shown in Figure 7.5(c). The hardware overhead introduced by warp-slices' independent re-convergence stacks is further discussed in Section 7.5.4.

The width of a warp-slice is determined by the SIMD width suggested by the ASU. We impose the constraint that all warp-slices share the same width so that a SIMD configuration can be simply parameterized as a tuple, denoted by $(width \times depth)$, which in

turn simplifies adaptation. Otherwise, the number of possible SIMD configurations will explode. Only when the available SIMD width is not divisible by the suggested SIMD width would those residue threads form warp-slices that are narrower than others.

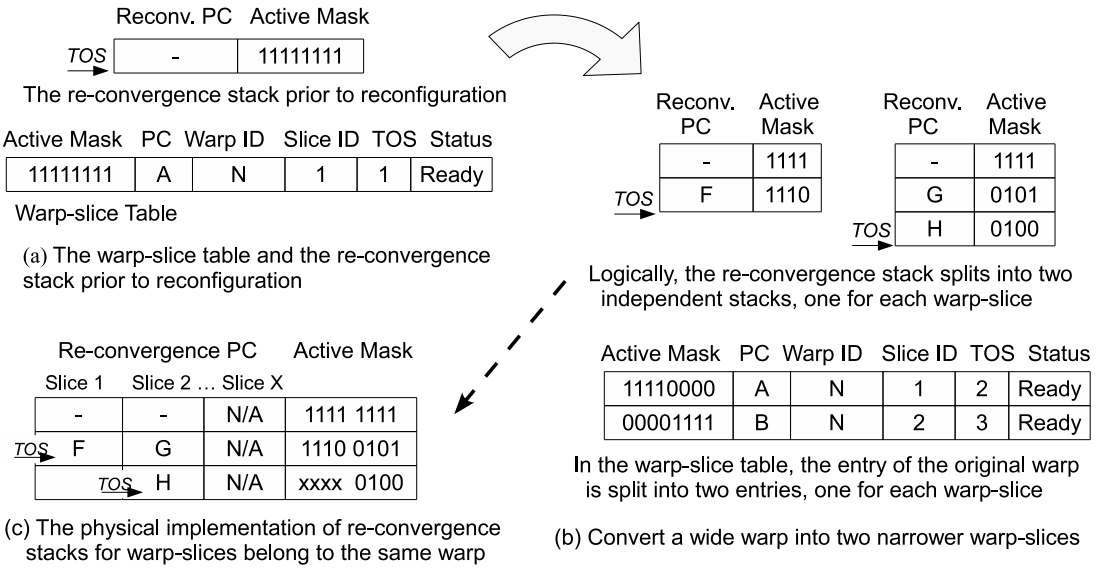


Figure 7.5: The warp-slice table and the re-convergence stack before and after a warp is split into two warp-slices. This only happens upon task completions when the re-convergence stack is empty. The single entry in the warp-slice table (a) is divided into two, as is the re-convergence stack (b). Physically, the re-convergence stacks for warp-slices of the same warp are combined into one (c). Modification of the re-convergence stack is only necessary for array organizations or SIMT. For vector organizations where branches are handled by predication, the warp-slice table alone would suffice.

Different from *warp-splits* in DWS, our technique constructs warp-slices regardless of threads' divergence histories, so that SIMD width and depth can be adjusted even when there is no memory latency divergence. To form a warp-slice, a WPU only has to define how many threads to be grouped into the new warp-slice, without specifying *which* threads to be grouped. We heuristically group threads operating on neighboring lanes into a warp-slice. Moreover, forming and merging of warp-slices only occur at the end of each sampling

interval, rather than upon individual cache hits or misses. Finally, each warp-slice has its independent re-convergence stack, allowing them to hide each other's latency in spite of the occurrences of conditional branches. In contrast, warp-splits belonging to the same warp share the same re-convergence stack, and may subject to frequent synchronization upon conditional branches, limiting the effectiveness in latency hiding.

Reconfiguring SIMD Width and Multi-threading Depth

The ASU suggests a new SIMD configuration at the end of an interval. We require that all thread contexts complete their current tasks before an interval can end, so that all thread contexts would have their re-convergence stack empty and they would arrive at the same PC.

The reconfiguration then takes two steps. First, all warp-slices are merged into complete warps by simply taking the "or" of their active masks in WST entries. Their re-convergence stacks can be combined straightforwardly because they are all empty. Second, warps are divided again; threads in neighboring lanes form a warp-slice with the desired SIMD width. Note that if the number of lanes is not divisible by the desired SIMD width, the remainder will form another warp-slice which can have a SIMD width smaller than desired. Only a subset of warp-slices are activated according to the desired multi-threading depth. If the desired SIMD width remains the same as the previous SIMD width and only the multi-threading depth is adjusted, the WPU can simply activate or suspend warp-slices without merging warp-slices and creating them again.

Delimiting Sampling Intervals

Our implementation requires a mechanism to signal a WPU when a thread context of its own completes a data-parallel task. This signal can be used to synchronize all thread contexts at the end of a sampling interval, so that they arrive at the same PC with an empty re-convergence stack. Such mechanism entitles narrower warp-slices belonging to the same warp to combine as a wider warp-slice. In addition, performance can be estimated by sending the task completion signal to a hardware counter which records the number of finished tasks within a sampling interval. Finally, we discuss in Section 7.5.3 about how to use such task completion signal to determine the appropriate length of a sampling interval.

To identify task boundaries, we construct the runtime data-parallel library in such a way that a thread's re-convergence stack is only popped empty when it finishes a task. Once a re-convergence stack is popped empty, it signals the WPU that one of its associate warp-slices just completed the assigned tasks. While alternate approaches exist, such as appending the existing instruction set architecture (ISA), such approaches are not the focus of this chapter.

7.4 Methodology

We use MV5 [64] to simulate WPUs. MV5 is a cycle-accurate, event-driven simulator based on M5, which was originally designed for a network of processors [84]. MV5 is chosen because it models SIMD and integrates WPUs with directory-based coherent caches and on-chip networks, and it can be used to simulate general purpose data-parallel applications. Because existing operating systems do not directly manage SIMD threads, applications are simulated in system emulation mode with a set of primitives to create threads in a SIMD manner.

The simulated applications are programmed in an OpenMP-style API implemented in MV5, where data parallelism is expressed in parallel `for` loops. We do not use the original Pthreads [172] or OpenMP [102] libraries because their current implementations are unable to create SIMD threads. Applications are cross-compiled to the Alpha ISA using G++ 4.1.0, with byte code explicitly inserted to signal SIMD management to the simulator. We use the O1 optimization level because higher levels of optimization sometimes misplace the inserted bytecode. The simulated runtime library is able to tile data-parallel tasks and execute them on available hardware thread contexts. We employ symbiotic tiling [65] to assign neighboring tasks to concurrent threads for locality optimization.

7.4.1 Simulation Infrastructure

To model WPUs, a single fetched instruction is executed by all threads within the same warp simultaneously. A re-convergence stack (see Section 7.3.2) is embedded into the hardware, which is pushed upon conditional branches with a bit mask marking threads falling into the corresponding path; it is later popped when the program executes a *post-dominator* that signals control flow re-convergence after a branch. Due to the lack of compiler support, we manually instrument application code with post-dominators. Each lane is modeled with an IPC of one except for memory references, which are modeled faithfully through the memory hierarchy (although we do not model memory controller reordering effects). A WPU switches warps upon every cache access with no extra latency—as modern GPUs do—simply by indexing appropriately into shared scheduling and register file resources. WPUs are simulated with up to 256 thread contexts and 64 lanes. Using our simulator, we study chip multi-processors (CMPs) with four homogeneous WPUs operating over cache hierarchies. Having more WPUs produces similar results but with longer simulation times.

The memory system is a two level coherent cache hierarchy with private L1 caches

Tech. Node	65 nm
WPU	1 GHz, 0.9 V Vdd, Alpha ISA, up to 256 hardware thread contexts a SIMD width of up to 64, in-order
I-Cache	16 KB, 4-way associative, 128 B line size 1 cycle hit latency, 4 MSHRs, LRU, write-back
D-Cache	32 KB, 8-way associative, 128 B line size MESI directory-based coherence 3 cycle hit latency, LRU, write-back 256 MSHRs each hosts up to 8 requests
L2 Cache	4096 KB, 16-way associative, 128 B line size 30 cycle hit latency, LRU, write-back 256 MSHRs each hosts up to 8 requests
Crossbar	300 MHz, 57 Gbytes/s
Memory	300 cycles access latency

Table 7.1: Hardware parameters used in studying Robust SIMD.

and a shared L2 as the last level cache (LLC). Each WPU has a private I-cache and D-cache. I-caches are not banked because only one instruction is fetched every cycle for all lanes. D-caches are always banked according to the number of lanes. We assume there is a perfect crossbar connecting the lanes with the D-cache banks. If bank conflicts occur, memory requests are serialized and a small queuing overhead (one cycle) is charged. The queuing overhead can be smaller than the hit latency because we assume requests can be pipelined. All caches are physically indexed and physically tagged with LRU replacement policy. Each Miss Status Holding Register (MSHR) hosts a cache line and it can hold up to eight requests if they all correspond to the same cache line. The L1 caches connect to L2 banks through a crossbar. Coherence is handled by a directory-based MESI protocol.

Table 7.1 summarizes the main architectural parameters. Note that the aggregate L2 access latency is broken down into L1 lookup latency, crossbar latency, and the L2 lookup latency. The L2 lookup latency contains both tag lookup and data lookup. The L2 then connects to the main memory through a 266 MHz memory bus with a bandwidth of 16 GB/s. The latency in accessing the main memory is assumed to be 300 cycles, and the memory controller is able to pipeline the requests.

7.4.2 Benchmarks

The throughput-oriented WPUs are targeted at data-parallel applications with large input data sets. We simulate a set of parallel benchmarks shown in Table 7.2. They are common, data-parallel kernels and applications originated from several benchmark suites including Minebench [98], Splash2 [97], and Rodinia [20]. They cover the application domains of scientific computing, image processing, physics simulation, and data mining. They represent data-parallel applications with varied data access and communication patterns [105]. We change the input sizes from the original benchmarks so that we have sufficient parallelism and still have reasonable simulation times.

