Pipelined Multithreading Transformations and Support Mechanisms

Ram Rangan

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science

June 2007

© Copyright by Ram Rangan, 2007.

All Rights Reserved

Abstract

Even though chip multiprocessors have emerged as the predominant organization for future microprocessors, the multiple on-chip cores do not directly result in improved application performance (especially for legacy applications, which are predominantly sequential C/C++ codes). Consequently, parallelizing applications to execute on multiple cores is essential to their success. Independent multithreading techniques, like DOALL extraction, create partially or fully independent threads, which communicate rarely, if at all. While such strategies keep high inter-thread communication costs from impacting program performance, they cannot be applied to parallelize general-purpose applications which are characterized by difficult-to-break recurrences. Even though cyclic multithreading techniques, such as DOACROSS, are more applicable, the cyclic inter-thread dependences created by these techniques cause them to have very low tolerance to rising inter-core latencies.

To address these problems, this work introduces a pipelined multithreading (PMT) transformation called Decoupled Software Pipelining (DSWP). DSWP, in particular, and PMT techniques, in general, are able to tolerate inter-core latencies, while still handling codes with complex recurrences. They achieve this by enforcing an acyclic communication discipline amongst threads, which allow threads to use inter-thread queues to communicate in a pipelined fashion. This dissertation demonstrates that DSWPed codes not only tolerate inter-core communication costs, but also effectively tolerate variable latency stalls in applications better than single-threaded execution on both in-order and out-of-order issue processors with comparable resources. It then performs a thorough analysis of the performance scalability of automatically generated DSWPed codes and identifies the conditions necessary to achieve peak PMT performance.

Next, the dissertation shows that even though PMT applications tolerate inter-core latencies well, the high frequency of inter-thread communication (once every 5 to 20 dynamic instructions) in these codes, makes them very sensitive to the intra-thread overhead imposed by communication operations. In order to understand the issues surrounding inter-thread communication for PMT applications, this dissertation undertakes a methodical exploration of the design space of communication support options for PMT. Three new communication mechanisms with varying cost-performance tradeoffs are introduced and are shown to perform 38% to 200% better than the state of the art.

Acknowledgments

I owe this work to my advisor, David August. I thank him for believing in me. I thank him for supporting me every step of the way, from recommending my admission to Princeton to feeding me for a whole three and a half years (with liberal grants from NSF Grant No. 0133712, NSF Grant No. 0305617, and Intel Corporation)[1], from showing me how to use gdb during my first summer to helping improve my writing skills over the years, from celebrating every small research success I had to consoling and encouraging me every single time I was not able to make a paper deadline or had a paper rejected, and so on. His tonic was definitely bitter many a time, but in hindsight, there appears to have been a method to the madness and the many situations I sulked about originally, have ultimately worked out to my benefit. His take-the-bull-by-the-horns approach to research and hard work will always be an inspiration for me.

The central idea of this dissertation, decoupled software pipelining (DSWP), was born out of discussions with Neil Vachharajani, my primary collaborator, and David August. I would like to thank Margaret Martonosi, George Cai, Li Shiuan Peh, and Jaswinder Pal Singh for serving on my thesis committee. Their collective wisdom has made this dissertation more complete in terms of providing an in-depth understanding of various aspects of DSWP behavior. I thank my advisor and my readers, George Cai and Margaret Martonosi, for carefully reading through my dissertation and suggesting fixes. Special thanks to Margaret Martonosi for her impressive turnaround time that enabled the timely scheduling of my final defense. As collaborator, George Cai's expert inputs helped improve the quality of the DSWP communication support work.

I would like to thank HP Labs and Intel for providing me with valuable internship opportunities. I had great fun during these internships and learned a lot from working closely with Shail Aditya and Scott Mahlke (at HP) and Shubu Mukherjee, Arijit Biswas, Paul Racunas, and Joel Emer (at Intel).

[1]Opinions, findings, conclusions, and recommendations expressed in this dissertation are not necessarily the views of the NSF or Intel Corporation

I would probably not be in computer architecture if not for my undergraduate advisor, Ranjani Parthasarathi. Her classes and exams gave me a good understanding of the basics. By allowing her students complete freedom and by being available for discussions, she enabled the exploration of research-quality ideas for class projects. For all this and more, I am deeply indebted to her.

My heartfelt thanks to all past and current members of the Liberty group. I thank Manish Vachharajani, David Penry, and Neil Vachharajani for developing the Liberty Simulation Environment; it really made microarchitecture modeling a straightforward process and helped me reason about and truly understand the innards of various hardware blocks. The cache, in particular! Thanks to David Penry for developing a robust IA64 emulator and a cycle-accurate Itanium 2 core model. Many thanks to Spyridon Triantafyllis, Matthew Bridges, Easwaran Raman, and the entire VELOCITY compiler team for their hard work. Guilherme Ottoni and Neil Vachharajani implemented the automatic DSWP algorithm used in this dissertation in the VELOCITY compiler. I had great fun working with Jonathan Chang, George Reis, Adam Stoler, Jason Blome, and Bolei Guo on various projects. The random discussions, jokes, movies, literati, text twist, tennis, and squash games with my lab mates kept me sane through graduate school.

Outside of my lab, Lakshmi and Mani, Easwar, Logo, the Shivakumar and the Mahesh families, Julia Chen, Kevin Ko, Ge Wang, Margaret and Bob Slighton, Steve Wallitt, Neal Kaufmann, the entire PUTTC crowd, and several others cheered me up so much that I almost never missed home. Close relatives and friends from school and college, Alwyn, Anusha, Arun, Arundhati, Kasturi athai, Mani, Ravi, Robs, Satish, Siva, Subha, and others have always provided great encouragement and support.

I thank Nitya for her unconditional love and friendship. A big thank you to my little sister, Ramya, for her love and affection over the years. I thank my grandparents for their blessings and prayers. I salute their indefatigable spirit that made them take great interest in my progress towards thesis completion, despite their myriad age-related problems. I thank all four grandparents for loving me so much and for all that they have taught me. Finally, words cannot express my love and gratitude for my parents who have given their all for both their children. Their innumerable sacrifices, their selfless love, and their keen interest in my success, are what have gotten me to this point. I am indeed fortunate to have such a wonderful family and I wish to dedicate my life's work (well, at least, five and a half years of it) to my parents and my grandparents for everything they have given me and done for me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

For years, steady clock rate improvements and aggressive instruction level parallelism (ILP) techniques have sustained application performance improvements. However, this trend is not sustainable. On the one hand, aggressive clocking leads to diminishing power [61] and performance [23] returns, power delivery and heat dissipation problems. On the other hand, aggressive exploitation of ILP in single-context execution [12, 43, 56] uses disproportionately more hardware and consumes more power, since it involves excessive speculation and/or additional bookkeeping (for example, checkpointing of processor state) to implement large logical instruction windows. Further, design and verification costs for large uniprocessor designs also become prohibitively expensive.

These trends have led the microprocessor industry to move away from traditional designs to multi-core architectures or chip multiprocessors (CMPs). IBM Power5, Intel Core-Duo, AMD Dual-Core Opteron, Sun Niagara, and STI Cell are all examples of current generation CMPs. CMPs replace a single large monolithic core with multiple smaller processing cores on the same die area. Besides greatly simplifying design and verification tasks, such designs overcome the clock speed, power, thermal, and scalability problems plaguing aggressive uniprocessor designs while continuing to provide additional computing power using additional transistors provided by technology scaling. While additional

```
for(i=1; i<=N; i++) // C
  a[i] = a[i] + 1;  // X
```

(a) Loop with no loop-carried dependences (DOALL)

```
while(ptr = ptr->next)      // L
  ptr->val = ptr->val + 1; // X
```

(b) Pointer chasing loop with a loop-carried dependence

Figure 1.1: Example loops. The loop in (a) can be parallelized using IMT. The loop in (b) can be parallelized with CMT or PMT. Sample parallelizations are shown in Figure 1.2

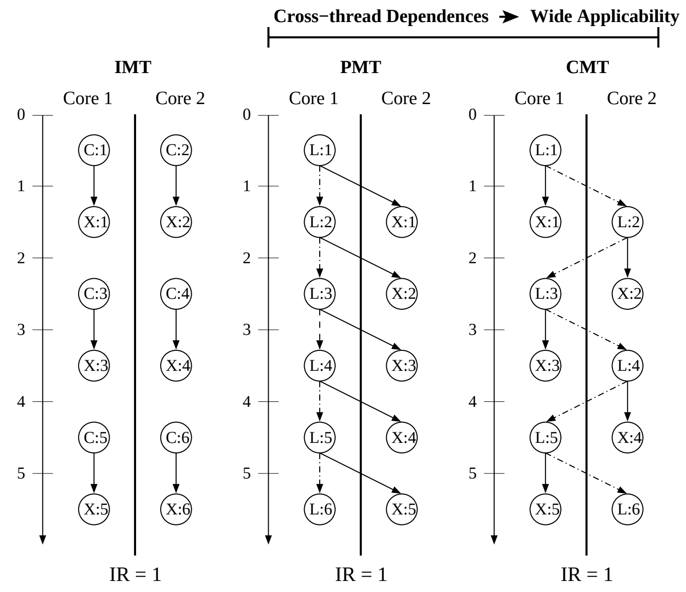
processors on the chip improve the throughput of many independent tasks, they, by themselves, do nothing to improve the performance of individual tasks. Worse still for task performance, processor manufacturers are considering using simpler cores in CMPs to improve power/performance. This trend implies that single task performance will not improve, and may actually degrade. Thus, performance improvement on a CMP requires programmers or compilers to parallelize individual tasks into multiple threads and expose thread-level parallelism (TLP).

1.1 Thread Level Parallelism Paradigms

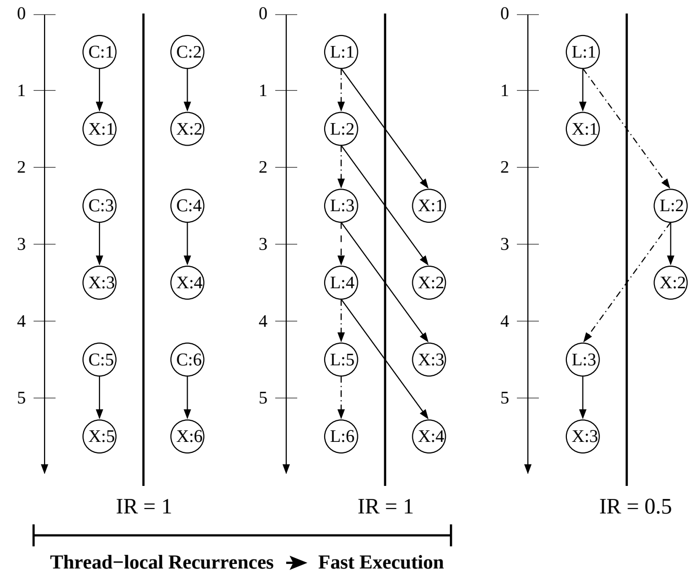
The major challenge in parallelizing a single task into multiple threads is handling the dependences between threads. Parallelization techniques must insert synchronization between threads to communicate these dependences. Unfortunately, if not handled carefully, synchronization for inter-thread dependences can eliminate parallelism by serializing execution across threads. Depending on the kind of inter-thread dependences they create, TLP techniques can be categorized into three principle paradigms: independent multithreading (IMT), cyclic multithreading (CMT), and pipelined multithreading (PMT). Figures 1.1 and 1.2 summarize the differences between these three paradigms.

1.1.1 Independent Multithreading (IMT)

In IMT, no inter-thread dependences exist. Independent threads execute concurrently to provide thread-level parallelism (TLP). DOALL loop parallelization [34] is a prime exam-



(a) Communication Latency = 1



(b) Communication Latency = 2

Figure 1.2: Execution schedules of the loops in Figure 1.1 using IMT, CMT, and PMT. Solid lines represent intra-iteration dependences. Dashed lines represent loop-carried (critical path) dependences. IR is the *initiation rate*, the number of iterations started per cycle.

ple of IMT. For example, consider the loop in Figure 1.1(a). When parallelized into two threads, it would execute according to the schedule shown in the IMT column of Figure 1.2, initiating, on average, one iteration per cycle and achieving a speedup of 2 over the sequential code. Unfortunately, while the parallelism achieved by IMT is often significant, its applicability to general-purpose codes is low. For example, DOALL parallelization could not be applied to the code in Figure 1.1(b) because of the pointer-chasing dependence. In general, the highly interdependent nature of code blocks in general-purpose programs routinely create such recurrences, resulting in few, if any, large, independent pieces of code for IMT.

1.1.2 Cyclic Multithreading (CMT)

CMT techniques handle programs with recurrences. Threads from CMT are characterized by one or more inter-thread dependence cycles. To better understand CMT, consider applying DOACROSS parallelization [13] to the loop in Figure 1.1(b). The pointer-chasing recurrence is mapped across threads such that independent code blocks from different loop iterations (for example, LD2 and X3) execute concurrently to provide TLP, as shown in the CMT column of Figure 1.2. Notice that the bi-directional communication between the two threads creates a cyclic dependence between threads.

By handling codes with recurrences, CMT enjoys significantly more applicability in general purpose codes than IMT. However, by mapping recurrences, which are on the critical path of a program's execution, across threads, CMT causes program performance to be highly sensitive to inter-thread communication latencies. Therefore, such techniques suffer a serious setback in the face of high inter-thread communication latencies in current and future generation CMPs. In the DOACROSS example shown in Figure 1.2, the rate at which loop iterations can be initiated (and thus the execution time for the loop) is limited by the communication latency. As the communication latency increases from 1 to 2, the rate of iteration initiation drops from 1 to 0.5, thus highlighting the sensitivity of CMT program performance to communication latency. In reality, as more and more cores are integrated on the same die, increased wire delays and contention in the shared memory subsystem may cause inter-core operand communication latencies to vary from few tens of cycles to even few hundreds of cycles, leading to poor CMT performance.