	Benchmark Description
FFT	Fast Fourier Transform (Splash2 [97]). Spectral methods. Butterfly computation Input: a 1-D array of 262,144 (2^{18}) numbers
Filter	Edge Detection of an Input Image. Convolution. Gathering a 3-by-3 neighborhood Input: a gray scale image of size 500×500
HotSpot	Thermal Simulation (Rodinia [20]). Iterative partial differential equation solver Input: a 300×300 2-D grid, 100 iterations
LU	LU Decomposition (Splash2 [97]). Dense linear algebra. Alternating row-major and column-major computation Input: a 300×300 matrix
Merge	Merge Sort. Element aggregation and reordering Input: a 1-D array of 300,000 integers
Short	Winning Path Search for Chess. Dynamic programming. Neighborhood calculation based on the the previous row Input: 6 steps each with 150,000 choices
KMeans	Unsupervised Classification (MineBench [98]). Distance aggregation using Map-Reduce. Input: 30,000 points in a 8-D space
SVM	Supervised Learning (MineBench [98]). Support vector machine's kernel computation. Input: 100,000 vectors with a 20-D space

Table 7.2: Simulated benchmarks with descriptions and input sizes.

7.4.3 Data-Parallel Programming Models

While there are commercial software systems for data-parallel applications (*e.g.* TBB [103], OpenMP [102], Cg [77], CUDA [78], and OpenCL [171]), they either lack properties described in Section 7.3.1 or do not execute properly in MV5's system emulation mode.

Therefore, we adopted the data-parallel programming model and user-level runtime threading library on MV5 from Meng et al. [65]. A nested, parallel `for` loop is abstracted as a generic C++ class which contains arrays that indicate loop boundaries and strides, and a member function which encapsulates code sections within the innermost loop. By taking a particular loop index as its argument, an invocation of the member function computes an individual task corresponding to one innermost loop iteration. Such an API is only designed to mimic the programming interface in existing parallel APIs in MV5 simulations. The coded benchmarks can be cross-compiled to the Alpha ISA using GCC 4.1.0. Given appropriate compiler support or code translation, our technique can work with existing APIs without modifying applications' source code.

With this API, data-parallel tasks are expressed as scalar operations, and loop boundaries can be easily interpreted by our runtime library. The runtime library then partitions the loop into blocks and assigns each block to a WPU. The WPU iterates through the block and fills a set of specialized registers with neighboring loop indices, which are later passed to concurrent, individual thread contexts. A hardware thread context then fetches a loop index from its associated specialized register, calls the user-defined function representing the innermost loop body, and instantiates an individual data-parallel task. Threads belonging to the same warp-slice proceed in lockstep. When a task is completed, the thread context fetches another loop index if available.

7.4.4 Load Balancing

We evenly partition a nested `for` loop according to the number of WPUs. Experiments show that given large input data, the workload is balanced among WPUs most of the time. In rare circumstances where workloads are not balanced, over-decomposition can be used; a loop space can be partitioned into smaller blocks so that if a WPU finishes the previous

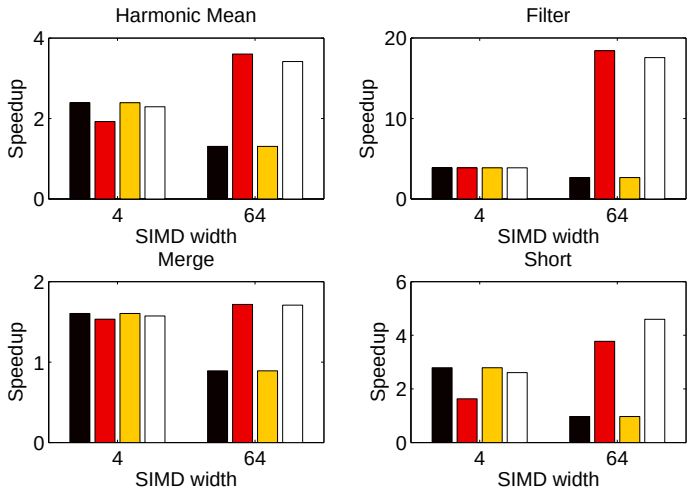
block early, it can fetch the next block. However, load balancing is not the focus of this chapter, and therefore we do not investigate over-decomposition.

Load balancing among threads on the same WPU is accomplished by using fine-grained task dispatching [65]. Whenever a loop index is fetched by a thread context, the hardware will refill the corresponding specialized register with a new loop index. As a result, threads that completed previous tasks are assigned more tasks immediately when available.

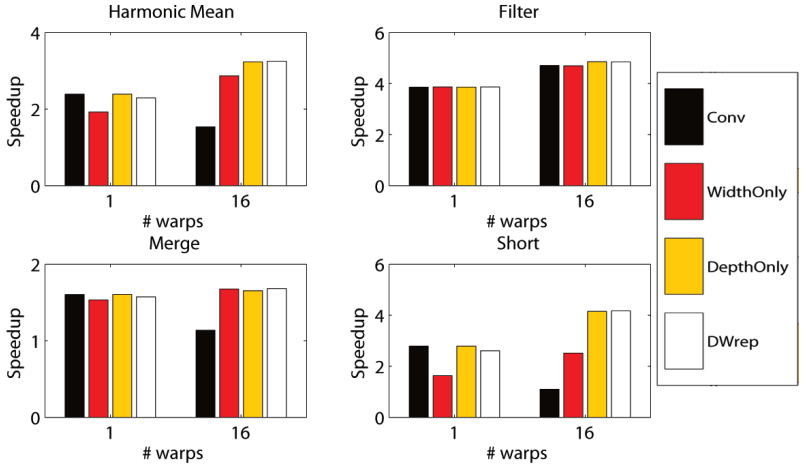
7.5 Adaptation Strategies

An adaptation strategy can be regarded as a simple finite state machine implemented by the adaptation state unit (ASU). In our simulation, the ASU acts as a hardware unit. The ASU determines the next SIMD organization according to performance feedback, increases or decreases SIMD width or multi-threading depth, and transits to the next state. The ASU can also use a lightweight, run-time callback function to determine the next SIMD organization. This function can be completed within tens of cycles, and is only called at the end of each sampling interval during the adaptation phase. Such overhead is negligible because one sampling interval usually takes at least thousands of cycles. Overall, the adaptation phase accounts for a small portion of the execution time and it does not scale with the input size.

We investigate various adaptation strategies and compare their effectiveness in optimizing SIMD width and multi-threading depth. All of them attempt to converge to their preferred configuration based on gradients. First, the ASU picks a dimension (*i.e.* SIMD width or multi-threading depth). It then attempts to increase and decrease the value along that dimension in the subsequent two intervals. The amount to increase or decrease is discussed in Section 7.5.2. The ASU then chooses the direction which yields better performance. In the following intervals, it suggests subsequent SIMD organizations in that



(a) Various SIMD widths with a single warp



(b) Various multi-threading depths with a SIMD width of 4

Figure 7.6: Comparing different adaptation strategies. Speedup is normalized to that of a system with single-threaded WPUs. With narrow SIMD and few warps, all adaptation strategies perform similarly. (a) When SIMD width increases from 4 to 64, *WidthOnly* and *DWrep* outperform *Conv* and *DepthOnly* since they trade SIMD width for depth. (b) When the number of warps increases from 1 to 16, *DepthOnly* and *DWrep* perform best because they are able to deactivate some warps to reduce cache contention.

direction until it no longer gains performance. At this point, it may choose another dimension to adapt. We have also tried greedy algorithms: instead of probing both directions before making a decision, the ASU would adapt along one direction as soon as it finds the direction beneficial. Our experiments show that such greedy algorithms are subject to noise and are more likely to converge to local minima.

7.5.1 Exploring Various Strategies

We investigate several adaptation strategies including:

- *WidthOnly*. Only the SIMD width is adjusted. The number of active warps remains constant. The preferred SIMD width is identified by turning off more lanes in subsequent sampling intervals until it begins to hurt performance.
- *DepthOnly*. Same as *WidthOnly* except that only the multi-threading depth is adjusted.
- *DWrep*. First, the multi-threading depth is adjusted as if in *DepthOnly*. Then, based on the resulting number of warps, different SIMD widths are evaluated in the same way as in *WidthOnly*. The process repeats until further changing either SIMD width or multi-threading depth no longer improves performance. Note that reducing SIMD width may generate more warp-slices, which increases the upper limit of multi-threading depth. We have also experimented with the same strategy with SIMD width adapted first, which yields similar performance.

We compare these strategies over various SIMD widths and multi-threading depths. The baseline system with conventional WPUs is referred to as *Conv*. Because of the space limitation, we only illustrate two scenarios in Figure 7.6. Our results show that *DWrep* performs persistently well on various organizations, therefore *DWrep* is used in Robust

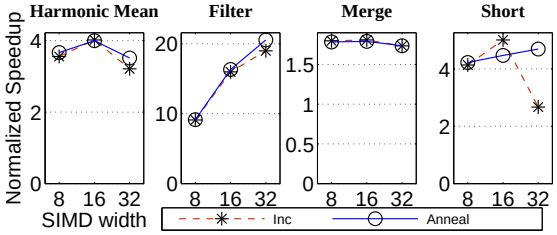
SIMD in the rest of this chapter. Note that when the initial configuration is already the optimal configuration, adaptive strategies may perform slightly worse than *Conv* due to adaptation phases with suboptimal configurations.

We have also experimented with adapting depth and width simultaneously and with allowing multiple WPUs to collaboratively explore and evaluate various SIMD organizations. These strategies are susceptible to noise and are outperformed by*DWrep*.

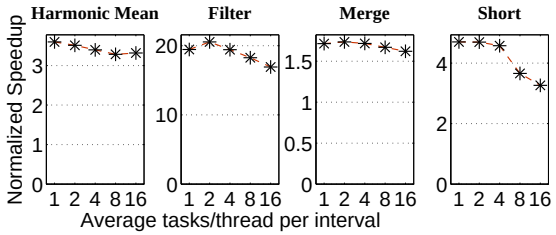
7.5.2 Convergence Robustness

During adaptation, the ASU has to offset the current SIMD organization with a certain stride to suggest the next configuration. A naïve implementation is to increment or decrement either SIMD width or multi-threading depth after each sampling interval. Such a convergence mechanism with a stride of one is named *Inc*.