1.1.3 Pipelined Multithreading (PMT)

It is in this context that the relatively unheard of DOPIPE parallelization technique [41][1] or more generally, pipelined multithreading, assumes significance. PMT, like CMT, handles codes with recurrences, but differs in that the resulting inter-thread dependences do not form a cycle, but rather a pipeline (more precisely an arbitrary directed-acyclic graph). By parallelizing codes with recurrences, PMT achieves the wide applicability of CMT, and by avoiding cyclic cross-thread dependences, PMT parallelized codes can tolerate long inter-thread communication latencies. Application of PMT to the loop in Figure 1.1(b) produces threads whose execution schedule is shown in the PMT column in Figure 1.2. Notice that the cross-thread dependences are acyclic; they only flow from thread 1 to thread 2. Therefore, the initiation rate does not change as the communication latency increases; one loop iteration can be initiated each cycle regardless of the communication latency. Communication latency only affects the pipeline "fill time" which is a one-time cost amortized over the entire execution of the loop.

The compelling advantages of PMT make it a strong candidate for use in program parallelization for current and future multi-core architectures. This dissertation presents a *non-speculative* PMT loop transformation called decoupled software pipelining (DSWP). It shows that the acyclic dependence flow property of PMT can not only enable DSWPed

[1]DOPIPE was originally proposed as a multithreading technique alongside DOACROSS to handle scientific codes with recurrences. Since the loop body of threads produced by DOACROSS remains identical across all threads, it enables DOACROSSed codes to spawn enough threads to match the available processor count. DOPIPE, on the other hand, splits the original loop's body among multiple threads and as a result the number of threads is essentially fixed at compile-time. The run-time scalability requirements of the scientific computing community resulted in DOACROSS becoming the preferred parallelization strategy.

applications to tolerate inter-core latencies, but also to effectively insulate critical path code from stalls due to variable latency instructions or code blocks in the off-critical path. This work then studies the performance scalability of DSWP in general-purpose applications and the communication support requirements for DSWP.

1.2 Contributions

- Decoupled Software Pipelining (DSWP) transformation: The performance of a program is limited by its critical path performance. Ideally, this critical path consists of only program dependences. However, single program counter (PC), limited dynamic scope and sequential commit requirements in traditional single-threaded processors lead to *false* resource dependences among instructions, prevent processors from understanding global instruction criticality and cause misprioritized instruction fetch and execution. For example, a TLB miss for an output-producing store can clog a pipeline, stalling progress down a pointer-chase dependence chain. Though a static compiler has much wider scope, the presence of variable latency instructions (e.g. loads) or code blocks (e.g. if-then-else hammocks) prevents single-threaded static code transformations like software pipelining from correctly identifying and optimally scheduling the global critical path.

  The decoupled software pipelining transformation presented in this work addresses the problem of prioritizing the execution of global critical path by statically splitting a given program loop into critical path and off-critical path instructions and executing them as concurrent threads on a CMP, or in general, on any multi-context architecture. Inter-thread dependences always flow from the critical path thread to the off-critical path thread. The unidirectional dependence in the resulting PMT code facilitates the use of a decoupling queue to buffer inter-thread values. While multi-context execution enables prioritized fetch and execution of critical path instructions, the decoupling buffer insulates either thread from variable latency stalls in the other thread and thus provides latency tolerance. This creates a non-contiguous logical instruction window that ensures that even though one of the contexts may be stalled, the other context(s)/thread(s) can continue to make forward progress by executing independent instructions from earlier loop iterations. In effect, they create a large effective instruction window, enabling DSWP to effectively tolerate variable latency stalls.

- Performance Scalability: The dissertation then presents a scalability analysis of DSWP, as a general-purpose multithreading strategy, by studying its performance when going up to 8 threads using an automatic DSWP framework [40]. The analysis identifies thread pipelines as being of two types - *linear* and *non-linear*. Linear pipelines are characterized by strict pairwise interactions amongst threads, i.e. each thread in the pipeline consumes from at most one upstream thread and produces to at most one downstream thread. Non-linear thread pipelines, on the other hand, are directed acyclic graphs. Even though, in principle, there are no cyclic inter-thread dependences in a PMT transformation such as DSWP, the use of finitely sized queues creates cyclic inter-thread dependences called *synchronization cycles*. Synchronization cycles require a producer of a queue item to block until the queue is non-empty. For reasonably sized queues (for example, 8 entries and above) and linear pipelines, synchronization cycles never lead to performance bottlenecks, since the slack provided by the queue sizing is sufficient to tolerate them. However, it will be shown that, for similarly sized queues, the synchronization cycles for non-linear pipelines have such high latencies that the slack provided by queue sizing is not sufficient to tolerate the long delays. While this was a major cause of performance degradation, for a few applications, increased coherence misses was determined to be the source of performance loss. The analysis concludes that a more communication-aware DSWP partitioning algorithm, possibly involving code duplication among threads, could avoid communication bottlenecks by generating only linear pipelines.

- Inter-Thread Operand Communication Support: While pipelined inter-thread communication tolerates inter-core or transit delay naturally, the intra-core overhead from fetching and executing communication operations, COMM-OP delay, is a recurring cost and application performance is highly sensitive to this overhead. In particular, threads in DSWPed programs communicate very frequently (once every 5-20 dynamic instructions) and are very sensitive to COMM-OP delay. In designing support mechanisms to sustain such high-frequency pipelined communication, also referred to as high-frequency *streaming* communication, it is important to optimize for performance, hardware cost, operating system overhead and hardware and OS design and modification effort.

  Based on a methodical design space exploration of several communication mechanisms, this dissertation presents three novel high-performance solutions for high-frequency PMT communication, with varying levels of hardware and OS impact. They are:

  1. Synchronization Array: This design uses a simple software interface, `produce` and `consume` instructions to write and read data to and from an inter-thread queue respectively, and uses dedicated hardware for synchronization and operand transport to deliver high inter-thread communication performance. The synchronization array, while conceptually a collection of queues, separates storage for synchronization information from storage for queue data and has the former distributed across cores. This enables `produce` and `consume` instructions to test queue fullness or emptiness locally (i.e. within a core), thereby enabling fast local processing and avoiding contention for any shared structure. A detailed design of the synchronization array, including handling of predicated `produce` and `consume` instructions, dealing with control speculative accesses and guar-

anteeing memory consistency, is presented.

2. Snoop-based synchronization This design maps `produce` and `consume` instructions to shared memory queues and includes extra logic in the cache controller to provide efficient inter-thread synchronization to deliver low-cost low-overhead communication between threads. Hardware enhancements to the basic snoop-based synchronization design that use a special *stream cache* and longer queue sizes are also discussed and evaluated.

3. Synchronization coalescing: This is a compiler technique to amortize synchronization and queue pointer update overhead across multiple communication operation sequences for shared memory based software queue implementations. It achieves significant performance improvement over existing shared memory software queue implementations.

1.3 Overview

The remainder of this dissertation is organized as follows. Chapter 2 highlights the misprioritization of program critical path in traditional single-threaded execution and presents the decoupled software pipelining (DSWP) transformation to effectively tolerate stalls due to variable latency instructions or code blocks. Chapter 3 presents a qualitative discussion on the importance of efficient inter-thread communication in PMT, identifies four essential components for any streaming communication support, and discusses the pros and cons of various design points along each axes in a detailed design space description. Subsequent to the design space description, this chapter introduces the high-performance synchronization array design and the snoop-based synchronization design as a low-cost alternative to the former. Based on the qualitative arguments presented in the design space description, it argues for why the snoop-based synchronization technique should perform almost on par with the synchronization array design.

Information about the evaluation infrastructure, the benchmarks used, analysis and performance measurement methodology is provided in Chapter 4. Evaluation of variable latency tolerance property of DSWP as well as the performance scalability study is presented in Chapter 5. While the experiments in Chapter 5 use the synchronization array communication architecture on account of its high performance design, Chapter 6 evaluates existing shared memory software queue implementations, alongside the synchronization array and the snoop-based synchronization mechanisms, to identify the key performance bottlenecks in each design point. Chapter 7 presents the synchronization coalescing and the stream cache optimizations which alleviate the bottlenecks identified in Chapter 6 and improve the performance of existing shared memory implementations and the snoop-based synchronization technique respectively. Chapter 8 summarizes key results from this work, highlights the relevance of this work to current trends in the microprocessor industry, and provides pointers to future research directions.

Chapter 2

Decoupled Software Pipelining

This chapter first presents background material and highlights the key problems in dealing with variable latency stalls in modern processors. Next, it argues for a multithreading solution to the above problem, not because processors ship with multiple cores these days, but more so to overcome the fundamental limitations of traditional single-threaded processors. In fact, besides power and thermal issues and difficulty in scaling up the clock rate, diminishing performance returns from single-threaded execution is also a key reason for the current paradigm shift. The decoupled software pipelining (DSWP) PMT transformation presented in this chapter addresses these fundamental limitations. Finally, this chapter alludes to how this transformation can be generalized and used as a general-purpose multithreading technique.

2.1 Limitations of single-threaded execution

Performance of any region of code, be it a whole program, a procedure, a loop, or a basic block, is limited by the performance of its dynamic critical path. Ideally, the critical path would contain only program dependences, including register, memory and control dependences and all non-critical instructions would be scheduled and executed completely in the “shadow” of the critical path (i.e. overlapped with the critical path).

Unfortunately, conventional microprocessors seldom achieve this overlap in practice. Traditional compilers take into account resource constraints and schedule all instructions, critical and non-critical, according to some heuristic and present a unified instruction stream to the underlying machine. The compiler assumes a fixed latency for each instruction as it generates this schedule. The hardware fetches and commits instructions in the order determined by the compiler. At runtime, single-threaded execution requires all instructions, regardless of their criticality, to uniformly compete for and share available resources (for example, fetch resources, register file ports, functional units, reorder buffer slots, etc.). The sequential fetch and commit necessarily creates a tight coupling between instruction execution on the critical and off-critical paths. This tight coupling combined with the fact that the compiler had to assume a fixed latency for all instructions while generating the instruction schedule leads to performance bottlenecks in the presence of variable latency instructions.

While static scheduling is very effective at hiding fixed latencies by placing independent instructions between a long latency instruction and its consumer, unfortunately, the compiler does not know how many instructions must be placed between a variable latency instruction and its consumer. Assuming best, average, or worst case latency for scheduling variable latency instructions can detrimentally affect overall run-time performance of the code. Out-of-order (OOO) execution mitigates this problem to an extent. Rather than stalling when a consumer, whose dependences are not satisfied, is encountered, the OOO processor will execute instructions from after the stalled consumer. However, since critical and non-critical instructions are fetched and executed with equal priority in traditional single-threaded execution, not only does it cause non-critical instructions to take up valuable machine resources causing resource hazard stalls for critical instructions, but it also prematurely exhausts the supply of independent instructions whose execution could have been overlapped with stalls on the critical path. As Section 2.3 will show, even an OOO processor will fall short of independent instructions to overlap with long stalls of variable latency instructions due to the sequential fetch and commit policy of traditional single-threaded execution. The problem can be reduced to that of maintaining a steady supply of independent instructions to overlap with periods of long stalls due to variable latency instructions. This situation calls for a more flexible instruction scheduling that will *not* tightly couple execution of critical and off-critical paths and instead, will allow each of them to make forward progress independent of stalls in the other path.

The predominant type of variable latency instructions, memory loads, have worst case latencies (i.e., cache-miss latencies) so large that it is often difficult to find sufficiently many independent instructions after a stalled consumer. Despite these limitations, static and dynamic instruction scheduling combined with larger and faster caches have been largely successful in improving instruction-level parallelism (ILP) for many programs. However, the problems described above become particularly pronounced in certain programs that have unpredictable memory access patterns and few independent instructions near long latency loads. Loops that traverse recursive data structures (e.g., linked lists, trees, graphs, etc.) exhibit exactly these characteristics. Data items are not typically accessed multiple times and subsequent data accesses are referenced through pointers in the current structure, resulting in poor spatial and temporal locality. Since most instructions in a given iteration of the loop are dependent on the pointer value that is loaded in the previous iteration, these codes prove to be difficult to execute efficiently.

To address these concerns, software, hardware, and hybrid prefetching schemes have been proposed [33, 53, 54]. These schemes increase the likelihood of cache hits when traversing recursive data structures, thus reducing the latency of data accesses when the data is actually needed. Software techniques insert prefetching load instructions that attempt to predict future accesses using information statically available to the compiler [33]. Hardware techniques dynamically analyze memory access traces and send prefetch requests to the memory subsystem to bring data into the processor caches [53]. Dependence-graph-based prefetching constructs dependence graphs leading to loads with poor locality and pre-execute these graphs to do prefetching [4]. Despite these efforts, the performance of recursive data structure loop code is still far from ideal. Since prefetching techniques are speculative, they may not always end up fetching the right addresses at the right time. Achieving good coverage (i.e., the ratio of the number of useful addresses prefetched to the total number of addresses accessed) without performing an excessive number of loads is extremely difficult. The prefetching of useless data (i.e., poor prefetch accuracy) not only does not help performance but may in fact hinder it by causing useful data to be evicted from the cache and by occupying valuable memory access bandwidth. Consequently, the accuracy of prefetchers is extremely important. Achieving high coverage *and* accuracy is very difficult in these speculative frameworks.

Besides failing to efficiently handle variable latency stalls, single-threaded execution also cannot efficiently handle variable trip count loops. This inefficiency arises due to a single PC architecture, which prevents the processor from fetching and overlapping execution of pre-loop and post-loop code with loop execution. While fixed trip count loops can be fully unrolled and statically scheduled alongside pre-loop and post-loop code, the same cannot be done for variable trip count loops. Dynamically, when such a loop is encountered on the off-critical path, no critical path instructions can be fetched until all iterations of this loop have been fetched. Alternately, when such a loop is encountered on the critical path, no off-critical path instructions can be fetched and this results in missed overlap opportunities. While this problem is quite similar to the variable latency problem described above, a solution to this problem not only requires decoupling the execution of critical and off-critical paths, but also requires the ability to fetch and commit instructions out of order. Stated simply, scope restrictions in single-threaded processors prevent them from finding useful independent instructions for parallel execution and prioritizing execution of critical path instructions. The single-threaded execution model forces critical and off-critical paths to compete for and share processor fetch and execute resources with the same priority.

Researchers have proposed ways to address the scope restriction problem. Slipstream processors achieve larger effective instruction windows by executing a compressed, albeit speculative, version of a program. This is called the advance or A-stream. A redundant or R-stream, that is sped up by the control flow and operand value information from the A-stream, serves as a checker thread, validates A-stream computations and triggers recovery as needed [46]. Zilles and Sohi use master/slave speculative parallelization to extend this to beyond two threads and show that the validation of the advance or master thread can be carried out in parallel by multiple redundant or slave threads [75].

Barnes et al. used two in-order processing cores coupled by a queue [5]. During long latency events, the "advance" pipeline does not stall on use. Instead it continues instruction fetch, executes independent instructions and defers execution of instructions dependent on resolution of the long-latency event. Instructions flow through the queue from the advance pipeline to the second "backup" pipeline. The backup pipeline resolves the long latency dependences and commits instructions. The inter-core queue in effect provides a large effective instruction window, enabling in-order processing cores to execute around stall-on-use conditions. Barnes et al.'s multipass pipelining technique achieves the same effect by recirculating instructions within a single core [6]. The Kilo-Instruction Processors [12] and Checkpoint Processing and Recovery [2] techniques remove commit restrictions through checkpointing of architectural and microarchitectural state to achieve large effective instruction windows and the TRIPS [56] project uses distributed processing across multiple small processing elements to achieve the same effect.

In the last decade or so, several thread-level speculation (TLS) techniques [19, 59, 62, 68, 72] have been proposed to enable single-threaded programs to run faster on multi-core processors. These techniques build larger logical instruction windows by speculatively spawning threads/tasks/traces on available processor cores, typically based on a control flow or value predictor's predictions. At any given point, the oldest thread in sequential program order is the non-speculative thread and all other threads are speculative. Threads commit in sequential program order. The techniques differ in the exact manner in which they identify these coarse-grained regions for spawning and parallel execution. In the multiscalar [59] and superthreaded [68] architectures, statically partitioned program control flow graph (CFG) regions called tasks are speculatively spawned off to execute on available processing elements. The superthreaded execution model also includes a rudimentary form of thread pipelining. The communication and computation parts of each thread are separated out to allow maximal execution overlap among all threads. However, thread spawning and commit are done strictly in sequential program order. Trace processors [72] dynamically construct instruction traces and employ a trace predictor to determine which trace to fetch and execute. Other TLS techniques [19, 62] break data dependences by statically and/or dynamically speculating on data values and spawn new threads with predicted input values. The predictions are validated at run time.

Other techniques use speculative or subordinate execution threads to run ahead of the original program thread and trigger branch mispredictions and cache misses earlier than the original thread [9, 10]. The work performed by these speculative pre-execution threads mainly serves to warm up microarchitectural structures like caches and branch predictors, thereby improving the performance of the primary non-speculative thread. Mutlu et al.'s run-ahead execution technique avoids spawning extra threads and instead uses checkpointing of architectural state to achieve the same effect [38]. Techniques like register integration can be used to incorporate the execution results in the non-speculative thread's architectural state [55].

Despite the abundance of innovative microarchitectural techniques presented above and summarized in Table 2.1, a common problem across all these techniques is that they were evolved to operate under the constraints of the traditional sequential single-threaded execution model. Instruction fetch and commit are *still* serialized. In a bid to overcome the fundamental restrictions imposed by the execution model, architects have gone to great lengths to build copious amounts of buffering in the processing core to hold un-issued or un-committed instructions or to hold speculative architectural state updates of executed-but-uncommitted instructions.

Name	Category	Key Idea
SlipStream [46],Master/Slave Speculative Parallelization [75]	IW size increase	Compressed approximate instruction stream gets to performance critical loads/branches faster. One or more trailing threads or streams validate the execution of the compressed stream.
Flea-Flicker [5, 6]	IW size increase	Unresolved instructions are held in intra- or inter-core queues for deferred processing. Routine sequential commit upon resolution.
Kilo-Instruction Processors [12], Checkpoint Processing and Recovery [2]	IW size increase	Checkpoint of arch and μarch state saved on blocked ROB conditions to continue instruction retirement past the blocked instruction. Restored if blocked instruction excepts/interrupts.
TRIPS [56]	IW size increase	Static mapping of instructions to distributed grid of OOO processing elements.
TLS [19, 59, 62, 68, 72]	IW size increase	Input dependences of dynamically far apart regions speculated and regions spawned as speculative threads on available cores. Committed in program order upon validating speculation.
Run-ahead execution [38]	μarch warmup	Architectural state checkpointed on blocked instruction. Instruction fetch and execution continue past blocked instruction in "run-ahead" mode and serve to prefetch data and instructions. Upon resolution of blocked instruction, pipeline is flushed and normal mode execution is resumed.
Subordinate microthreading [9]	μarch warmup	Micro-coded threads spawned off at strategic points from main thread. They issue prefetches or update branch predictors.
Speculative pre-computation [10, 55]	μarch warmup	p-thread slices warmup caches. Results can be optionally register integrated with the main thread.

Table 2.1: Multi-core techniques to improve single-threading performance

The decoupled software pipelining transformation, presented in this chapter, overcomes these limitations. The next section presents an overview of the transformation and its salient characteristics.

2.2 Overview of DSWP

To tolerate variable latency and to overcome scope restrictions imposed by single PC architectures, a program transformation called *decoupled software pipelining* (DSWP) is presented in this chapter. DSWP avoids heavy hardware usage by attacking the fundamental problem of working within a single-threaded execution model, and moving to a multi-threaded execution model. Only useful, about-to-execute instructions are even brought into the core. The rest are conveniently left in the instruction cache or their results committed and retired from the processor core. Unlike the single-threaded multi-core techniques presented in Table 2.1, DSWP is an entirely *non-speculative* technique. Each DSWP thread performs useful work towards program completion. In other words, DSWP does not require any speculative thread spawns nor does it have to throw away work done by any thread.

DSWP threads are typically long-running and do not require frequent thread spawns.

The concurrent multithreading model of DSWP means that each participating thread commits its register and memory state concurrently and independently of other threads. Since, the exact input-output relationship among the threads is known statically, only operand values corresponding to inter-thread dependences are communicated through queues. The concurrent model avoids buffering or communication of all other thread-local results which are committed independently. These factors make the amount of inter-thread queue storage for DSWP insignificant, compared to the speculative storage in techniques like TLS and other single-threaded multi-core techniques.

The remainder of this chapter is organized as follows. Section 2.3 examines the pattern of recursive data structure traversals and illustrates why current static and dynamic scheduling techniques are not effective in identifying parallelism. Section 2.4 describes how DSWP parallelizes single-threaded pointer-chasing loops in the context of a simultaneous multithreading (SMT) core or a chip multiprocessor (CMP). While this chapter focuses on the DSWP transformation, Chapter 3 provides a detailed discussion on various communication support options for DSWP.

2.3 RDS Loops and Latency Tolerance

As was mentioned earlier, recursive data structure loops suffer from poor cache locality, requiring aggressive strategies to tolerate latency. Figure 2.1 illustrates a typical RDS loop. Each iteration of the loop processes a node in the data structure and contains code to fetch the next node to be processed. The diagram below the program code illustrates the data dependences that exist among the instructions in the program. As is evident from the figure, the critical path in this loop's execution is the chain of load instructions resulting from the loop-carried dependence of `r1`. Ideally, all of the loop's computation could be overlapped with the computation along this critical path.

```
while(ptr = ptr->next) {
  ptr->val = ptr->val + 1;
}
```

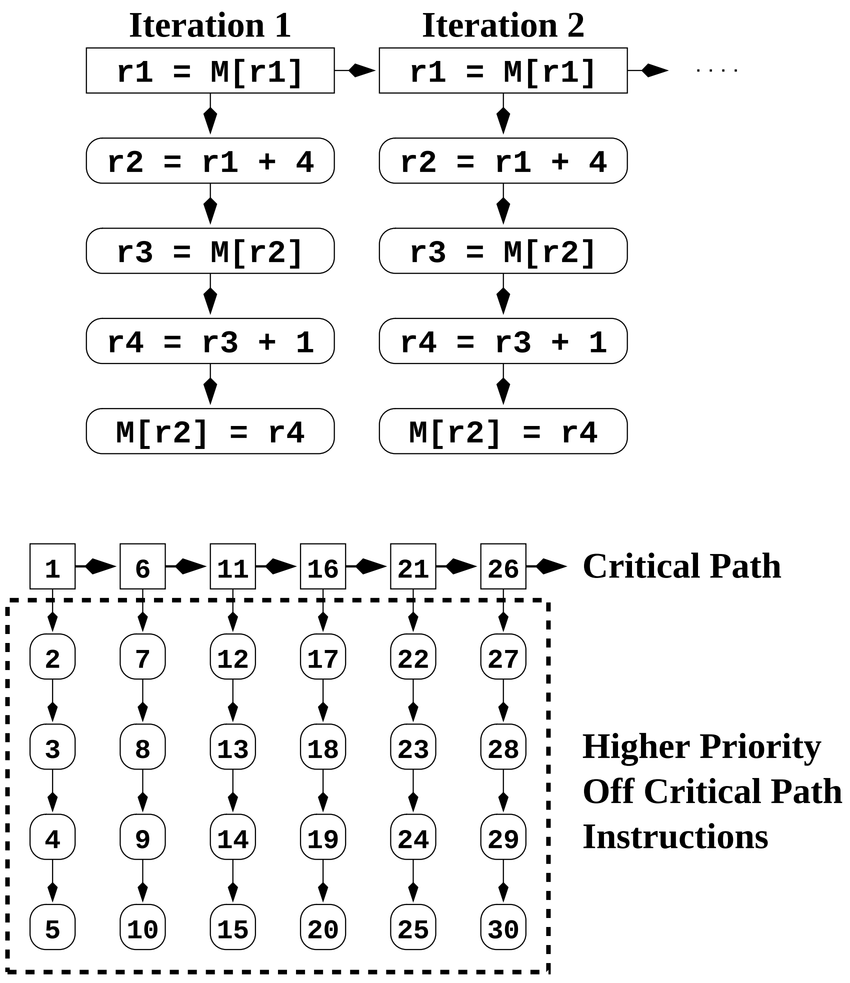


Figure 2.1: RDS loop traversal illustrating misprioritized instruction execution

Unfortunately, modern single-threaded processors, both in-order and out-of-order, implement a sequential fetch and dispatch policy. Thus, rather than attempting to execute along the loop's critical path, the processor fetches sequential instructions, delaying the execution of the next critical path instruction. In the figure, this in-order fetch policy results in the 24 instructions inside the dashed box to be fetched and dispatched to allow the fetch and dispatch of six instructions on the critical path. This misappropriation of resources to instructions off the critical path is not restricted to fetch. Dynamic schedulers typically give priority to the oldest instruction in the dynamic scheduling window. Thus, even during execution in an out-of-order processor, priority is given to relatively unimportant instructions rather than those on the critical path.

This misprioritization of instructions can result in lost opportunities for overlapping latency. Consider, for example, the choice between executing instruction 6 or instruction 2 following the completion of instruction 1. If instruction 2 is selected in favor of instruction 6, and instruction 6, when executed, would miss in the cache, then executing instruction 2 first will not only delay this miss by one cycle but will also consume an instruction (i.e., itself) whose execution could have been overlapped with the miss. Even if instruction 6 were to hit in the cache, executing down the 2-3-4-5 path of instructions would delay the execution of instruction 6, which ultimately will delay a cache miss on one of the critical path loads. Each instruction executed preferentially over a ready critical-path instruction is an instruction that delays the occurrence of a future critical-path cache miss *and* one that removes an opportunity for latency tolerance by decreasing the pool of independent instructions.

Note that the delays incurred by executing off-critical-path instructions need not be short. For example, as the length of a loop increases, the number of cycles required to fetch a subsequent copy of the loop-carried load may become significant. Alternatively, cache misses off the critical path could cause chains of dependent instructions to collect in the machine's issue window thus forcing the system to stall until the load returns from the cache. For example, if, in Figure 2.1, instructions 3 and 8 miss in the cache, a nine instruction reorder buffer (ROB) would be necessary to start the next critical path load. A smaller ROB would result in the subsequent critical path load being delayed until the cache misses resolve.

Avoiding such delays is possible in aggressive out-of-order processors but would require the addition of significant resources. In the tiny example loop shown in Figure 2.1, promptly bringing a critical-path load into the instruction window would cost five instructions of issue/dispatch width and five entries in the ROB. While possibly not impractical for loops of size five, as loop lengths increase, the cost of tracking *all* instructions between consecutive critical-path instructions will become excessive.