However, our experiments show that *Inc* converges slowly and is susceptible to noise. To address these issues, we propose to first use a larger stride at beginning, and then gradually decrease the stride until it reaches one (i.e. incremental adjustment). We name this method *quasi-annealing* (*Anneal*), and it can be combined with any adaptation strategies described in Section 7.5.1. In our evaluation, the initial stride for SIMD width is set to half of number of available lanes. The initial stride for multi-threading depth is set to half of the number of available warps. The stride is reduced by half once a preferred configuration is identified using the coarser stride. The adaptation then continues with the finer stride until the stride reaches one. Figure 7.7a compares *Inc* and *Anneal* and shows that the latter performs better with wide SIMD.



(a) Comparing Robust SIMD performance with a converging stride of one (*Inc*) and that with annealed strides (*Anneal*). WPUs each have two warps and we vary the SIMD width.



(b) Comparing various sampling intervals on WPUs with two 32-wide warps.

Figure 7.7: Sensitivity study for convergence rate and interval length. Speedup is normalized to a system with four single-threaded, in-order cores. We show three benchmarks that exhibit different responses. The overall trend is shown by taking the harmonic mean of the performances across all benchmarks.

7.5.3 Length of Sampling Intervals

The length of the sampling interval plays an important role. We found that the appropriate sampling interval in cycles varies across applications. This is because the amount of computation within a data-parallel task differs significantly across applications or even application phases. A short sampling interval may fail to capture the execution of complete data-parallel tasks, which may lead to biased, noisy performance measurements and mislead adaptation. On the other hand, long sampling intervals prolong the adaptation time.

To address this issue, we propose to adjust the sampling interval according to task lengths. Prior to adaptation, we predefine a desired number of data-parallel tasks (*i.e.* executions of the code section within the innermost parallel `for` loop) to be completed by an average thread in one sampling interval. This number is set to two in the rest of the chapter based on results of the sensitivity study illustrated in Figure 7.7b. This number, multiplied by the thread count per WPU, gives the minimum threshold of the number of tasks to be completed by a WPU within one sampling interval. In each sampling interval, the WPU uses a hardware counter to count the number of completed tasks until it exceeds

the minimum threshold. Once such a threshold is reached, the WPU will suspend any active threads once they finish their current tasks; it will then reconfigure its SIMD width and depth. This charges an extra synchronization overhead, but our experiments show that the accumulated synchronization overhead accounts for less than 9% of the entire adaptation phase. The adaptation phase, in turn, accounts for an average of 6% of the total execution time in our experiments, and it does not scale up with larger input size.

By ending a sampling interval after completing a predefined number of data-parallel tasks, the length of the sampling interval is set to an appropriate granularity. The length of the interval in cycles is then divided by the number of completed tasks to estimate performance as cycles per task (CPT). The lower the CPT is, the more preferred the SIMD organization is. The cycles-per-task is used later by the ASU to guide adaptation. Note that different WPUs monitor their sampling intervals independently. The mechanism to signal task completion is described in Section 7.3.2.

However, there are cases where a parallel `for` loop contains only a few long-running tasks, resulting in insufficient tasks for adaptation. Merge, LU, and KMeans all have a portion of their execution time that exhibits such behavior, often in their latter phases of reduction. In these cases, we apply the heuristic that if a parallel `for` loop has too few tasks to warrant 30 sampling intervals, a default SIMD organization with two eight-wide warps will be applied instead.

7.5.4 Hardware Overhead

Additional hardware units introduced to an adaptive WPU mainly include the adaptation state unit (ASU), the warp-slice table (WST), and counters for completed tasks within one sampling interval. The ASU, if implemented in hardware, includes a comparator to evaluate performance measurements, an adder to offset the width and depth, a multiplier to scale the

offset strides, and several multiplexors to choose the appropriate strides and directions. It also includes several registers to store current SIMD width and depth, the recorded optimal width and depth, the recorded optimal performance measure, the adaptation strides, and the current state of the ASU. Given that most of these values are below 256 and can be stored within a single byte, we estimate that the ASU requires 20 bytes of storage.

Assuming the WPU has two warps with a SIMD width of 32 and it supports up to 8 warp-slices, each entry in the WST would require 14 bytes: 32 bits for the active mask, 1 bit for the warp ID, 64 bits for the PC, 3 bits for the warp-slice ID, 2 bits for the warp status, and 8 bits for storing the TOS if a re-convergence stack is used. A maximum of 8 entries are needed, resulting in a total of 112 bytes. For SIMT or array organizations, the duplicated re-convergence stack would add an additional column of PCs for each warp-slice. Assuming the stack can grow as high as 8 layers and each warp can be divided to at most eight warp-slices, this results in additional 1024 bytes. Adding the storage requirements for the WST and the ASU, this yields a total of 1164 bytes, or approximately 1 KB of storage.

To estimate area overhead, we measure realistic sizes for different units of a core according to a publicly available die photo of the AMD Opteron processor in 130nm technology. We scale the functional unit areas to 65nm, assuming a 0.7 scaling factor per generation. We assume each SIMD lane has a 32 bit data path (adjacent lanes are combined if 64 bit results are needed). We also measure the cache area per 1 KB of capacity and scale that according to the cache capacity. Assuming the WPU described above has a 32 KB D-cache and a 16 KB I-cache, the additional storage consumes less than 1% of a WPU's area.

Finally, since warps can be divided into independent scheduling entities, the hardware scheduler has to accommodate more entries and its priority encoder may be further complicated. Nevertheless, it is unlikely that a wide warp needs to be partitioned into more than four warp-slices, and such deep slicing only takes place when a wide WPU is originated

with one or two warps. This approximately increases the cost of the scheduling structure by a factor of four. Overall, our experiments show that wide, adaptive WPUs attempt no more than 16 warps or warp-slices.

7.6 Evaluation

We evaluate Robust SIMD (denoted as *RobustSIMD*) by demonstrating its performance on various SIMD applications and D-cache settings. Its results are compared to conventional SIMD organizations (denoted as *Conv*) as well as SIMD organizations with dynamic warp subdivision (denoted as *DWS*). In Section 7.6.2, we also propose a technique denoted as *RobustDWS* which combines Robust SIMD with DWS so that they compensate for each other. In this section, a SIMD organization with a width of W and a depth of D is denoted as ($W \times D$). Averaged performance is calculated using harmonic mean. Performance measurements include both adaptation phase and execution phase. The adaptation phase accounts for about 6% of the total execution time on average, and it never exceeds 15% of the total execution time. We envision that the overhead in adaptation will become even smaller with larger input sizes, because time spent in adaptation is independent of the total number of data-parallel tasks.

7.6.1 Robustness Across Various Benchmarks

We demonstrate performance robustness by testing *RobustSIMD* with various applications and D-cache settings and compare it to *Conv*. Starting with WPUs with private 8-way associative, 32 KB D-caches, we conduct space exploration of various SIMD widths and depths at powers of two. Results shown are limited to a maximum of 64 hardware thread contexts per WPU. Our experiments show that more threads per WPU will merely degrade

1	Benchmarks	Harmonic Mean	FFT	Filter	HotSpot	LU	Merge	Short	KMeans	SVM
2	***32KB, 8-way assoc. D-caches***									
3	Best *Conv* (8×4)	3.47	3.23	9.47	6.22	4.03	1.26	4.28	4.56	4.30
4	App-specific best width×depth		8×4	32×1	8×4	8×4	4×2	16×1	8×2	2×8
5	Optimal speedup	4.15	3.23	20.18	6.22	4.03	1.76	4.96	5.20	5.02
6	Best *RobustSIMD* (32×1)	4.07	3.36	20.32	6.31	3.18	1.79	4.88	7.62	4.00
7	Best *DWS* (16×2)	4.14	3.51	11.68	5.49	3.80	1.95	3.97	6.92	4.79
8	Best *RobustDWS* (16×2)	4.27	3.33	16.62	6.16	3.66	1.96	4.96	6.90	4.59
9	***Reduce D-caches to 16KB, 8-way***									
10	*Conv* (8×4)	1.42	2.00	4.26	1.26	0.74	1.02	0.98	2.59	2.73
11	App-specific best width×depth		4×4	16×1	8×1	8×1	4×2	8×1	8×1	2×8
12	Optimal speedup	3.39	2.80	10.59	4.04	2.81	1.71	3.94	4.32	4.24
13	*RobustSIMD* (32×1)	3.00	2.83	10.81	3.12	2.40	1.53	3.88	5.88	2.51
14	*DWS* (16×2)	1.83	2.28	4.31	1.49	0.96	1.82	1.01	5.54	3.77
15	*RobustDWS* (16×2)	3.36	2.85	10.77	3.76	2.42	1.85	4.00	5.56	3.49
16	***Reduce D-caches to 16KB, 4-way***									
17	*Conv* (8×4)	1.24	2.03	2.75	1.11	0.64	0.85	0.88	2.14	2.70
18	App-specific best width×depth		4×4	8×1	4×2	4×2	2×4	8×1	4×2	2×8
19	Optimal speedup	2.92	2.94	6.99	3.14	2.17	1.57	3.29	3.51	4.01
20	*RobustSIMD* (32×1)	2.63	2.84	9.07	3.09	2.03	1.19	3.19	4.82	2.44
21	*DWS* (16×2)	1.62	2.23	2.76	1.29	0.84	1.70	0.91	4.90	3.66
22	*RobustDWS* (16×2)	2.98	2.93	9.83	2.71	2.03	1.72	3.14	4.91	3.66

Table 7.3: Performance comparison between *Conv*, *RobustSIMD*, *DWS*, and *RobustDWS* across various benchmarks and D-cache settings.

performance for all benchmarks due to cache thrashing.

First, we find that among all *Conv* organizations, the organization with four 8-wide warps yields the best averaged performance across all benchmarks (using harmonic mean). However, for most benchmarks, this SIMD organization is not the best preferred (Row 4 in Table 7.3). Comparing Row 3 to Row 5 in Table 7.3, we show that the best *Conv* organization loses 16% performance compared to the optimal case where each benchmark runs on its optimal SIMD organization.