The same effect can be achieved without requiring excessive resources in the pipeline. In a multi-threaded architecture, it is possible to partition the loop into multiple threads such that one thread is constantly fetching and executing instructions from the critical path, thus maximizing the potential for latency tolerance. The next section presents the Decoupled Software Pipelining (DSWP) transformation to create these threads from a sequential loop.

2.4 RDS Parallelization

As illustrated in the previous section, RDS loops typically consist of two relatively independent chains of instructions. The first, which makes up the critical path of execution, is the traversal code. The second is the chain of computations performed on each node returned by the traversal. While the data dependence of these sequences is often unidirectional (i.e., the computation chain is dependent on the traversal chain, but not vice-versa), variable latency instructions coupled with processor resource limitations create artificial dependencies between the two chains. While the resulting stalls are more pronounced in in-order processors, they can cause prolonged stalls even on aggressive out-of-order machines.

To overcome this, it is necessary for the processor architecture to allow for a more

```
while(ptr = ptr->next) {
  ptr->val = ptr->val + 1;
}
```

(a) Recursive Data Structure Loop

```
while(ptr = ptr->next) {
  produce(ptr);
}
```

(b) Traversal Loop

```
while(ptr = consume()) {
  ptr->val = ptr->val + 1;
}
```

(c) Computation Loop

Figure 2.2: Splitting RDS Loops

decoupled execution of the two RDS loop pieces. Ideally, variable latency instructions from one piece should not impact code from the other piece unless the instructions from the two pieces are in fact dependent. To achieve decoupled fetch and execution, this dissertation proposes decoupled software pipelining (DSWP) which statically splits the original RDS loop into two distinct threads and executes the threads on a thread-parallel processor such as an SMT [69] core or chip-multiprocessor (CMP) [20] system. By enforcing uni-directional dependences amongst threads, this technique facilitates the use of inter-thread queues to buffer values and provide pipelined communication. More importantly, the queue doubles up as a decoupling buffer and insulates either thread from variable latency stalls in the other thread. Thus, DSWPed execution will allow the traversal code to run unhindered by computation code. Further, since the traversal code defines the longest data dependence chain in the loop, executing instructions from this piece as quickly as possible is critical to obtaining maximum performance in RDS traversal loops.

Consider the loop shown in Figure 2.1, reproduced in Figure 2.2a. The traversal slice consists of the critical path code, `ptr=ptr->next` and the computation slice consists of `ptr->val=ptr->val+1`. A natural way to thread this loop into a traversal and computation thread is shown in Figure 2.2. In the figure, the `produce` function enqueues the pointer onto a queue and the `consume` function dequeues the pointer. If the queue is full, the `produce` function will block waiting for a slot in the queue. The `consume` function will block waiting for data, if the queue is empty. In this way, the traversal and computation threads behave as a traditional decoupled producer-consumer pair.

The above parallelization lets the traversal thread make forward progress even in the event of stalls in the computation thread, and thus the traversal thread will have an opportunity to buffer data for the computation thread's consumption. The buffered data also allows the computation thread to be relatively independent of the stalls in the traversal thread since its dependence on the traversal thread is only through the `consume` function, and its dependences will be satisfied by one of the buffered values. Thus, this parallelization effectively decouples the behavior of the two code slices and allows useful code to be overlapped with long variable-latency instructions without resorting to speculation or extremely large instruction windows.

In addition to tolerating latency in the computation slice, the proposed threading also allows the traversal thread to execute far ahead of the corresponding single-threaded program due to the reduced size of the traversal loop compared to the original loop. On machines with finite issue width, this reduced size translates into more rapid initiation of the RDS traversing loads provided that the previous dynamic instance of the loads have completed. Thus, the reduced size loop allows the program to take better advantage of traversal cache hits to initiate traversal cache misses early. From an ILP standpoint, this allows for an overlap between traversal and computation instructions from distant iterations.

This threading technique is called decoupled software pipelining to highlight the parallel execution of the threads. The decoupled flow of data from the traversal to the work threads "stages" through the inter-thread queue, which can be implemented with any of the designs presented in Chapter 3.

2.5 Decoupled Software Pipelining

Thus far, it was assumed that the hardware would be provided with properly parallelized code. This parallelization is a tedious process and should be performed in the compiler. This section presents a decoupled software pipelining algorithm suitable for inclusion in a compiler.

Although RDS loop parallelization would be typically performed at the assembly code level, this process is illustrated here in `C` for clarity. Consider the sample code of Figure 2.2a. As was mentioned previously, this loop structure is typical of recursive data structure access loops. Line 1 fetches the next data item for the computation to work on, and line 2 performs the computation in the loop. In order to parallelize the loop, the pieces of the code that are responsible for the traversal of the recursive data structure must first be identified. Since a data structure is recursive if elements in the data structure point to other instances of the data structure, either directly or indirectly, RDS loops can be identified by searching for this pattern in the code. Specifically, load instructions that are data dependent on previous instances of the same instruction must be identified. These induction pointer loads (IPL) form the kernel of the traversal slice [39]. IPLs can be identified using augmented techniques for identifying induction variables [16, 39]. In the example in Figure 2.2a, the assignment within the *while* condition expression on line 1 is the IPL.

Once the IPL is identified, the backward slice of the IPL forms the base of the traversal thread. In Figure 2.2a, the backward slice of the IPL consists of the loop (i.e., the backward branch), the IPL, and the initialization of `ptr`. To complete the thread, a `produce` instruction is inserted to communicate the value loaded by the IPL and, if initialization code exists, an instruction to communicate this initial value. Figure 2.2b illustrates the traversal thread code that would arise from applying this technique to the loop shown in Figure 2.2a.

The computation thread is essentially the inverse of the traversal thread. Both loops share identical structure. Those instructions that are not part of the backward slice of the IPL but within the loop being split are inserted into the computation thread. In place of the IPL, a `consume` instruction is inserted. Just as in the traversal thread, it is necessary to include `consume` instructions to account for loop initialization. Figure 2.2c shows the code computation thread corresponding to the original loop shown in Figure 2.2a. For each data dependent pair of instructions split across the two threads, a dependence identifier is assigned to the corresponding producer-consumer pair.

In order for this algorithm to be effective, the compiler must be able to identify the traversal slice, including dependent memory operations. Existing memory dependence analysis techniques identify if there are stores in the loop that will affect later traversals (i.e., loops that modify the RDS). DSWP must handle these loops appropriately or exclude them from optimization. In cases where memory analysis is too conservative, not enough instructions will be placed in the computation thread and the benefits of the technique will be negated. Further, the compiler must balance the work of the original loop between the traversal and computation thread to realize optimal performance.

While memory dependence analysis is important for decoupled software pipelining, existing analysis used to compute load-store dependences is sufficient. DSWP does not attempt to identify completely independent code to split into threads. The analysis required for a general parallelization technique is much more sophisticated. Indeed, that problem has been heavily studied and has proved extremely difficult to solve. Instead, DSWP builds two threads that operate in a pipelined fashion, where the traversal thread feeds information to the computation thread.

2.6 Automatic DSWP

While the technique outlined in the previous section is specific to RDS codes, Ottoni et al. [40] present an automatic technique to apply DSWP to generic program loops. The automatic technique makes a departure from the notion of identifying critical and off-critical paths of regions targeted for DSWP. Instead, it focuses on identifying more generic program recurrences and achieves acyclic dependence flow among threads by ensuring that no single recurrence crosses thread boundaries. This approach enables the automatic DSWP algorithm to be a truly general-purpose multithreading technique. The algorithm is described below.

The automatic DSWP technique (*autoDSWP* for short) takes a loop's dependence graph, which contains all register, memory and control dependences, as input. In order to create an acyclic thread dependence graph for pipelined parallelism, it first identifies strongly connected components (SCCs) in the input dependence graph. The graph formed by these SCCs, by definition, will be a directed acyclic graph (DAG_{SCC}). The algorithm then partitions the DAG_{SCC} into the requisite number of threads while making sure that no cyclic inter-thread dependences are created. In this manner, *autoDSWP* parallelizes program loops into pipelined threads.

The current thread model for DSWP is as follows. Execution begins as a single thread, called *primary thread*. It spawns all necessary *auxiliary threads* at the beginning of a program. When the primary thread reaches a DSWPed loop, auxiliary threads are set up with necessary loop live-in values. Similarly, upon loop termination, loop live-outs from auxiliary threads have to be communicated back to the primary thread. While this does create a cycle in the thread dependence graph, any dynamic cost due to this once-per-loop-invocation event, will be rendered negligible by long-running pipelined multithreaded loop bodies. However, this cycle can become a significant overhead for short-running loops, which are not good candidates for DSWP in the first place.

Performance improvement is obtained from coarse-grained overlap among pipelined threads and from improved variable latency tolerance provided by decoupled execution. The amount of overlap is, of course, limited by the performance of the slowest thread. Since the granularity of scheduling is a single SCC, the maximum theoretical speedup attainable from *autoDSWP* is $\frac{1}{\text{Weight of heaviest SCC}}$. Thus, there is an upper bound to the performance obtainable from merely scheduling and balancing the SCCs across available threads. Optimizations that may be needed to further break or parallelize the individual SCCs to expose more parallelism are interesting avenues for future research.

This dissertation uses code produced by an *autoDSWP* implementation in the VELOCITY compiler framework [67] to demonstrate variable latency tolerance of DSWP, to study the performance scalability of DSWP when moving to beyond two threads and to explore various communication support options for PMT.

2.7 Summary

This chapter has presented the decoupled software pipelining (DSWP) transformation, which, through concurrent execution of statically partitioned threads, exposes coarse-grained thread-level parallelism. Decoupled execution of these threads, enabled by PMT, will enable DSWP to tolerate variable latency stalls. The variable latency stall tolerance property of DSWP was first demonstrated on manually parallelized RDS codes in [51]. This presentation of DSWP in this dissertation includes an improved evaluation with automatically generated DSWP codes, more benchmarks, and a modern simulator with a validated core model and a detailed memory hierarchy.

Before proceeding to describe the various factors affecting the design of communication support for PMT in the next chapter, it is important to understand the distinction between the storage required to hold non-speculative inter-thread queue operands in DSWP and the storage required for techniques such as TLS. As mentioned in Section 2.2, DSWP, on account of its multithreaded execution model, is able to commit instructions from each thread independently of other threads. This enables DSWP to free up thread-local storage independently of other threads and only operands for true inter-thread dependences need to be buffered in inter-thread queues. This is in stark contrast to TLS techniques, wherein, the need to achieve single-threaded execution semantics, forces the system to buffer results of each and every operation of speculative threads, until they become non-speculative. The mulithreaded execution model of DSWP obviates the need for such copious storage.

The next chapter discusses various factors affecting communication support design, including design options for queue storage and queue access operations and introduces two novel communication designs - the synchronization array and the snoop-based synchronization mechanisms. The evaluation of the latency tolerance and scalability aspects of DSWP is performed with the best-performing synchronization array design and is presented in Chapter 5.

Chapter 3

Communication Support for PMT

Proper architectural support mechanisms are key to the success of any compiler technique. Chapters 1 and 2 motivated PMT, introduced the decoupled software pipelining (DSWP) loop transformation as an instance of PMT, showed that it has very desirable latency tolerance properties, and also discussed its applicability as a general-purpose multithreading technique. While PMT techniques show promise, current architectures are without sufficient architectural and operating system support to allow efficient streaming communication of data from one thread to another. It is important to understand various aspects of inter-thread communication in DSWP, before undertaking a full-fledged performance evaluation of DSWP. Accordingly, this chapter will highlight key aspects of pipelined inter-thread communication and will qualitatively discuss cost-performance tradeoffs of various communication support mechanisms.

Dedicated hardware structures like hardware FIFOs [58], and scalar operand networks (SONs) [65] result in *sub-optimal* use of hardware, since they are used exclusively for inter-thread operand transfers. Memory traffic, for instance, cannot be multiplexed on these dedicated interconnects. Besides resulting in sub-optimal use of hardware, such dedicated structures (interconnect and storage) consume extra power, demand chip redesign effort, and often necessitate complex operating system modifications to handle context switches and virtualization. In spite of these numerous concerns, dedicated hardware solutions offer the best performance. But it is important for architects to understand the cost-performance tradeoffs of such dedicated solutions vis-à-vis more cost-efficient, but low-performing, solutions.

This chapter takes a top-down approach to describing the design tradeoffs and how the choice of various design components influence these tradeoffs. The chapter argues that even though dedicated hardware solutions may have the best performance, by letting application behavior guide the design of underlying support mechanisms, it is possible to obtain low-cost low-complexity solutions that perform almost as well as high-performance solutions. It observes that even though pipelined multithreaded applications can tolerate inter-core latency or transit delay by pipelining communication through inter-thread queues, their high communication frequency makes them very sensitive to the recurring intra-thread overhead imposed by communication, referred to as COMM-OP delay. An empirical characterization of pipelined streaming applications, presented in Chapter 6, reveals that DSWPed threads communicate at the rate of once every 5 to 20 dynamic instructions. Such *high-frequency* communication entails that individual communication operations be as efficient as possible.

Armed with this understanding of the application behavior, this chapter then provides a detailed design space characterization that describes the various tradeoffs in implementing inter-thread queues for high-frequency streaming communication. It identifies four essential ingredients for any streaming mechanism: *communication operation sequences* to specify architectural reads and writes to inter-thread queues, a *synchronization* mechanism to prevent read (write) operations on empty (full) queues, intermediate storage for queue data (*queue backing store*) before they are consumed, and an *interconnect* fabric connecting processors and various backing stores. Although the design choice for each of these axes is orthogonal, certain design possibilities fit together more naturally than others.