We then identify that with *RobustSIMD*, WPUs originated from a single 32-wide warp perform best (Row 6 in Table 7.3). Its overall performance is almost identical to the optimal case where each benchmark is hosted by their preferred *Conv* SIMD organization. Using *RobustSIMD*, WPUs with a few wide warps are likely to perform well uniformly across all benchmarks. Such organization maximizes throughput when there is ample data parallelism, little divergence behavior, and few long latency memory accesses. When such

conditions are not met, *RobustSIMD* transform wide WPUs to those with deeper multi-threading or fewer active thread contexts. As Figure 7.8 shows, instead of performance degradation beyond a certain point, *RobustSIMD* often yields similar or even better performance than *Conv* with wider warps.

By comparing Row 5 and Row 6 in Table 7.3, we found several benchmarks where the best *RobustSIMD* organization performs better than the application-specific best *Conv* organizations. The reasons are twofold. First, *RobustSIMD* is able to converge to *nonstandard* organizations where SIMD widths and depths are no longer a power of two. Second, *RobustSIMD* can adapt to phase changes where applications may prefer different SIMD organization at different periods of time.

RobustSIMD also reduces design risks. As can be observed from Figure 7.8, near-optimal performance can be achieved by a wider selection of SIMD widths and depths. In contract, *Conv* is more susceptible to performance degradation when the SIMD width and depth are not optimal.

Nevertheless, adaptation may subject to noise and *RobustSIMD* may converge to sub-optimal organizations. This is more likely to occur when the starting SIMD organization is unbalanced (i.e. with too many active thread contexts) or when the application demonstrates relatively unpredictable behavior. Despite this fact, the suboptimal performance of *RobustSIMD* in such cases still outperforms equivalent *Conv* organizations, as can be observed in Figure 7.8 when SIMD width reaches 64.

7.6.2 Robust SIMD Combined with Dynamic Warp Subdivision

Dynamic warp subdivision (DWS) addresses branch and memory divergences: upon selected divergent branches, a warp is split into two warp-splits, each taking a different control path; upon divergent cache accesses, a warp is split into two warp-splits as well, one

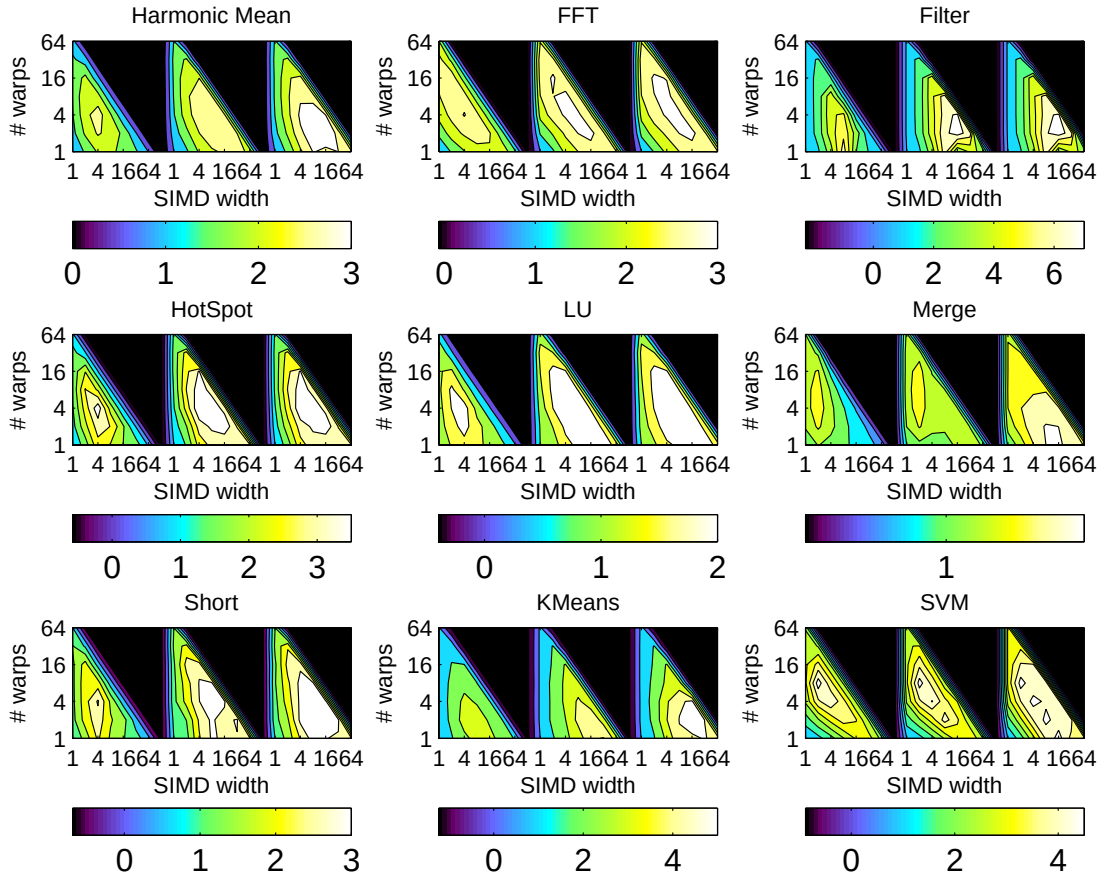


Figure 7.8: Comparing *Conv* (left), *RobustSIMD* (middle), and *RobustDWS* (right) over various SIMD organizations. Speedup is normalized to the organization with four single-threaded, in-order cores. D-caches are 8-way associative and are sized 32 KB. We experiment with SIMD widths and depths at powers of two, with up to 64 thread contexts per WPU.

with threads that hit the cache, the other with threads that miss the cache. Similar to Robust SIMD, DWS can trade SIMD width for multi-threading depth as well. However, this also means DWS risks pipeline under-utilization resulting from reduced SIMD width. DWS assumes that the benefit always outweighs the harm and therefore it splits warps whenever predefined conditions are met. Robust SIMD, on the other hand, evaluates the performance

of a particular organization prior to settling down on it; if the experimental organization performs badly, it is able to restore the previously attempted SIMD width and depth. Moreover, Robust SIMD provides the option to deactivate thread contexts in the case of cache contention. By comparing Row 6 and Row 7 in Table 7.3, we see that overall, the best *DWS* organization performs similarly to the best *RobustSIMD* organization. However, the same *DWS* organization does not perform consistently well when D-caches are resized for power saving, a case which we study in detail in Section 7.6.3.

We also observe that there are several benchmarks that usually prefer *DWS*. For example, LU and Merge spend significant time in parallel `for` loops which have only a few tasks. In these cases, *RobustSIMD* is not able to sufficiently sample and explore the configuration space. SVM has fast-changing phases exhibiting different divergence behavior and memory intensities. It is therefore difficult for Robust SIMD to adapt and converge to a stable organization. We therefore propose *RobustDWS* that chooses between Robust SIMD and DWS judiciously. For each data-parallel `for` loop, Robust SIMD is used at first to converge to a preferred organization. At the end of the adaptation phase, DWS is turned on and a *DWS phase* is inserted prior to the execution phase. In DWS phase, benchmarks execute with full SIMD width and depth as if they are not adjusted yet. At the end of this phase, performance is measured and compared against that which resulted from Robust SIMD. The WPU then picks the one that performs the best.

Integrating Robust SIMD and DWS is straightforward in hardware. The warp-slice table in Robust SIMD is structurally the same as the warp-split table in DWS. Therefore, the same warp-slice table can be used as the warp-split table as well. The re-convergence stacks for warp-slices remain the same since DWS does not modify the re-convergence stack. The only additional logic is a register that records which threads hit the cache and which missed during a SIMD cache access. The value of this register will be used to split a warp-slice into two; the WPU simply modifies the active mask of an existing warp-slice by

taking the "and" of the current active mask and the hit mask, and adds another entry into the warp-slice table by taking the "xor" of the two masks.

Our experiments show that the best *RobustDWS* organization has two 16-wide warps for each WPU, and it performs best overall. It improves the performance of *RobustSIMD* by another 5% with 32 KB D-caches and around 12% with 16 KB D-caches. We have also investigated alternative ways of combining Robust SIMD with DWS. For example, DWS can be kept on while Robust SIMD is adapting width and depth. As another option, we can use DWS but adjusting the number of active warp-splits. Neither of them provide robust performance due to interference among the two.

7.6.3 Robustness Across Various D-Cache Settings

We further evaluate the same SIMD organizations when the memory system reduces its capacity for power saving or error resilience purposes. With smaller cache capacity, the preferred active thread count per WPU tends to decrease because of contention. In addition, memory latency divergence increases with smaller caches to some extent. As a result, the preferred SIMD width is likely to decrease. The preferred multi-threading depth may increase or decrease, depending on the tradeoff between latency hiding and cache contention. Such trends can be observed from Table 7.3 by comparing Row 4 to Row 11 and Row 18.