Finally, this chapter describes two of the three novel communication designs presented in this dissertation - the synchronization array design, a dedicated hardware solution, and the snoop-based synchronization design, a low-cost alternative to the former. While the synchronization array design optimizes on all of the design axes, the snoop-based synchronization design relies on hardware support only for synchronization and uses low-cost design options for other components. The presentation of the third technique, synchronization coalescing, and the stream cache optimization to snoop-based synchronization will be motivated by the experimental evaluation in Chapter 6 and hence, will be deferred to Chapter 7.

The current chapter is organized as follows. The importance of reduced COMM-OP delay is qualitatively demonstrated in Section 3.1. Section 3.2 presents a detail characterization of the design space of streaming communication mechanisms. Sections 3.3 and 3.4 present detailed designs of the synchronization array and the snoop-based synchronization techniques respectively.

3.1 High-Frequency Streaming

This section first provides more precise definitions for *transit delay* and *communication operation (COMM-OP) delay*. Then, it characterizes streaming communication in PMT codes and illustrates why transit delays are tolerated and why reducing COMM-OP delays can increase application performance. It extends Taylor, et al.'s treatment [65] of communication latency components with a discussion of their respective impact on streaming communication.

Transit delay refers to the amount of time necessary to communicate a data value from one processor core to another. This delay is *exclusive* of all the time necessary to produce a value or to initiate communication, but rather measures the effects of signal propagation delay, bus contention, network routing latency, and the like. This delay will tend to increase with the physical separation between cores or as wire delay increases.

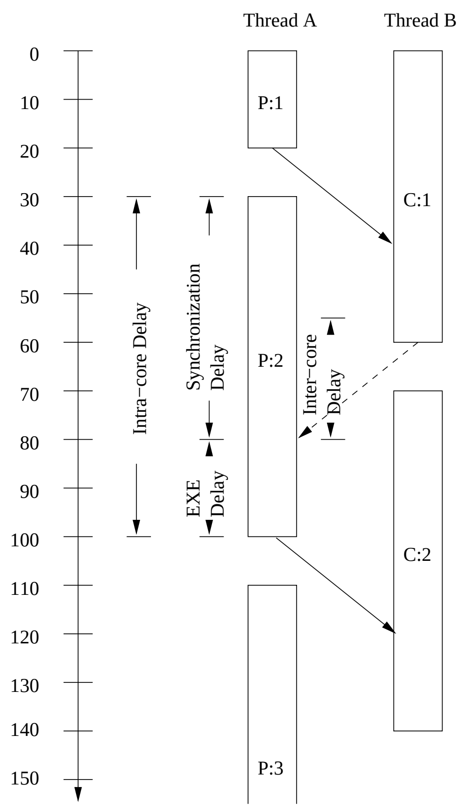


Figure 3.1: Transit and COMM-OP delays.

COMM-OP delay, on the other hand, is a measure of the overhead experienced by a *single* core due to communication. More formally, the COMM-OP delay for a particular thread is the difference between the execution time of the thread with communication operations and the execution time of the same thread when communication operations have 0-latency and consume no resources.

For shared memory communication, for example, COMM-OP delay is caused by the execution of additional instructions necessary for communicating values *and* synchronizing threads. Depending on the code containing the communication operations and the implementation details of the memory subsystem, these extra instructions can slow down a thread by occupying valuable processor resources such as fetch bandwidth, functional units, and memory ports, and by causing execution stalls due to memory fences and interconnect contention.

Figure 3.1 illustrates how COMM-OP delay and transit delay affect the execution of a

```
   Producer Thread A
L: while(ptr = ptr->next) {
P:    produce(ptr);
   }

   Consumer Thread B
C: while(ptr = consume()) {
X:    ptr->val = ptr->val + 1;
   }
```

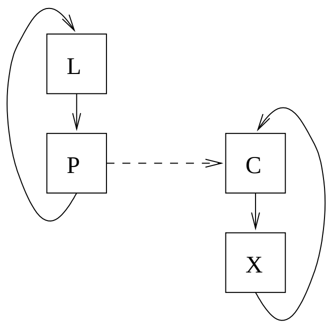


Figure 3.2: A PMT example.

pair of threads communicating via a single shared buffer (*e.g.* a shared-memory variable). To send a value from thread A to thread B, thread A executes a code sequence which ensures that the shared buffer is empty, then fills the shared buffer with the value to be communicated. The time during which this happens is labeled the COMM-OP delay or intra-core delay for thread A. Thread B will observe the value after the transit delay has elapsed. Thread B executes a code sequence that ensures the shared buffer is full, reads the value from the buffer, and finally marks the buffer empty. Only after the consumption notification from thread B reaches thread A, can another value be transferred using the same shared buffer (The notification is shown as a dashed edge from the consume operation of thread B to the producer operation of thread A in Figure 3.1).

Recall, pipelined multithreaded codes execute as concurrent, long-running, communicating threads. Figures 3.2 shows a two-thread pipeline and its control-flow graph along with its inter-thread dependence. Figure 3.3 illustrates the execution schedule of produce and consume operations of the program from Figure 3.2 under three different scenarios with progressively better communication behavior. In all three cases, the “L” and “X” operations are each assumed to take 10 cycles. Produce and consume operations are denoted by P:n and C:n respectively, where 'n' indicates the iteration number of the corresponding communication operation. To start with, the produce and consume operations will each be assumed to take 20 cycles to perform the necessary synchronization and initiate communi-

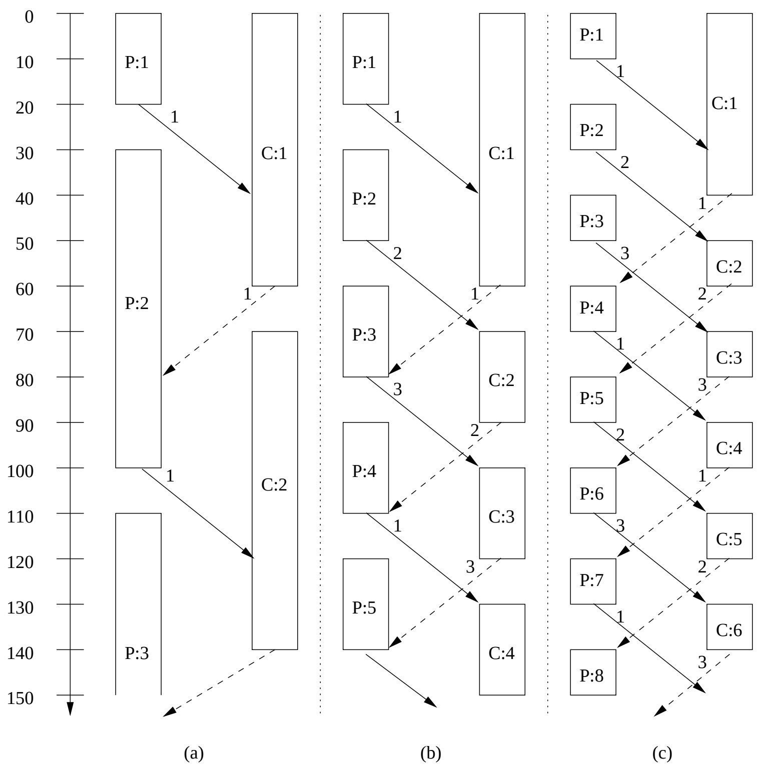


Figure 3.3: Effect of transit and COMM-OP delays on streaming codes.

cation with the other thread. The inter-thread dependence edges going from the producer to the consumer is shown as a solid arrow, while edges going the other way are shown as dashed arrows. The number on each edge indicates which queue slot the corresponding communication operation is accessing.

First, Figure 3.3a shows the execution schedule of the program using only a single shared buffer. Notice that the application is able to complete only two iterations in the 150-cycle snapshot shown. In this case, because of the nature of the communication support i.e. a single shared buffer, the transit delay becomes part of the COMM-OP delay for the produce and consume operations. This clearly is a bad design, since it negates the main benefit of PMT - that of tolerating transit delay.

However, if, instead of a single buffer, a queue of buffers is used for communication, the threads can execute more efficiently. This is illustrated in Figure 3.3b. With a queue of buffers, there are two prominent improvements - first, the COMM-OP delay of a thread can now be overlapped with the computation delay and the COMM-OP delay of the other thread and useful work can be done during transit delays and second, the extra buffering provides enough slack that transit delays are not part of COMM-OP delay anymore.

Other than the initial time taken for the first value to arrive in thread B, transit delays do not affect the timing of the system. Compared to the non-pipelined situation with only a single buffer location, the throughput (measured as iterations per time unit) has increased by a factor of 2.5. In Figure 3.3b, 5 iterations are executed in 150 cycles. (For computing throughput, the producer's performance shall be used since the consumer is affected by the one-time fill delay.)

The time taken to complete one loop iteration is given by the sum of computation time and COMM-OP delay. While pipelining can remove transit delays from the critical path by overlapping it with useful computation, the 20-cycle intrinsic COMM-OP delay continues to be a recurring overhead for every loop iteration and serves to prolong the loop iteration time. Unlike transit delays COMM-OP delays cannot be wished away altogether since every loop iteration involves communication from one thread to another. Therefore, the best one can hope for, is to minimize the overhead for individual communication operations as much as possible to improve the performance of PMT codes, especially for PMT codes with high-frequency streaming behavior. Figure 3.3c shows that 8 loop iterations can be completed in 150 cycles by reducing the COMM-OP delay from 20 to 10 cycles.

Notice that this is accomplished with exactly 3 inter-thread buffer locations, as was the case with Figure 3.3b. The performance improvement is entirely due to reduced COMM-OP delay.

Recognizing the distinction between COMM-OP delay and transit delay will serve as a guide when exploring the design space of communication mechanisms. Section 3.2 discusses the pros and cons of several streaming communication design points and how each of them deal with the two different latency aspects of inter-thread communication.

3.2 Design Space

In designing high-frequency streaming support for future CMPs, architects will have to make trade-offs between hardware (area) costs, design effort, and OS costs to come up with the best design to meet desired performance goals. Any streaming support mechanism has four essential ingredients: *communication operation sequences* to specify architectural reads and writes to inter-thread queues, a *synchronization* mechanism to prevent read (write) operations on empty (full) queues, intermediate storage for queue data (*queue backing store*) before they are consumed, and an *interconnect* fabric connecting processors and various backing stores. For exposition purposes, interconnects will be split into two sub-axes *dedicated interconnects* and *pipelined interconnects*. Although the design choice for each of these axes is orthogonal, certain design possibilities fit together more naturally than others.

3.2.1 Communication Operation Sequences

To avoid oversubscribing a processor core's fetch and execution resources, the communication operation sequences cannot be too long. Additionally, to avoid extending the loop critical path, the dependence height of the sequence must also remain relatively short. Finally, to enable decoupling between producer and consumer loops, the code sequences must allow queuing behavior; sequences that use only a single buffer location for communication should be avoided.

Software Queues Using Shared Memory

- **Description:** Producer/consumer communication and synchronization can be implemented on conventional shared memory multiprocessors using software queues. Code for such an implementation is shown in Figure 3.4. The queue is composed of a head index, a tail index, and a shared memory array of condition variable, data item

```
void produce(int value) {
  // spin until tail empty
  while(q[tail].full);
  // q[tail].full == 0
  q[tail].data = value;
  q[tail].full = 1;
  tail = (tail+1)%q_size;

}
```

```
int consume() {
  // spin until head full
  while(!q[head].full);
  // q[head].full == 1
  value = q[head].data;
  q[head].full = 0;
  head = (head+1)%q_size;
  return value;
}
```

Figure 3.4: Produce and consume code sequences for shared-memory based software queues.

pairs. The tail (head) index is updated exclusively by the producer (consumer). To access the queue, the producer (consumer) spins until the condition variable for the tail (head) queue slot indicates the slot is empty (full). Once the queue slot becomes available, the producer (consumer) can write (read) the data to (from) the queue and signal the condition variable that the slot is now full (empty). Once the data item is written (read), the tail (head) index should be updated to point to the new tail (head) slot. Since only a single thread will produce data into each queue and only a single thread will consume data from each queue, the head and tail pointers can be stored locally on the consumer and producer cores respectively. Additionally, no mutexes are required to protect the queue (although, the appropriate memory fence instructions are required to enforce the correct ordering of operations). Use of fine-grained condition variables allow an efficient implementation of software queues [63].

- **Advantages:** The key advantage of this methodology is that it requires no modifications to existing instruction set architectures (ISA) or microarchitectures. Conventional shared memory mechanisms (cache coherence and memory consistency) provide the complete foundation for this software implementation.

- **Shortcomings:** Its main drawback is that the code sequences to produce and consume a single datum are quite lengthy. The C code shown in Figure 3.4 will likely expand into many instructions. The COMM-OP delay overhead resulting from these additional instructions and dependences may offset any gains obtained by partitioning the original code among multiple threads. Further, the presence of uncounted loops in these code sequences make static ILP techniques inapplicable, and the presence of memory fence operations limit dynamic ILP and leave very little scope for performance improvements.

Produce and Consume Instructions

- **Description:** Producer/consumer communication can also be implemented by augmenting an existing ISA with special `produce` and `consume` instructions [15, 45]. The `produce` instruction has a source operand identifying a particular hardware queue, and a source operand to produce. Similarly, the `consume` instruction has a source operand to identify the queue to consume from and a destination operand to hold the read value. The hardware is responsible for delivering values between cores and for blocking the pipeline when attempting to either write to full queues or read from empty queues. In a system with more than two cores, application registers could be used to configure the target core for each hardware queue. The specific hardware used to implement this is independent of the ISA as long as the queue semantics are guaranteed.

- **Advantages:** This methodology overcomes many of the shortcomings experienced by software queue implementations. The produce and consume instruction sequences are reduced from tens of instructions down to a single instruction, resulting in smaller COMM-OP delays. This reduces both the resource over-subscription and the dependence height in the application code, thereby improving COMM-OP delays.