We experimented with two types of cache organizations. Row 9 to Row 15 in Table 7.3 show the scenario where cache capacity is reduced in half by varying the number of cache sets; Row 16 to Row 22 in Table 7.3 show the case where the cache capacity is reduced in half by varying the set-associativity (*i.e.* turning off some ways). In these cases, the application-specific optimal speedups reduce by 18% and 30%, respectively. However, the *Conv* organization that works best with larger caches drops its performance by around 60%

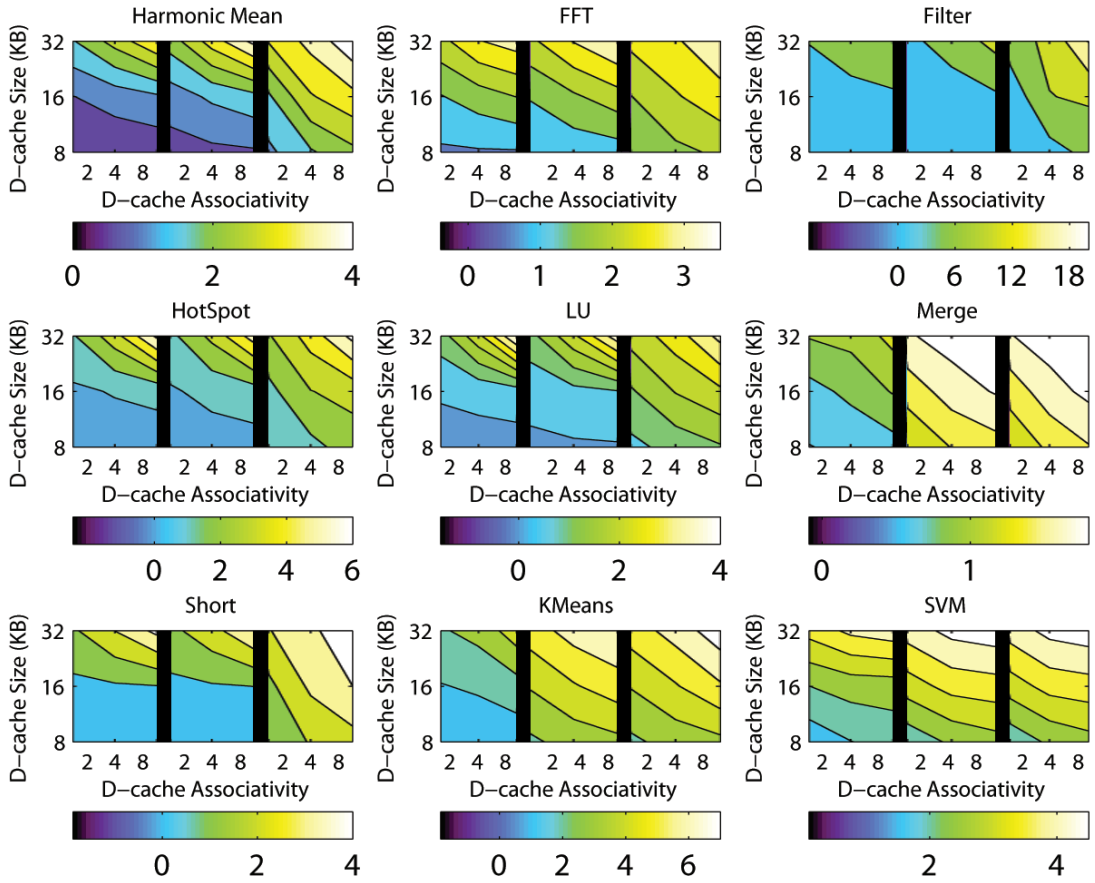


Figure 7.9: Comparing the best *Conv* organization with four 8-wide warps (left), the best *DWS* organization with two 16-wide warps (middle), and the best *RobustDWS* organization with two 16-wide warps (right) over various D-cache sizes and associativities. Speedup is normalized to the organization with four single-threaded, in-order cores with 32 KB, 8-way associative D-caches.

in both cases. The *DWS* organization drops its performance by around 60% as well. In other words, the *Conv* and *DWS* organizations that have near-optimal performance with 32 KB D-caches can only generate half of the optimal performance in the case of 16 KB D-caches. On the other hand, *RobustSIMD* yields 98% of the optimal performance with 32 KB D-caches and around 90% of the optimal performance with 16 KB D-caches. Finally,

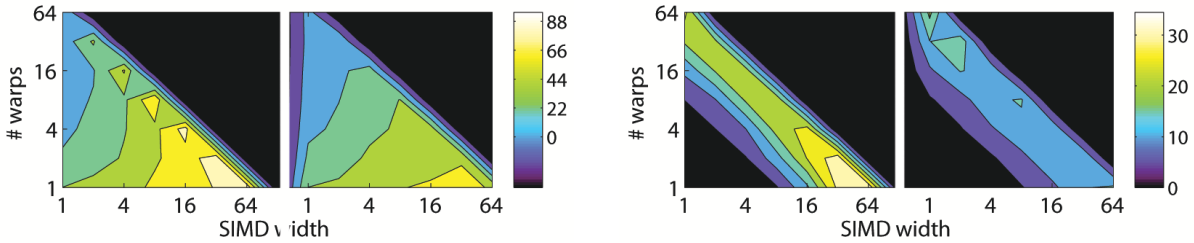
RobustDWS achieves 99% of the optimal performance in all cases.

Results of sensitivity studies show that both *Conv* and *DWS* are much more sensitive to D-cache capacity than *RobustSIMD* and *RobustDWS* (Figure 7.9). Specifically, compared to the best *Conv* organization with 32 KB D-caches (Row 3 in Table 7.3), the best *RobustDWS* organization is able to achieve 97% of its performance with only half of the D-cache capacity (Row 15 in Table 7.3)! This leads to significant power savings with minimal performance loss.

7.6.4 Effectiveness in Latency Hiding and Contention Reduction

The benefits of Robust SIMD result from better latency hiding and less cache contention. As Figure 7.10a illustrates, the percentage of time that WPUs stall waiting for memory accesses increases with higher thread count or fewer warps. It also increases with wider warps because wider warps are more prone to memory latency divergence. Therefore, performance conventionally increases first with wider warps but eventually starts to degrade, as can be observed from Figure 7.8. By transforming wide warps into multiple narrower warp-slices, *RobustSIMD* reduces the possibility for threads to suspend unnecessarily due to peer threads in the same warp missing the D-cache or branching into a different path. Not being blocked, these threads can continue to execute and issue further memory requests; this may increase memory level parallelism as well. Robust SIMD may also deactivate threads to reduce contention. As a result, Robust SIMD significantly reduces the time that WPUs stall waiting for memory (Figure 7.10a).

As Figure 7.10b shows, D-cache contention increases along with the number of threads per WPU, regardless of SIMD widths or multi-threading depths. Although deeper multi-threading is able to hide more latency, it risks cache thrashing which may prolong the average memory access time to such an extent that the demand of latency hiding increases



(a) Percentage of time that WPUs stall waiting for memory accesses.

(b) D-cache miss rate (%)

Figure 7.10: The impact of Robust SIMD on latency hiding and D-cache misses. In each figure, measurements for *Conv* are shown on the left and those for *RobustSIMD* are shown on the right.

beyond the capability of the multi-threading depth. This explains the phenomenon in all benchmarks where performance first increases with deeper multi-threading and then decreases, as shown in Figure 7.8. Figure 7.10b demonstrates that by turning off SIMD lanes or suspending active warps, Robust SIMD significantly reduces D-cache contention whenever there are too many active threads.

7.7 Conclusions and Future Work

This chapter proposes Robust SIMD, an adaptive technique that allows SIMD organizations to dynamically adjust SIMD width and multi-threading depth according to runtime performance feedback. The key emphasis is robustness in SIMD performance under various software and hardware settings. This enables diverse applications to converge to their preferred SIMD organization and also allows caches to reduce their capacities to save power without sacrificing performance significantly. Robust SIMD also reduces design complexity in exploring the space of SIMD width and depth. Compared to the performance generated by running every benchmark on its individually favored SIMD organization, the *same* Robust SIMD organization performs similarly, sometimes even better due to phase adaptation, and

outperforms the best fixed SIMD organization by 17%. Our experiments also show that even with dynamic warp subdivision, a fixed SIMD organization that performs best with 32 KB D-caches can only yield less than 50% of the optimal performance when D-cache capacities are reduced to 16 KB. By combining Robust SIMD and DWS, the proposed organization achieves 99% of the optimal performance persistently. In fact, it achieves 97% of the performance achievable by a conventional organization with D-caches that double in size.

To further reduce adaptation time and to enable Robust SIMD to adapt to data-parallel `for` loops with only a few tasks, we can employ persistent adaptation across multiple executions so that the preferred organization for an application can be generated using multiple executions and stored afterwards in a file. The runtime system can then load such information and use the learned SIMD organization directly for future executions. Moreover, we assume that the best SIMD configuration stays the same during each parallel execution phase because data-parallel threads are mostly homogeneous. However, such assumption may not be true if threads perform irregular data accesses, or if the underlying memory system reconfigures itself right after the adaptation phase; therefore, it will be helpful to restart the adaptation phase in such circumstances.

Chapter 8

Conclusions and Future Work

8.1 Thesis Summary

This dissertation has shown the recent trends of throughput oriented architecture design where a single chip often accommodates multiple cores and each core has multiple threads. The dissertation then describes several challenges posed by highly multi-threaded cores. Solutions are proposed for each identified challenge.

Extensive inter-thread data sharing Threads on the same core contend for the same D-cache. Previous techniques mainly focus on spatial locality within an individual thread. In Chapter 3, we propose a fine-grained task scheduling mechanism, symbiotic affinity scheduling (SAS), which leverages temporal locality among concurrent threads on the same core. Neighboring tasks are more likely to access adjacent data, therefore they are assigned to concurrent threads on the same core. The scheduler demonstrates an average speedup of 1.69× and average energy savings of 33% on the data-parallel benchmarks we studied. Based on conventional cache-blocking techniques, the exploited temporal locality brings

additional 30% performance gains. It also outperforms adaptive contention reduction techniques by 17%. This work was published in [65].

Magnified nonuniform data distribution In Chapter 4, we show that nonuniform data distribution caused by stack base allocation in the middleware is magnified by the large thread count, which leads to severe conflict misses in the shared cache. We then propose to randomize the stack base, which provides a mean speedup of 2.7X for 32 cores according to our simulations. To further eliminate page fragmentation, we designed a non-inclusive, semi-coherent cache (NISC) organization that excludes private data from cache coherence. While cache replacement policies fail to improve performance significantly, NISC alone scales performance up to at least 32 cores in most cases. Comparing to the best baseline configuration at 8 cores, NISC provides a mean speedup of 1.5X at 16 cores and 2.0X at 32 cores. This work was published in [64].

Abundant computation resources help reduce communication To reduce data communication, computation can be replicated to produce the same data on multiple consumer nodes. However, the right amount of computation to replicate is not obvious; it depends on the properties of applications and architectures. In Chapter 5, we establish an analytical performance model that predicts the optimal amount of computation to replicate base on NVIDIA's Tesla [13] architecture. The configuration suggested by the performance model achieves a speedup no less than 95% of the optimal speedup for our benchmarks. This work was published in [67].

SIMD constraints lead to unnecessary pipeline stalls SIMD threads have to wait until all threads in the same SIMD group finish their memory references before they can proceed to the next instruction. Moreover, when SIMD threads diverge to different branch paths, the resulting branch paths have to be executed one after another. Chapter 6 proposes dynamic warp subdivision (DWS) that allows threads that hit the cache to run ahead to issue more memory references and leverage memory level parallelism. DWS also allows

SIMD threads that fall into different branches to interleave their execution for better latency hiding. This is achieved by judiciously subdividing a warp into narrower warp-splits, each is simply an extra schedulable entity. Our experiments show that DWS generates an average speedup of 1.7X. This work was published in [68].

The need for adaptive architectures that offer robust performance The optimal SIMD width and multi-threading depth varies for different applications and memory systems, and such sensitivity motivates the design of adaptive SIMD organizations that offer robust performance. In Chapter 7, we propose Robust SIMD, an adaptive technique that allows SIMD organizations to dynamically adjust SIMD width and multi-threading depth according to runtime performance feedback. The key emphasis is robustness in SIMD performance under various software and hardware settings. This enables diverse applications to converge to their preferred SIMD organization. It also allows caches to reduce their capacities to save power without sacrificing performance significantly. In addition, Robust SIMD reduces design complexity in exploring the space of SIMD width and depth. Compared to the performance generated by running every benchmark on its individually favored SIMD organization, the *same* Robust SIMD organization performs similarly, sometimes even better due to phase adaptation, and outperforms the best fixed SIMD organization by 17%. By combining Robust SIMD and DWS, the proposed organization achieves 99% of the optimal performance persistently. This work was submitted to MICRO-43 [69].

8.2 Lessons Learned

We found that inefficiency in data movement often comes from the software or the middleware. Although hardware can be modified to tolerate such inefficiency, we advocate to co-design the middleware and hardware so that hazards in data movement can be removed at the first place. Such is the case with stack randomization in Chapter 4. Co-designing

middleware and hardware would require new tools that help architects detect potential middleware hazards such as non-uniform data allocation.

Moreover, various optimization techniques require both application-specific and hardware-specific parameters to identify the optimal configuration. As we demonstrate in Chapter 5, future systems can benefit from an integrated parallel framework that combines programmer hint, run-time profiling, and performance models to identify the best configuration for a particular application.

Finally, the CMT design space is huge; typical hardware parameters include number of threads per core, SIMD width (when applicable), degree of multi-threading, cache capacity and associativity, *etc*. Furthermore, applications respond differently to different architecture setups. As a result, there is a need to ensure performance robustness where the architecture allows the same application to perform well under various circumstances. Besides adapting SIMD width and multi-threading depth, as described in Chapter 7, we can also adaptive the underlying memory system altogether.

8.3 Future Directions

While this dissertation has addressed several challenges in data management for multi-threaded cores, our solutions are mostly focused on restructuring the computation and reconfiguring the multi-threaded pipeline. In our study, we also realize that various components in the memory system have to be modified to support multi-threaded cores. Moreover, little has been studied about latency hiding in SIMD architectures, especially when they operate over cache hierarchies. We propose several future directions to explore.

8.3.1 Memory System Organization for CMT

Multi-threaded cores may issue several memory requests in one cycle. This may affect both individual memory components and the overall memory organization, as described below.

First, D-caches need higher associativity to compensate for the increased contention caused by more threads per core. However, highly associative caches lead to longer latency and more power consumption. As a result, it may help maintain a low hit latency by adding an additional L0 cache which hosts frequently reused data.

Second, the TLB has to be banked or multi-ported to support multiple lookups in each cycle. Coalescing can be used to combine TLB lookups that fall into the same page. The TLB can also speculate which entries will be referenced in the near future to shorten the lookup latency. Moreover, TLB misses and page faults can be especially expensive for multi-threaded cores. This is because the thread that caused the page fault may operate in lockstep with other threads. It will help improve performance by allowing other threads to run ahead despite of the page fault.

Architectures can also leverage high level knowledge from programmers for efficient data management. Such information can guide data prefetching. It can also help decide the appropriate replacement policy for a data block; for example, *streaming data* that is consumed only once does not have to be cached at all.

Locality can be significantly improved if data is reorganized according to its access patterns. For example, organizing a multi-dimensional data in the pattern of a z-space filling curve usually yields better locality for neighborhood gathering operations. The only additional unit is a simple hardware unit that translates address in the z-space filling curve to the conventional address. Such reorganization can take place in both cache hierarchies and physical memory.

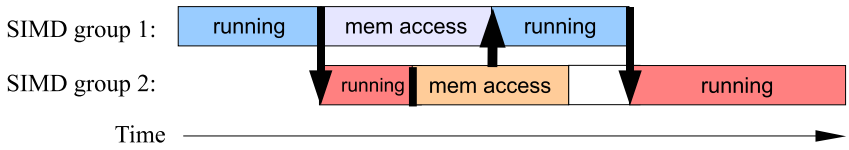
Finally, due to the high degree of contention in CMT's L1 storage, it would be interesting to explore an inverted cache hierarchy where the L1 caches dominate the on-chip storage, and the L2 only stores data that is very likely to be reused across cores. For instance, iterative stencil loops, such as those described in Chapter 5, only need to move data in the halo region across cores, therefore only halo data needs to be stored in the shared cache.

8.3.2 Better Latency Hiding for SIMD Architecture

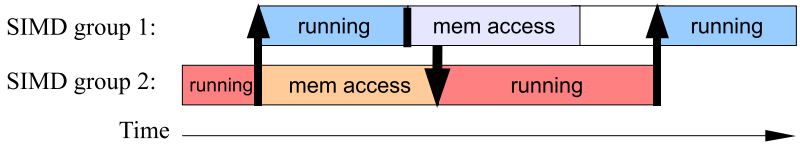
Different from conventional single-threaded processors, SIMD architectures embed simple, in-order cores and it uses multi-threading to hide latency. The effectiveness of latency hiding has not been well studied yet and this raises several interesting topics.

Warp scheduling policy may play an important role in latency hiding. Conventional round-robin (RR) scheduling prioritizes all SIMD groups equally, and is therefore unable to identify warps that can issue their long latency memory accesses in the near future. As Figure 8.1 illustrates, this may also cause longer pipeline stalls because fewer instructions are used to hide memory latency. It may improve latency hiding by speculating which warp is likely to miss the cache in the near future and giving it more priority.

An additional lightweight out-of-order(OOO) structure [173] may exploit more ILP for warp instructions. With more ILP, subsequent memory instructions may be issued earlier, improving MLP as well. For example, the GT200 architecture introduces scoreboard to allow non-blocking memory accesses that enable subsequent independent instructions to be issued and committed in order. A more aggressive, but lightweight OOO structure may exploit more ILP, bypassing those instructions that depend on the memory accesses. According to the degree of additional ILP that OOO structures exploit, SIMD cores may need fewer warps to hide the same latency caused by structural hazards or memory accesses,



(a) Execution trace if the scheduler is round-robin



(b) More latency can be hidden if the scheduler gives higher priority to SIMD groups that incur cache misses shortly.

Figure 8.1: By giving higher scheduling priority to warps are likely to miss sooner, more MLP can be exploited than with a round-robin scheduler.

reducing the area overhead. To further save area, the number of OOO structures can be reduced if the generated out-of-order instruction sequences can be reused by multiple homogeneous warps warps that execute the same instructions within a given period of time one after another.

Synchronously executing homogeneous warps may further reduce area and save energy. Several benchmarks will be studied to characterize the frequency of branch divergence that yield warps with heterogeneous instruction sequences. Given that warps are likely to be homogeneous, they can be synchronized so that energy can be saved by reducing the number of fetch, decode, and scoreboard operations. Synchronous warps may reduce the area overhead of the OOO structures described above: instead of one OOO structure per warp, multiple warps can share the same OOO structure.

Bibliography

[1] P. Dubey, "A Platform 2015 Workload Model: Recognition, Mining and Synthesis Moves Computers to the Era of Tera," *White Paper, Intel Corporation*, vol. 96, 2008.

[2] N. Nagarajayya, "Improving Application Efficiency Through Chip Multi-Threading," Mar. 2005.

[3] Jack L. Lo et al., "Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading," *ACM Trans. on Computer Systems*, vol. 15, no. 3, 1997.

[4] S. Moy and E. Lindholm, "Method and system for programmable pipelined graphics processing with branching instructions," *US Patent 6,947,047*, 2005.

[5] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium 4 Processor," *Intel Technology Journal*, 2001.

[6] R. Kalla, S. Balaram, and J.M. Tendler, "IBM Power5 chip: a dual-core multi-threaded processor," *Micro, IEEE*, vol. 24, no. 2, Mar-Apr 2004.

[7] Q. Jacobson, "UltraSPARC IV Processors," 2003.

[8] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro*, vol. 25, no. 2, 2005.

[9] M. Gschwind, "Chip multiprocessing and the Cell Broadband Engine," in *CF'06*, 2006.

[10] Y. Nishikawa, M. Koibuchi, M. Yoshimi, K. Miura, and H. Amano, "Performance Improvement Methodology for ClearSpeed's CSX600," in *ICPP*, 2007.

[11] R. M. Russell, "The CRAY-1 computer system," *Commun. ACM*, vol. 21, no. 1, 1978.

[12] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hern, T. Juan, G. Lowney, M Mattina, and A. Seznec, "Tarantula: A Vector Extension to the Alpha Architecture," in *ISCA*, 2002.

[13] NVIDIA Corporation, "GeForce GTX 280 Specifications," 2008.

[14] "NVIDIAs Next Generation CUDA Compute Architecture: Fermi," *NVIDIA Corporation*, 2009.

[15] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, no. 3, 2008.

[16] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. López-Lagunas, P. R. Mattson, and J. D. Owens, "A bandwidth-efficient architecture for media processing," in *MICRO 31*, 1998.

[17] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi, "Merrimac: Supercomputing with Streams," in *SC*, 2003.

[18] NVIDIA Corporation, "GeForce GTX 280 Specifications," 2008.

[19] S. A. McKee, "Reflections on the memory wall," in *CF*, 2004.

[20] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A Performance Study of General Purpose Applications on Graphisc Processors using CUDA," *JPDC*, 2008.

[21] L. Carter, J. Ferrante, and S. F. Hummel, "Hierarchical tiling for improved superscalar performance," in *IPPS*, 1995.

[22] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *SC*, 2008.

[23] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the Memory Hierarchy," in *SC*, 2006.

[24] M. Frigo and V. Strumpen, "Cache oblivious stencil computations," in *ICS*, 2005.