- **Shortcomings:** The principal shortcoming of this methodology is the need to augment the ISA and the core microarchitecture. However, a concise expression of produce and consume semantics may be well worth the incremental core design and verification costs.

Register-Mapped Queues

- **Description:** The instruction and dependence height overhead of produce and consume operations can be further reduced using register-mapped queues [18, 64]. Rather than modifying the ISA by adding produce and consume operations, a certain portion of the register address space is reserved to refer to inter-core queues rather than traditional registers. The microarchitecture is free to implement the underlying operand network [65] using any mechanism.

- **Advantages:** The main benefit is that, since any instruction can deposit its result into a communication queue and any instruction can read an operand from the communication queues, loops will contain fewer instructions and have lower dependence height than the corresponding loops with produce and consume operations. This reduced instruction count and dependence height may prove critical in resource-bound loops.

- **Shortcomings:** On the flip side, this methodology shares its shortcomings with the explicit produce and consume instruction methodology described earlier. It additionally creates increased architectural register pressure since the register address space needs to be split between architectural registers and register-mapped queues. For loops with a large number of live values, register pressure may prove to be a dominant factor in loop performance. Consequently, for loops with a large number of live values, decreased performance due to additional spill and fill code may outweigh the advantages of eliminating produce and consume instructions.

3.2.2 Dedicated Interconnects

Good interconnect design is key to streaming performance. For operations consuming data or synchronization information over the interconnect, any time spent stalled due to interconnect contention adds directly to the COMM-OP delay for that operation. Similarly, for operations producing data or synchronization information, interconnect contention may cause operations to backup in the processor pipeline, adding to the COMM-OP delay of those operations. Streaming data could either share on-chip interconnects with regular data accesses or could use dedicated interconnects. This is an important design consideration given the impact on chip area of routing resources. As Kumar et al. [29] identified, on-chip interconnects occupy large areas, which may force designers to reclaim some area from on-chip caches, often with adverse impact. Ideally, high-frequency streaming support should not require new routing resources, but rather, should be efficiently multiplexed with other requests on existing interconnects. However, depending on the application being run, high contention for the shared interconnect may cause communication operations to stall more often (increasing COMM-OP delays) than on a dedicated interconnect.

3.2.3 Pipelined Interconnects

While the transit delay of the interconnect is not important, the rate at which it can accept new requests directly affects COMM-OP delay. Pipelined interconnects increase the rate at which new requests can be serviced by the interconnect. For a non-pipelined interconnect with an N cycle latency, only one request can be carried by the interconnect every N cycles. However, an M-stage pipelined interconnect can initiate a new request every $\frac{N}{M}$ cycles. This increased throughput reduces contention for the interconnect, thereby reducing COMM-OP delay (and also improving the performance of other operations sharing the interconnect). This disparity between pipelined and non-pipelined interconnect will become more pronounced as larger scale CMPs become more commonplace. Of course, this improved performance does not come for free. Pipelined interconnects are more complex to build than non-pipelined interconnects. Furthermore, memory systems using pipelined interconnects must use coherence protocols more sophisticated than simple snoop-based protocols to deal with multiple inflight requests in the interconnect.

3.2.4 Synchronization

Concurrently executing threads require a synchronization mechanism to determine when it is permissible to read from or write to a queue entry. The time from when a produce (consume) operation begins execution to when it can actually write (read) data to (from) the queue entry is called *synchronization delay*. Since every communication operation has to synchronize before reading or writing data, this delay directly affects the COMM-OP delay. The key to reducing synchronization delay for a communication operation is to ensure that the necessary synchronization information is delivered to its processor core well ahead of the operation's execution. Recall from Section 3.1 that pipelining communication (i.e. communicating using a queue of buffer locations rather than a single buffer location) increases the time between successive synchronizations on a single buffer location. This pipelining offers synchronization mechanisms the necessary slack to deliver synchronization information before it is read by a synchronization operation. The synchronization design options, discussed in this section, vary in the amounts of software and hardware logic, the backing store used for synchronization data and the level of OS support.

Software Techniques

A detailed description of software synchronization was given in Section 3.2.1. The synchronization data consists of an array of full-empty (FE) *condition variables* that are set and reset by produce and consume operations respectively. Since both produce and consume operations modify the same memory locations, with traditional caching mechanisms, synchronization delays will be significantly increased due to frequent cache misses; the first access to a particular queue slot will incur a compulsory miss and subsequent operations will incur coherence misses. (The producer and consumer may benefit from spatial locality if multiple queue entries are located in a single cache line.) This latency will be directly observed by both the producer and consumer since each must read the condition variable before being able to proceed. To avoid such penalties, the condition variables for a queue slot should be moved from the core that writes it to the core that reads it well ahead of the read operation. Prefetching and other microarchitectural optimizations like write-forwarding (described below) may be able to mitigate these shortcomings.

Other software queue implementations that track global queue occupancy rather than individual slot occupancy are possible. However, such mechanisms require a coarse-grain lock that guards access to the entire queue data structure. Consequently, produce and consume operations cannot occur simultaneously even if they are accessing separate portions of the queue. Such implementations will incur costly synchronization delays, since significant contention for the queue lock will exist and cache lines that store the queue occupancy must ping-pong between the producing and consuming cores. The ping-ponging can be minimized by using lazy pointer updates, valid bits and sense reversal, as described by Mukherjee et al. [37].

Hardware Techniques

Just as in software techniques, the key to low synchronization delay is to ensure that synchronization data is maintained as close as possible to the processor core in which it is *read*. For example, maintaining full-empty (FE) bits close to the consumer core may reduce consume synchronization delay, but will increase produce synchronization delay by forcing produce operations to go all the way to the consumer core to read FE bits. Keeping the bits in a centralized location will affect both produce and consume synchronization delays. Instead, if the FE bits are replicated and one copy is maintained at the producer and consumer cores, low produce *and* consume synchronization delays can be achieved. In such a replicated setup, the two copies may be out of step with each other due to delays in propagation of updates from one core to another. However, such delays will not affect correctness (since the out-of-date information is conservative), nor will it affect performance provided there are enough empty (full) queue slots to write (read) data to (from).

Quite a few implementations for such mechanisms exist, and they are often influenced by the choice of queue backing store. For example, StreamLine [7] uses distributed occupancy counters to track memory accesses to special stream pages. The synchronization array (to be presented in Section 3.3), for instance, maintains distributed head and tail pointers to a dedicated circular buffer. By using dedicated synchronization storage specialized for streaming communication, these techniques avoid problems that stem from using the generic memory subsystem. However, in these schemes, the additional hardware synchronization state has to be added to the OS context and needs to be saved and restored on context switches. Additionally, special ISA and microarchitectural extensions and/or OS support may be needed to identify queue read/write operations to the synchronization hardware.

3.2.5 Queue Backing Store

The time from when a consume operation requests data from the backing store to when it receives the data contributes directly to COMM-OP delay. Just as with synchronization, this delay can be minimized by ensuring that data is stored as close as possible to the processor core that will consume the data and by ensuring that the backing store is not oversubscribed. Conversely, adding new, dedicated backing stores to a CMP design increases both the amount of die area dedicated to streaming communication and the amount of OS support required for context switches and virtualization. This section will discuss various design choices for the queue backing store in the context of this trade-off. Note, while this section discusses design options for queue data storage, the issues and mechanisms described here apply equally to synchronization storage.

Shared Memory Store

The memory subsystem serves as a natural store for data being communicated between threads. Architectures and operating systems already provide mechanisms to share data through memory, and most memory systems are equipped with caches to buffer data close to a processor core reducing access time. Streaming communication, however, does not exhibit the same locality as traditional memory accesses. Consequently, designers must decide how streaming accesses should interact with the traditional memory hierarchy. Since streaming accesses exhibit poor temporal locality (producing and consuming threads stride across the queue, rather than access the same element multiple times), certain caches in the hierarchy may want to avoid caching streaming data since caching streaming accesses may lead to eviction of useful data.

While there is poor temporal locality within a core, writes to queue locations are soon followed by reads to the same location by other cores. Consequently, caching lines for queue storage in private caches increases coherence traffic between cores producing and consuming values. Worse still, the delay introduced by these coherence requests contribute directly to COMM-OP delay since the request is demand initiated by the consume operation. Unfortunately, avoiding caching in private caches requires that every produce or consume operation access a shared level of the memory hierarchy creating contention for its ports. Since access times to centralized caches are already typically quite large, forcing all produce and consume operations to contend for few ports will likely lead to large COMM-OP delay in addition to affecting normal memory accesses.

Streaming accesses do, however, exhibit spatial locality. Streaming produce and consume operations stride across the queue data. Consequently, caching queue storage in a core's private cache can reduce COMM-OP delay. For a consumer thread, the first access to a line will incur a large access delay, but successive consumes will be cache hits. Furthermore, if there is sufficient decoupling between a producer thread and consumer thread (*i.e.* they are writing to and reading from distinct cache lines), then coherence traffic occurs at the cache line granularity rather than for each produce and consume operation.

Unfortunately, in situations with little or no decoupling, false cache-line sharing occurs since the producer and consumer threads will be accessing nearby queue entries that fall in a single cache-line. This false sharing can create significant coherence traffic and signifi-

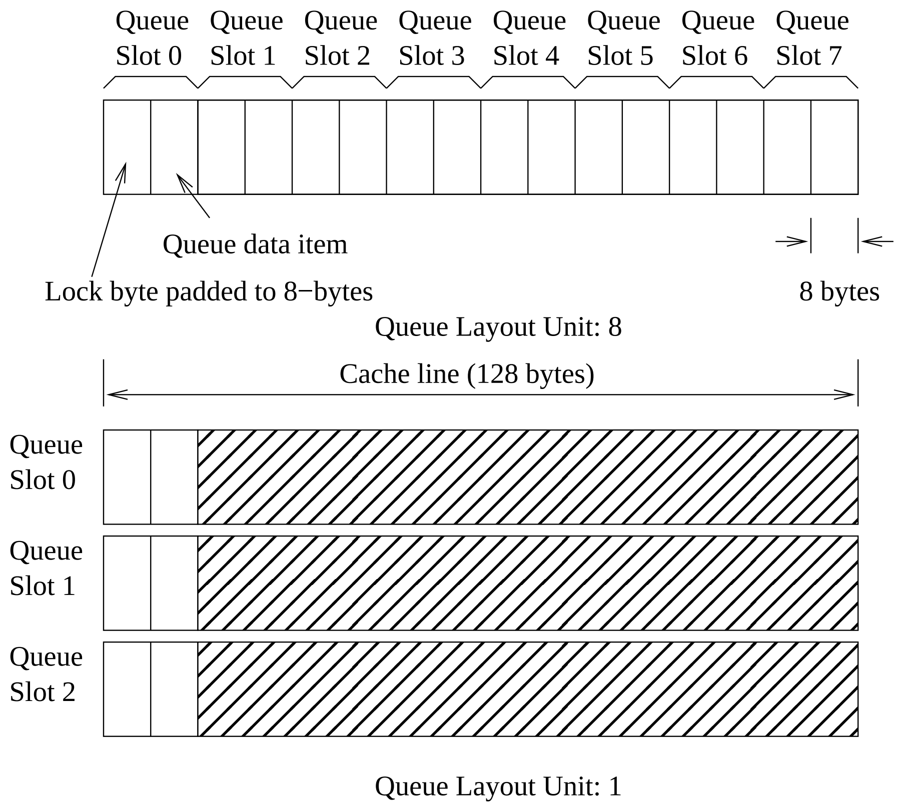


Figure 3.5: Two queue layouts for memory backing stores.

cantly increase COMM-OP delay. Different cache-line layouts can mitigate this problem at the expense of wasted space in the caches. Two possible layouts are shown in Figure 3.5. In the figure, synchronization and queue data are co-located to improve locality of accesses. The layout at the top of the figure places 8 queue entries on a single cache line and can suffer from false sharing. The queue layout on the bottom of the figure, conversely, pads the size of each queue entry so that there is only entry per cache line. By construction, this layout will not experience any false sharing, but wastes large portions of the cache. The layout can be made as dense or sparse as desired. The number of queue entries per cache line is referred to as the *queue layout unit* (QLU).

Prefetching and Write-forwarding. Typically, consume operations (at least ones accessing the first queue entry on a cache line) miss in the local cache and have to fetch data from a remote cache. Such remote data fetches increase the consume COMM-OP delay compared to a local cache hit. In order to bring down this latency, two mechanisms have been proposed in the literature - remote prefetching and write-forwarding. In remote prefetching, the consumer thread issues prefetch instructions before it actually needs the data and tries to overlap the remote data fetch latency with other useful work. The consumer, however, must determine when to issue the prefetch, as overly eager prefetchers may prematurely steal cache lines from the producer's cache. This may cause the producer to slow down appreciably. The second technique, write-forwarding [1, 32, 44, 48], addresses this problem by making the producer thread forward shared cache lines to the consumer's cache *after* it is done producing its data. This way, the timing of inter-core data transfer can be optimized so that neither thread suffers any unnecessary slowdown. Write-forwarding could either be implemented with special completers on store instructions or in processors (e.g. MIPS) that support software-installable TLBs, the OS could mark certain pages as "streaming" and the memory subsystem could be modified to appropriately deal with accesses to such pages. Other mechanisms have been proposed to eagerly transfer lines from a producer's cache to a consumer's cache [47], however, the short time spans between lock access and data access present in high-frequency streaming codes makes them inappropriate for this domain.

Next, a streaming-specific optimization to standard write-forwarding schemes is presented. Stream instructions exhibit strong locality when accessing a cache line since accesses are made to consecutive stream locations. The optimization ensures that this spatial locality is not damaged by write-forwarding. Rather than forwarding the cache-line after each queue entry is filled, the cache controller forwards a line after N queue entries on the line are filled. Typically, the parameter N is set equal to the QLU so that a line is forwarded only after all queue entries on the line are filled. The implementation cost within the cache controller is minimal since it need only be parameterized with the value N and the size of each queue entry. Since accesses to successive queue entries will occur in order, accesses to certain regions of each line will initiate write-forwarding. This optimization will help reduce the average COMM-OP delay by ensuring the maximum number of cache hits possible to a line before forwarding it. For example, for a layout that has only one queue slot per cache line (QLU 1), *N* is equal to 1. However, for a denser layout scheme that packs 8 data items onto a cache line (QLU 8), forwarding will be done on detecting a write to the last 8th segment (N=1/8th) of the cache line and so on. If the queue slots are equally spaced out on a cache line, WFF is equal to 1/QLU.

Dedicated Store

A backing store implemented with dedicated hardware is another possibility. Having a dedicated backing store is advantageous as it does not pollute the memory subsystem with short-lived streaming data. It also helps avoid all streaming related coherence traffic. By ensuring that streaming traffic does not contend with shared memory requests, dedicated stores can prevent normal memory traffic from increasing COMM-OP delays (or streaming operations from decreasing the performance of normal memory operations).

While dedicated stores can potentially improve performance, they do not come without a cost. Dedicated stores consume valuable die area possibly reducing available on-chip cache memory and negatively impacting the performance of non-streaming sections of applications. Additionally, the contents of the dedicated store become part of the OS context for a process. As such, OS and hardware support is necessary to context switch and virtualize these resources.

- **Centralized Dedicated Store:** A centralized dedicated store adds a single streaming-specific memory to the CMP. With this design all cores can share all the storage added to the CMP. Unfortunately, this design suffers from scalability problems as more cores try to access the single structure. Additionally, since the structure is centrally located, for all but the smallest CMPs, the structure will be farther from cores than the local caches. Consequently, the time required to access the store will likely be larger than a local cache hit. This translates to a comparatively larger COMM-OP delay.

- **Distributed Dedicated Store:** Alternatively, streaming-specific memories can be added to each core in the CMP, rather than adding a single central structure. This design scales better than the centralized dedicated store since a single common structure does not need to be accessed by all cores. Additionally, each piece of the distributed store can be located close to the consuming processor reducing the COMM-OP delay for consume operations (recall that a distant backing store does not increase the COMM-OP delay for produce operations). Unfortunately, this design prevents cores from sharing the added storage. Consequently, more die area may be consumed for the dedicated store.

Network Backed Queues

Intermediate nodes in an on-chip network, often buffer data to implement pipelined interconnects. By preserving data ordering they act as effective FIFOs [65]. They inherit all the advantages of dedicated stores. Their distributed nature also makes them scalable. Unfortunately, the amount of decoupling available to the executing threads is directly proportional to the physical separation of their cores on the chip. The larger the separation the more the available decoupling. Relying solely on this storage can affect performance as threads executing on nearby cores will not get sufficient decoupling to tolerate variable latency stalls in the individual threads. Additionally, when network buffers are the sole carriers of inter-thread architectural state, the OS overhead for context switches becomes more pronounced. While switching out a consumer thread, the OS has to check network buffers along the paths from every producer to this consumer before doing the switch. Alternatively, every time data arrives at a node for a thread that has been switched out, an interrupt could be triggered to make the OS append the incoming data to the swapped out context state. The reason why conventional memory networks do not have this problem is because in network-backed queues the network holds the sole copy of the inter-thread data and thus any in-flight data must be explicitly saved and restored. However, for conventional memory networks, there is *always* a safe copy of the data somewhere in the memory subsystem, so that, even if a thread is context-switched, it can perfectly replay its computation by re-requesting the data from memory.

3.3 The Synchronization Array

The synchronization array design draws upon the arguments made in the design space description and optimizes on all design axes to achieve high performance. In particular, the synchronization array design uses special `produce` and `consume` instructions to reduce instruction overhead, uses dedicated backing store to avoid contention with memory accesses, uses distributed hardware synchronization and uses a dedicated network to carry inter-thread queue operand traffic.

This section expands upon the original discussion of the synchronization array, presented in [51]. It first motivates the design of the synchronization array and describes its functionality and operation. Then, it discusses synchronization array scalability issues and finally shows how the synchronization array can be integrated with a commodity processor pipeline such as that of the Itanium® 2.

3.3.1 Operation

The instruction set architecture (ISA) abstraction of the synchronization array is a set of blocking queues accessed via `produce` and `consume` instructions. The `produce` instruction takes an immediate dependence number and a register identifier as operands. The value in the register is enqueued in the virtual queue identified by the dependence number. The `consume` instruction dequeues data in a similar fashion. The compiler introduces `produce` and `consume` instructions at the source and destination of any dependence going from one thread partition to another. While ordinary `produce` and `consume` instructions suffice to communicate register and control dependences from one thread to another, communicating memory dependences require special memory fence semantics on the `produce` and `consume` instructions to guarantee correct memory consistency. Variants of `produce` and `consume` instructions - `produce.rel` and `consume.acq` instructions respectively - are used to communicate memory dependences. A `produce.rel` instruction has *release* semantics with respect to memory stores. It will execute only after all prior stores have committed (i.e. made their effects visible) in its thread. On the other hand, a `consume.acq` has *acquire* semantics with respect to memory loads. An uncommitted `consume.acq` instruction will prevent any subsequent memory load instructions in that thread from issuing. In order to correctly synchronize a memory dependence between a store in one thread and a load in another thread, the compiler inserts a `produce.rel` instruction (to some queue) *after* the store in the first thread and a `consume.acq` instruction (to the same queue) *before* the load in the second thread. This will guarantee correct memory semantics.

Correct synchronization and communication between a `produce-consume` pair involves two main operations - *which queue location to access* and *when to access this location*. In the past, FIFO channels have been used to implement inter-core queues [58]. FIFO channels are organized as an array of latches and data is shifted from one latch to the next in consecutive cycles. In such implementations, determining which queue location to access is straightforward. The producer (consumer) thread writes to (reads from) the latch at the producing (consuming) end of the FIFO channel. The second question of when to access this location is also resolved by looking at the fullness/emptiness of the end latch. While the implementation of the basic operations is straightforward with FIFOs, such implementations suffer from two limitations.

- FIFOs **serialize all communication** through a queue at the producer and consumer ends preventing out-of-order issue processors from speculatively writing to and reading from queue locations. While speculative writes are not useful in practice[1], speculative out-of-order reads from a queue is key to good performance on out-of-order issue processors with large instruction windows.

- FIFO channels are **not scalable**. Since the `produce` and `consume` instructions are exposed at the ISA level, any thread on any core can execute these instructions. Thus, in order to support dedicated queues among any two threads, a FIFO implementation necessarily has to have FIFO channels running between every pair of cores or should have logic to efficiently multiplex and demultiplex data coming from and going to various cores.

The synchronization array microarchitecture is designed to address these concerns. Unlike FIFO channels, the synchronization array separates the determination of which queue location to access and when to do the access into two different entities.

[1]They are useful only at times when a queue is empty. However, if the queue is empty most of the time, then it calls for a redistribution of work amongst communicating threads.

The task of determining which queue location to access is handled by a special synchronization array rename logic. The compiler and rename logic together ensure that both sides of a communicating pair of `produce` and `consume` instructions are mapped to the exact same queue slot or dependence array slot. For each dependence number d, the renamer maintains a count of the number of outstanding mappings and a next mapping field that indicates which queue slot the next `produce` or `consume` instruction to dependence number d will be mapped to. The next mapping for d is incremented (with wrap around) after a successful rename operation.

Through careful code generation, the compiler ensures that dynamically there is a one-to-one correspondence between `produce` and `consume` instructions across communicating threads for each dependence number. This guarantee combined with the fact that the `produce` and `consume` instructions flow through the rename stage in program order in their respective cores, ensures that both instructions of every `produce-consume` dependence-pair are renamed to the same unique dependence array slot. The renamer also reclaims dependence array slots after a successful synchronization. This is very similar to the rename logic in modern out-of-order processors. However, free-list maintenance is simplified in the SA rename logic, since there is only one unique `produce` and `consume` instruction per allocated slot.

Queue data storage and the full/empty bit for a queue slot is maintained in the synchronization array itself. Figure 3.6 shows the structure of the SA in more detail. The state machine in each dependence array entry, shown in Figure 3.7, ensures that the `consume` instruction reads the data value only *after* the corresponding `produce` instruction is done producing. When the `produce` instruction executes, it writes the data into the data field, writes the producer tag field with its own tag and updates other architectural state as shown in Figure 3.7. When a `consume` is ready to execute, the SA checks to see if data is available for read (i.e. checks to see if the `Full` bit is set). If it is, it immediately returns the data and frees the entry. If not, the SA remembers the tag of the `consume` instruction

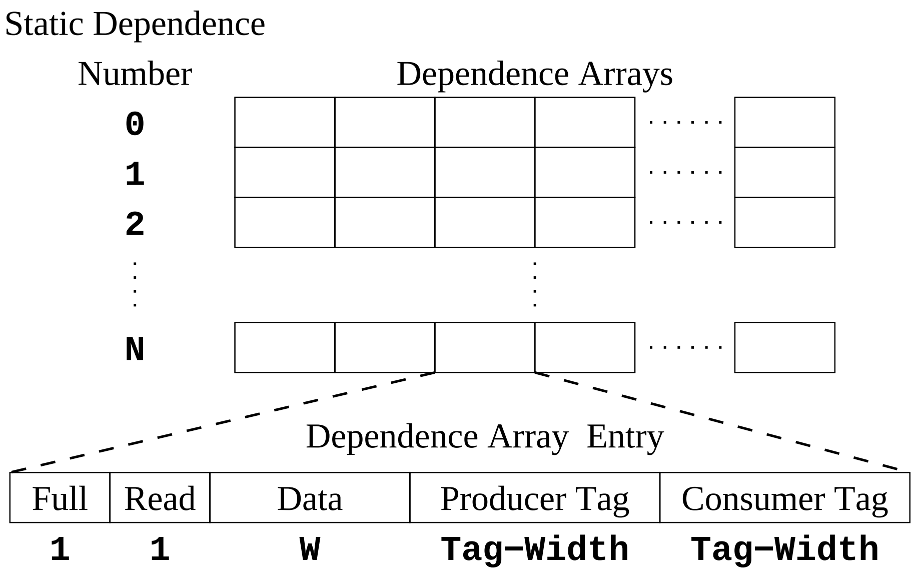


Figure 3.6: Synchronization Array structure

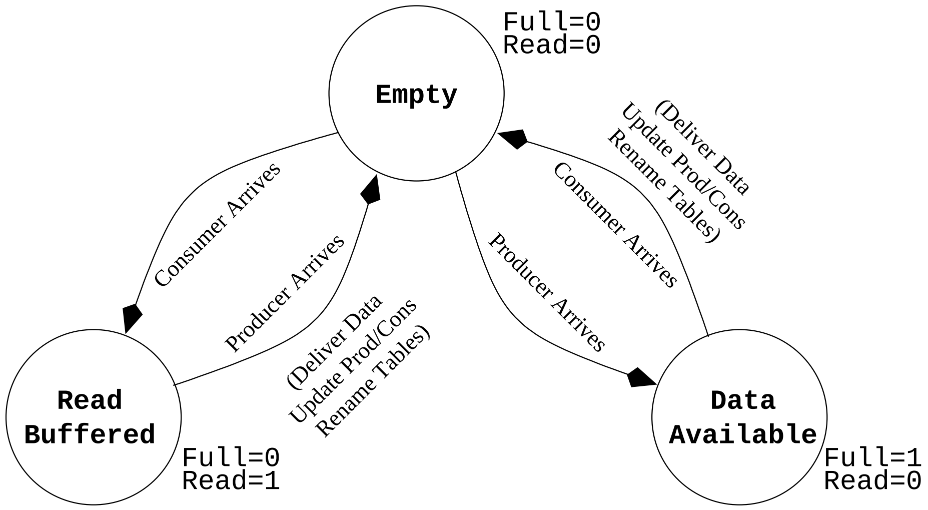


Figure 3.7: Synchronization Array State Machine

in the consumer tag field. In such cases, `consume` instructions take multiple cycles and any dependent instructions will stall the corresponding pipeline. Overall, there is a def-use latency of at least one cycle between a `consume` instruction and any dependent instructions. When the corresponding `produce` instruction arrives, the SA delivers the data to the writeback stage of the processor along with the consumer tag. This simple protocol ensures correct synchronization between two communicating instructions.

Since queues have space enough to buffer exactly one `produce` and one `consume` instruction per slot, the renamer stalls the pipeline whenever the outstanding mappings count for any dependence number d becomes equal to the number of slots per queue. The SA notifies the producer and consumer core with the tag of the `produce` and `consume` instructions, respectively, whenever an array slot reverts to the `Empty` state. This allows the renamer to reclaim dependence array slots and mark them as available for future allocations. In this way, the synchronization array and rename logic preserve the illusion of a queue. Once past the rename stage, the instructions can execute out-of-order with respect to each other. This removes the serialization bottleneck seen with FIFOs. Scalability concerns will be addressed in Section 3.3.3.