[25] W. Jalby and U. Meier, "Optimizing Matrix Operations on a Parallel Multiprocessor with a Hierarchical Memory System," 1986.

[26] I. Kodukula, N. Ahmed, and K. Pingali, "Data-centric multi-level blocking," in *PLDI*, 1997.

[27] Milind Kulkarni, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew, "Optimistic parallelism benefits from data partitioning," in *ASPLOS 13*, 2008.

[28] K. S. McKinley and O. Temam, "Quantifying loop nest locality using SPEC'95 and the perfect benchmarks," *ACM Trans. Comput. Syst.*, vol. 17, no. 4, 1999.

[29] V. K. Pingali, S. A. McKee, W. C. Hseih, and J. B. Carter, "Computation regrouping: restructuring programs for temporal data cache locality," in *ICS*, 2002.

[30] J. Ramanujam, "Tiling of iteration spaces for multicomputers," in *Proc. 1990 Int. Conf. Parallel Processing, Vol*, 1990.

[31] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li, "Thread scheduling for cache locality," in *ASPLOS-VII*. 1996, ACM.

[32] J. Torrellas, A. Tucker, and A. Gupta, "Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors," *J. Parallel Distrib. Comput.*, vol. 24, no. 2, 1995.

[33] M. Chaudhuri, "PageNUCA: Selected policies for page-grain locality management in large shared chip-multiprocessor caches," in *HPCA*, Feb. 2009.

[34] S. Jenks and J.-L. Gaudiot, "An evaluation of thread migration for exploiting distributed array locality," in *HPCA*, 2002.

[35] E. P. Markatos and T. J. LeBlanc, "Using processor affinity in loop scheduling on shared-memory multiprocessors," in *SC*, 1992.

[36] G. E. Blelloch, P. B. Gibbons, and Y. Matias, "Provably efficient scheduling for languages with fine-grained parallelism," in *ACM Proc. of Annu. Symp. on Para. Alg. and Archi.*, 1995.

[37] R. D. Blumofe, *Executing multithreaded programs efficiently*, Ph.D. thesis, 1995.

[38] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson, "Scheduling threads for constructive cache sharing on CMPs," in *SPAA*, 2007.

[39] S. Subramaniam and D. L. Eager, "Affinity scheduling of unbalanced workloads," in *SC*, 1994.

[40] S. Parekh, S. Eggers, and H. Levy, "Thread-sensitive scheduling for smt processors," Tech. Rep., 2000.

[41] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *ASPLOS*, 2000.

[42] G. E. Suh, S. Devadas, and L. Rudolph, "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning," in *HPCA*. 2002, IEEE Computer Society.

[43] Y. Zhang, M. Burcea, V. Cheng, R. Ho, and M. Voss, "An Adaptive OpenMP Loop Scheduler for Hyperthreaded SMPs," in *PDCS*, 2004.

[44] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture," in *HPCA*, 2005.

[45] A. Settle, D. Connors, E. Gibert, and A. González, "A dynamically reconfigurable cache for multithreaded processors," *J. Embedded Comput.*, vol. 2, no. 2, 2006.

[46] S. Wang and L. Wang, "Thread-associative memory for multicore and multithreaded computing," in *ISLPED*, 2006.

[47] L. Hsu, R. Iyer, S. Makineni, S. Reinhardt, and D. Newell, "Exploring the cache design space for large scale CMPs," *dasCMP*, vol. 33, no. 4, 2005.

[48] L. Zhao, R. Iyer, M. Upton, and D. Newell, "Towards Hybrid Last Level Caches for Chip-Multiprocessors," *dasCMP*, 2008.

[49] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Optimizing Replication, Communication, and Capacity Allocation in CMPs," *ISCA*, vol. 33, no. 2, 2005.

[50] J. Chang and G. S. Sohi, "Cooperative Caching for Chip Multiprocessors," in *ISCA*. 2006, IEEE Computer Society.

[51] M. Zhang and Krste Asanovic, "Victim Migration: Dynamically Adapting between Private and Shared CMP Caches," in *MIT Technical Report MIT-CSAIL-TR-2005-064, MIT-LCS-TR-1006*, 2005.

[52] M. Zhang and Krste Asanovic, "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors," in *ISCA*, 2005.

[53] Y. Zheng, B. T. Davis, and M. Jordan, "Performance evaluation of exclusive cache hierarchies," in *ISPASS*. 2004, IEEE Computer Society.

[54] M. Zahran, K. Albayraktaroglu, and M. Franklin, "Non-Inclusion Property in Multi-level Caches Revisited," in *IJCA*, 2007.

[55] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A tagless coherence directory," in *MICRO 42*, 2009.

[56] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: Bulk Enforcement of Sequential Consistency," in *ISCA*, 2007.

[57] A. R. Alameldeen and D. A. Wood, "Adaptive Cache Compression for High-Performance Processors," in *ISCA*, 2004.

[58] M. K. Qureshi, M. A. Suleman, and Y. N. Patt, "Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines," in *HPCA*. 2007, IEEE Computer Society.

[59] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, "Temporal Streaming of Shared Memory," *ISCA*, 2005.

[60] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, C. Gniady, A. Ailamaki, and B. Falsafi, "Store-Ordered Streaming of Shared Memory," in *PACT*, 2005.

[61] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial Memory Streaming," *ISCA*, 2006.

[62] T. Ramirez, A. Pajuelo, O.J. Santana, and M. Valero, "Runahead Threads to improve SMT performance," *HPCA*, Feb. 2008.

[63] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen, "Dynamic speculative precomputation," in *MICRO 34*, 2001.

[64] J. Meng and K. Skadron, "Avoiding Cache Thrashing due to Private Data Placement in Last-level Cache For Manycore Scaling," in *ICCD*, Oct 2009.

[65] J. Meng, J. W. Sheaffer, and K. Skadron, "Exploiting Inter-thread Temporal Locality for Chip Multithreading," in *IPDPS*, 2010.

[66] Z. Li and Y. Song, "Automatic tiling of iterative stencil loops," *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 6, 2004.

[67] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs," in *ICS*, 2009.

[68] J. Meng, D. Tarjan, and K. Skadron, "Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance," in *ISCA*, 2010.

[69] J. Meng, J. W. Sheaffer, and K. Skadron, "Robust SIMD: Dynamically Adapted SIMD Width and Multi-Threading Depth," in *submitted to MICRO*, 2010.

[70] D. Luebke and G. Humphreys, "How GPUs work," 2007.

[71] T. Y. Yeh, P. Faloutsos, S. J. Patel, and G. Reinman, "ParallAX: an architecture for real-time physics," in *ISCA*, 2007.

[72] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, M. P. Eastwood, J. Gagliardo, J. P. Grossman, C. R. Ho, D. J. Ierardi, I. Kolossváry, J. L. Klepeis, T. Layman, C. McLeavey, M. A. Moraes, R. Mueller, E. C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, and S. C. Wang, "Anton, a special-purpose machine for molecular dynamics simulation," in *ISCA*, 2007.

[73] M. Bedford T., J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," *IEEE Micro*, vol. 22, no. 2, 2002.

[74] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, "Piranha: a scalable architecture based on single-chip multiprocessing," *ISCA*, 2000.

[75] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance," in *ISCA*, 2004.

[76] Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi, "Core architecture optimization for heterogeneous chip multiprocessors," in *PACT*, 2006.

[77] W. R. Mark, R. Steven, R. Steven Glanville, K. Akeley, and M. J. Kilgard, "Cg: A System for Programming Graphics Hardware in a C-like Language," *SIGGRAPH*, 2003.

[78] NVIDIA Corporation, "NVIDIA CUDA compute unified device architecture programming guide," 2007.

[79] AMD, "The future is fusion: The Industry-Changing Impact of Accelerated Computing," 2008.

[80] D. M. Tullsen, "Simulation and Modeling of a Simultaneous Multithreading Processor," *the 22nd Annual Computer Measurement Group Conference*, 1996.

[81] P. M. Ortego and P. Sack, "SESC: SuperESCalar Smulator," 2004.

[82] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *Computer*, vol. 35, no. 2, 2002.

[83] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, "SimFlex: a fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 4, 2004.

[84] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *IEEE Micro*, vol. 26, no. 4, 2006.

[85] D. C. Burger, T. M. Austin, and S. Bennett, "Evaluating Future Microprocessors: the SimpleScalar Tool Set," Tech. Report TR-1308, July 1996.

[86] M. Yourst, "PTLsim Users Guide and Reference: The Anatomy of an x86-64 Out of Order Microprocessor," Tech. Rep., SUNY Binghamton.

[87] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, 2005.

[88] N. Agarwal, L.-S. Peh, and N. Jha, "Garnet: A Detailed Interconnection Network Model inside a Full-system Simulation Framework," Tech. Rep. CE-P08-001, Princeton University, 2008.

[89] D. Tarjan, S. Thoziyoor, and N. P. Jouppi, "CACTI 4.0," Tech. Rep. HPL-2006-86, HP Laboratories Palo Alto, 2006.

[90] A. Pullini, F. Angiolini, S. Murali, D. Atienza, G. De Micheli, and L. Benini, "Bringing NoCs to 65 nm," *IEEE Micro*, vol. 27, no. 5, 2007.

[91] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," in *ISCA 27*, Jun 2000.

[92] I. Hur and C. Lin, "A Comprehensive Approach to DRAM Power Management," *HPCA*, 2008.

[93] Altair Coorporate, "PBS Professional," 2008.

[94] T. Williams and C. Kelley, "gnuplot: An Interactive Plotting Program," .

[95] J. D. Davis, J. Laudon, and K. Olukotun, "Maximizing CMP Throughput with Mediocre Cores," in *PACT*, 2005.

[96] Intel Corporation, "Picture the Future now: Intel AVX," .

[97] J. P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared Memory," *ISCA'95*, vol. 20, no. 1, Mar. 1992.

[98] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, "MineBench: A Benchmark Suite for Data Mining Workloads," *IISWC*, Oct. 2006.

[99] J. L. Lo, S. J. Eggers, H. M. Levy, S. S. Parekh, and D. M. Tullsen, "Tuning compiler optimizations for simultaneous multithreading," in *MICRO 30*, 1997.

[100] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: architectural support for fine-grained parallelism on chip multiprocessors," *ISCA*, vol. 35, no. 2, 2007.

[101] OpenMP Architecture Review Board, "OpenMP Application Program Interface," May 2008.

[102] L. Dagum, "OpenMP: A Proposed Industry Standard API for Shared Memory Programming," October 1997.

[103] Intel Corporation, "Intel Threading Building Blocks," .

[104] W. Huang, M. R. Stan, K. Skadron, S. Ghosh, K. Sankaranarayanan, and S. Velusamy, "Compact Thermal Modeling for Temperature-Aware Design," in *DAC*, 2004.

[105] K. Asanovic, R. Bodik, B. Christopher C., J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 18 2006.

[106] K. N. Premnath and J. Abraham, "Three-dimensional multi-relaxation time (MRT) lattice-Boltzmann models for multiphase flow," *J. Comput. Phys.*, vol. 224, no. 2, 2007.

[107] G. Allen, W. Benger, T. Dramlitsch, T. Goodale, H.-C. Hege, G. Lanfermann, A.é Merzky, T. Radke, and E. Seidel, "Cactus Grid Computing: Review of Current Development," in *Euro-Par '01*. 2001, Springer-Verlag.

[108] Inc. XILINX, "Virtex-II Pro and Virtex-II Pro X FPGA User Guide," .

[109] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart Memories: a modular reconfigurable architecture," in *ISCA*, 2000.

[110] S.-H. Yang, B. Falsafi, M. D. Powell, and T. N. Vijaykumar, "Exploiting Choice in Resizable Cache Design to Optimize Deep-Submicron Processor Energy-Delay," in *HPCA*. 2002, IEEE Computer Society.

[111] N. P. Jouppi and S. J. E. Wilton, "Tradeoffs in two-level on-chip caching," in *ISCA'94*, Apr. 1994.

[112] X. Chen, Y. Yang, G. Gopalakrishnan, and C.-T. Chou, "Reducing Verification Complexity of a Multicore Coherence Protocol Using Assume/Guarantee," in *FMCAD*, 2006.

[113] S. B. Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: An Operating System for Many Cores," in *OSDI '08*, December 2008.

[114] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos, "Scalable locality-conscious multithreaded memory allocation," in *ISMM*, 2006.

[115] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: a scalable memory allocator for multithreaded applications," *SIGPLAN Not.*, vol. 35, no. 11, 2000.

[116] H. Shacham, E. j. Goh, N. Modadugu, B. Pfaff, and D. Boneh, "On the effectiveness of address-space randomization," in *CCS*, 2004.

[117] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *ISCA*, 2007.

[118] H. S. Lee and G. S. Tyson, "Region-based caching: an energy-delay efficient memory architecture for embedded processors," in *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2000.

[119] S. Park, H. w. Park, and S. Ha, "A novel technique to use scratch-pad memory for stack management," in *DATE*, 2007.

[120] P. R. Panda, N. D. Dutt, and A. Nicolau, "On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems," *DAES*, vol. 5, no. 3, July 2000.

[121] C. McCurdy and C. Fischer, "A localizing directory coherence protocol," in *WMPI*, 2004.

[122] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," *SIGOPS*, vol. 36, no. 5, 2002.

[123] John Hennessy and David Patterson, *Computer Architecture - A Quantitative Approach*, Morgan Kaufmann, 2003.

[124] "Reference Guide: R700 Family Instruction Set Architecture," 2009.

[125] "NVIDIA compute PTX: Parallel Thread Execution," *NVIDIA Corporation*, 2007.

[126] M. Ripeanu, A. Iamnitchi, and I. Foster, "Cactus application: Performance predictions in a grid environment," in *EuroPar*, 2001.

[127] G. Allen, T. Dramlitsch, I. Foster, N. T. Karonis, M. Ripeanu, E. Seidel, and B. Toonen, "Supporting efficient execution in heterogeneous distributed computing environments with cactus and globus," in *SC*, 2001.

[128] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, no. 2, 2008.

[129] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick, "Impact of modern memory subsystems on cache optimizations for stencil computations," in *MSP*, 2005.

[130] M. Kowarschik, C. Weiβ, W. Karl, and U. Rüde, "Cache-aware multigrid methods for solving Poisson's equation in two dimensions," *Computing*, vol. 64, no. 4, 2000.

[131] G. Rivera and C.-W. Tseng, "Tiling optimizations for 3D scientific computations," in *SC*, 2000.

[132] M. Bromley, S. Heller, T. McNerney, and G. L. Steele, Jr., "Fortran at ten gigaflops: the connection machine convolution compiler," *PLDI*, vol. 26, no. 6, 1991.

[133] D. Wonnacott, "Time skewing for parallel computers," in *WLCPC*, 1999.

[134] S. J. Deitz, B. L. Chamberlain, and L. Snyder, "Eliminating redundancies in sum-of-product array computations," in *ICS*, 2001.

[135] L. Renganarayana, M. Harthikote-Matha, R. Dewri, and S. Rajopadhye, "Towards Optimal Multi-level Tiling for Stencil Computations," *IPDPS*, March 2007.

[136] L. Renganarayana and S. Rajopadhye, "Positivity, posynomials and tile size selection," in *SC*, 2008.

[137] N. Manjikian and T. S. Abdelrahman, "Fusion of loops for parallelism and locality," *Parallel Distrib. Syst.*, vol. 8, 1997.

[138] D. Wonnacott, "Achieving Scalable Locality with Time Skewing," *Int. J. Parallel Program.*, vol. 30, no. 3, 2002.

[139] L. Chen Z.-Q. Zhang X.-B. Feng, "Redundant Computation Partition on Distributed-Memory Systems," in *ICA3PP*, 2002.

[140] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," *PLDI*, vol. 42, no. 6, 2007.

[141] S. Chatterjee, J.R. Gilbert, and R. Schreiber, "Mobile and replicated alignment of arrays in data-parallel programs," *SC*, Nov. 1993.

[142] P. Lee, "Techniques for compiling programs on distributed memory multicomputers," *Parallel Comput.*, vol. 21, 1995.

[143] C.-H. Huang and P. Sadayappan, "Communication-free hyperplane partitioning of nested loops," *J. Parallel Distrib. Comput.*, vol. 19, no. 2, 1993.

[144] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A Performance Study of General Purpose Applications on Graphics Processors using CUDA," June 2008.

[145] N. Goodnight, "CUDA/OpenGL Fluid Simulation," Tech. Rep., April 2007.

[146] Z. Yang, Y. Zhu, and Y. Pu, "Parallel Image Processing Based on CUDA," *Int. Conf. Comput. Sci. and Software Eng.*, vol. 3, Dec. 2008.

[147] S.-Z. Ueng, S. Baghsorkhi, M. Lathara, and W. m. Hwu, "CUDA-lite: Reducing GPU Programming Complexity," in *LCPC*, 2008.

[148] NVIDIA Corporation, "NVIDIA CUDA Visual Profiler," June 2008.

[149] L. C. Evans, *Partial Differential Equations*, American Mathematical Society, 1998.

[150] M. Alpert, "Not just fun and games," *Scientific American*, April 1999.

[151] Electronic Educational Devices Inc., "Watts up? Electricity Meter Operator's Manual," 2002.

[152] C. Kozyrakis, "A Media-Enhanced Vector Architecture for Embedded Memory Systems," Tech. Rep., University of California, Berkeley, 1999.

[153] S. E. Orcutt, "Implementation of Permutation Functions in Illiac IV-Type Computers," *IEEE Trans. Comput.*, vol. 25, no. 9, 1976.

[154] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," in *MICRO 40*, 2007.

[155] U. J. Kapasi, J. Dally, W, S. Rixner, P. R. Mattson, J. D. Owens, and B. Khailany, "Efficient conditional operations for data-parallel architectures," in *MICRO 33*, 2000.

[156] R. A. Lorie and H. R. Strong, "Method for conditional branch execution in SIMD vector processors," *US Patent 4,435,758*, 1984.

[157] Y. Takahashi, "A Mechanism for SIMD Execution of SPMD Programs," in *HPC-ASIA*, 1997.

[158] D. Talla and L. K. John, "Cost-effective Hardware Acceleration of Multimedia Applications," in *ICCD*, 2001.

[159] D. Tarjan, J. Meng, and K. Skadron, "Increasing Memory Miss Tolerance for SIMD Cores," in *SC*, 2009.

[160] S. Rivoire, R. Schultz, T. Okuda, and C. Kozyrakis, "Vector Lane Threading," in *ICPP*, 2006.

[161] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The Vector-Thread Architecture," in *ISCA*, 2004.

[162] S. Choi and D. Yeung, "Learning-Based SMT Processor Resource Distribution via Hill-Climbing," in *ISCA*, 2006.

[163] D. M. Tullsen and J. A. Brown, "Handling long-latency loads in a simultaneous multithreading processor," in *MICRO 34*, 2001.

[164] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A Case for MLP-Aware Cache Replacement," *ISCA*, vol. 34, no. 2, 2006.

[165] ATI, "Radeon 9700 Pro," http://mirror.ati.com/products/pc/radeon9700pro, 2002.

[166] M. Erez, J. H. Ahn, J. Gummaraju, M. Rosenblum, and W. J. Dally, "Executing irregular scientific applications on stream architectures," in *ICS*, 2007.

[167] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser, "Many-Core vs. Many-Thread Machines: Stay Away From the Valley," *IEEE Comput. Archit. Lett.*, vol. 8, no. 1, 2009.

[168] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart Memories: a modular reconfigurable architecture," in *ISCA*, 2000.

[169] S.-H. Yang, B. Falsafi, M. D. Powell, and T. N. Vijaykumar, "Exploiting Choice in Resizable Cache Design to Optimize Deep-Submicron Processor Energy-Delay," in *HPCA*, 2002.

[170] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu, "Trading off Cache Capacity for Reliability to Enable Low Voltage Operation," in *ISCA*, 2008.

[171] Khronos Group Std., "The OpenCL Specification, Version 1.0," April 2009.

[172] R. A. Alfieri, "An efficient kernel-based implementation of POSIX threads," in *USTC'94*. 1994, USENIX Association.

[173] D. Tarjan, M. Boyer, and K. Skadron, "Federation: repurposing scalar cores for out-of-order instruction issue," in *DAC*, 2008.