While the decentralized hardware avoids lookups into the SA, in the absence of global broadcasts of issue and completion activity from all other cores, the rename logic of a core can only deliver accurate information about slots allocated to and accessed by dependence numbers for which this core is the exclusive producer or consumer. As a result, for a given dependence number, there must be a unique core that produces data and a unique core that consumes it. The designated producer-consumer pair of cores for a particular dependence number is allowed to change only after all issued `produce` and `consume` instructions corresponding to that dependence number have retired. In order to support multiple dependencies between the two threads, the SA is organized as an array of dependence arrays as shown in Figure 3.6. This provides the compiler with a lot of scheduling flexibility with respect to `produce` and `consume` instructions in each thread. For example, by allocating a unique dependence number to every true inter-thread dependence, `produce` and `consume` instructions in a program control block can be moved to any location within that block. The compiler can also constrain SA instruction movement by assigning them the same dependence number.

3.3.2 Handling Control Speculation

All modern processors use control speculation as a standard mode of operation to achieve high instruction throughput, in the presence of frequent branching instructions. Support for control speculation in a processor pipeline demands that any microarchitectural element, whose state is essential for program correctness, be able to recover from misspeculation events. There are two basic issues that arise when handling control speculation in a synchronization array implementation. First, can `produce` and `consume` instructions be *renamed* speculatively? Second, what does it mean for `produce` and `consume` instructions to *execute* speculatively?

Renaming

The rename logic described above is quite similar to register renaming in traditional out-of-order processors. Thus, similar recovery mechanisms can be used to recover the synchronization array renamer, upon misspeculation, to a correct prior state. In this dissertation, an Alpha 21264 [11, 25] style checkpointing scheme is used to save the renamer's state upon encountering a branch instruction and restoring the renamer to the corresponding checkpointed state on a branch misspeculation.

Synchronization array access

`Consume` instructions, like load instructions, are at the head of dependence chains and will greatly benefit from speculative synchronization array access, instead of having to wait until they commit, to access the synchronization array. Speculative read access can drastically lower the average def-to-use latency between a `consume` instruction and an instruction dependent on it. The only thing to watch out for in handling speculative consumes is that the synchronization array should not reclaim a slot on a speculative access and instead should wait for the core to signal to it that the instruction has successfully *committed* and that the slot can be reclaimed. While conceptually, `consume` instructions are in charge of reading the synchronization array as well as initiating slot reclamation, the two actions will have to be performed at different time steps during the lifetime of a `consume` instruction, in order to support speculative `consume` instructions. `Produce` instructions, on the other hand, being at the bottom of dependence chains, do not really benefit from speculative write access. Further, if another thread consumed a value written by a control speculative `produce` instruction from one thread, it causes the speculation to "leak" from the producing core to the consuming core. Thus, any misspeculation recovery should be a multi-core recovery mechanism and is expensive. Besides, as mentioned earlier in footnote 1, speculative `produce` operations are useful only when the queues are empty most of the time, a situation the compiler/partitioner should strive to avoid.

3.3.3 Performance Scalability

The above description simply places the synchronization array between two cores and takes advantage of the physical proximity to enable fast core-to-synchronization array communication. As the number of on-chip cores increase, depending on the location of the synchronization array, the access latencies may vary, perhaps even non-uniformly across cores.

Increased latency will affect the transfer delay of four different types of signals from/to the synchronization array. Given a producer and a consumer core, the four different types of signals are: a request signal from the producer core to the SA, a consume signal from the SA to the consumer core, a slot reclamation signal from the consumer core to the SA to enable it to reclaim queue items after successful synchronization and a similar "synchronization over" signal from the SA to the producer core to inform it about slot reclamations.

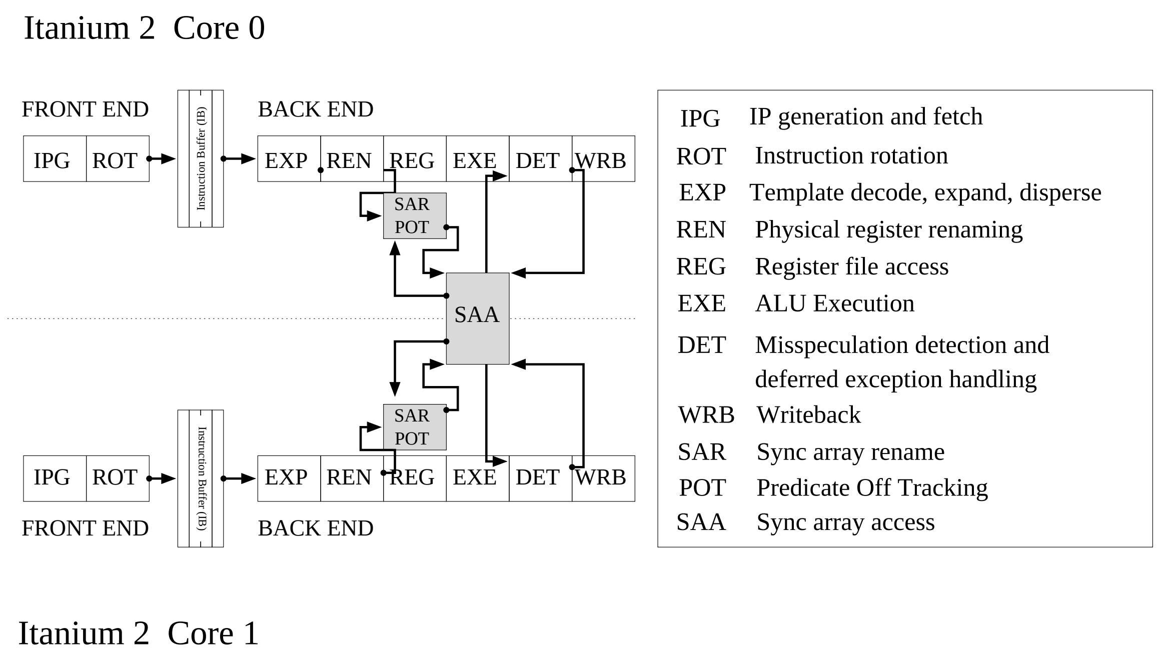
Placing the synchronization array closer to a producing core would enable the former to quickly notify the producing core of queue slot reclamations. Three different organizations are possible for the synchronization array when moving to larger multi-core processors. It can either be located on the producer side, on the consumer side or in some central location. Empirical results indicate that placing the synchronization array on the consumer side offers the best performance. The second best performance is obtained from the organization that locates the synchronization array midway between communication producer and consumer cores. The worst organization is locating the synchronization array on the producer side. The reason for this bias towards locating the synchronization array closer to the consumer side is that, such an organization lowers `consume`-to-use latency which improves the consumer thread's performance. This in turn initiates recovery of slots at a faster rate, thereby improving the producer thread's performance. Moving SA closer to the producer core slows down the consumer which ultimately slows down the overall execution. Experimental evaluation confirms this intuition and indicates that locating the SA storage closer to the consumer core yields better performance than locating it closer to the producer core.

3.3.4 Integrating with the Itanium® 2 pipeline

This section explains how the synchronization array can be easily integrated with a processor pipeline. The Itanium® 2 pipeline is chosen as an example for exposition purposes. This section first presents a brief description of the main pipeline of an Itanium® 2 processor [24] before explaining how synchronization array renaming and access logic is integrated into the processor pipeline.

The Itanium® 2 is an implementation of the Intel IA64 Architecture that can fetch and issue up to 6 instructions per cycle. Its main pipeline has 8 stages, split into a front end and a back end. Figure 3.8 shows the processor's integer pipeline along with the logic and datapaths added for synchronization array integration.

The front-end of the Itanium® 2 pipeline consists of two stages, Instruction Pointer



IPG	IP generation and fetch
ROT	Instruction rotation
EXP	Template decode, expand, disperse
REN	Physical register renaming
REG	Register file access
EXE	ALU Execution
DET	Misspeculation detection and deferred exception handling
WRB	Writeback
SAR	Sync array rename
POT	Predicate Off Tracking
SAA	Sync array access

Figure 3.8: Dual-core CMP configuration with Synchronization Array. The individual cores are Itanium® 2 cores. Shaded pipeline stages indicate extensions to the Itanium® 2 datapath to support synchronization array instructions. The dark arrows indicate new datapaths added. The $SAR/POT \rightarrow SAA$ datapath carries speculative SA accesses. The $DET \rightarrow SAA$ datapath carries non-speculative SA accesses. The $SAA \rightarrow SAR$ signal carries synchronization over notifications from the SA to the cores and the $SAA \rightarrow DET$ datapath carries the queue value read by `consume` instructions from the SA.

Generation (IPG) and Bundle Rotation (ROT), that prepare instructions for execution and feed them into the Instruction Buffer (IB). The back-end fetches instruction groups from the IB and organizes the individual instructions into *issue groups* that are guaranteed to consist of independent instructions (EXP). Structural hazard detection for functional units and register file ports is performed in EXP as well. Register names are then coordinated with the Register Stack Engine and renamed accordingly (REN). The register values are then fetched from the appropriate register file (REG). It is at this stage that the normal in-order pipeline stalls an issue group if all source operands are not ready. Instructions then continue through execution (EXE), exception detection and branch resolution (DET), and finally write their results back to the register file (WRB). Floating point and memory instructions have extra execution stages between DET and WRB.

In the SAR stage, `produce` and `consume` instructions are renamed to the correct SA slots and they proceed to access the synchronization array in the SAA stage. Conceptually, the renaming has to happen at least a cycle before the synchronization array is accessed. While renaming for `produce` instructions can be delayed until past the DET stage, `consume` instructions must be renamed no later than the REG stage since they must access the synchronization array in the EXE stage to provide a 1 cycle `consume`-to-use latency. Renaming cannot be done before the REN stage since the Itanium® 2 pipeline explodes bundles into instructions only in the EXP stage. Thus, renaming can be done either in parallel with REG stage or in parallel with REN stage without any performance penalty. This dissertation models the synchronization array renaming logic in the REG stage as shown in Figure 3.8. However, detailed timing measurements and critical path analysis of the processor pipeline will be needed to determine which stage such logic should go into.

A `produce` instruction is considered *done* after it writes to its SA slot. A `consume` instruction returning from the SA writes back its value to the register file in the WRB stage, even though the read value is available on the bypass logic after the EXE stage. The consuming core informs the SA whenever a `consume` instruction commits. Upon being notified of consume commits, the SA informs the SAR logic of both the producing and consuming cores so that both cores can reclaim the corresponding SA slot.

Supporting Predication

Supporting predicated produce and consume instructions is important for several reasons. Predication [3, 36] allows compilers to eliminate hard-to-predict branches and compile two or more control flow paths into a straight line code sequence. Dynamically, predication eliminates pipeline flushes caused by such branches. Statically, it provides compiler with increased scheduling flexibility. In the IA64 architecture, predication and rotating registers can be combined to achieve very tight software-pipelined [30, 52] loops.

However, predication support for produce and consume instructions presents challenges for the design and implementation of the rename logic and integrating it with the main processor pipeline without affecting processor cycle time. Recall that the compiler guarantees that dynamically, exactly one produce instruction writes to a slot and that exactly one consume instruction reads that value from that slot. Allowing predicated produce and consume instructions means that the compiler can generate multiple static produce or consume instructions for the same queue, but predicate them so that dynamically there is a one-to-one correspondence between produce and consume instructions.

Execution for predicated-off instructions is short-circuited and they do not access the synchronization array. On the other hand, true produce and consume instructions, i.e. unpredicated or predicated-on instructions, access the synchronization array and update state as described above. In the current design, the SAR logic hands out consecutive mappings from a modulo-N space, where N is the number of queue or dependence array entries, on both the producer and consumer sides. Suppose a sequence of six predicated produce instructions to queue 1 is presented to the SAR logic. Assume that only the sixth instruction is a true produce instruction and the five preceding produce instructions are predicated off. Without special support for predication, the first true produce instruction will be renamed to queue slot 5, whereas it should have been renamed to queue slot 0 since all prior `produce` instructions to queue 1 were predicated off.

Conceptually, the rename logic should also be made aware of predicated off instructions so that it can ensure appropriate dynamic one-to-one correspondence between true `produce` and `consume` instructions. However, predicate values are known only in the REG stage. Doing conditional renaming based on the predicate value in REG stage will, in effect, serialize the SAR logic behind the baseline REG logic. This is so, because the SAR logic will now have to wait for scoreboarding and register file access before proceeding with synchronization array renaming. Such a design can greatly increase the critical path of the processor pipeline and affect the cycle time. Optionally, if a latch separates the SAR logic from the REG stage, then the bypass logic will be affected, since back-to-back operand bypass will no longer be possible for source operands coming from the synchronization array and will consequently degrade performance. What is needed is a mechanism that will not affect the cycle time nor instruction throughput.

To guarantee correct queue semantics in the presence of predication without impacting performance or processor cycle time, extra logic, called the predicate-off tracker (POT), is added to the REG stage alongside the SAR logic. The POT logic maintains d counters, where d is the total number of queues or dependence arrays. Each counter keeps track of the number of predicated off `produce` or `consume` instructions to the corresponding queue. Upon branch misspeculation, each predicate-off counter will have to be restored along with the restoration of the SA renamer's state. Since the SAR logic hands out new mappings unconditionally to all `produce` and `consume` instructions, for a given `produce` or `consume` instruction, subtracting the predicate-off count from its mapping will yield the correct physical queue slot number that ought to be accessed by the instruction. The POT logic tags each `produce` or `consume` instruction, upon entry into the REG stage, with the appropriate physical slot number by subtracting the current predicate-off counter value of the queue being accessed from the SAR mapping of that instruction.

For example, suppose a sequence of six predicated `produce` instructions are presented to the SAR logic. And also suppose that all except the sixth instruction are predicated off. The sixth instruction is given a mapping of 5 by SAR, the five `produce` instructions before it are all predicated off which causes the predicate-off count to be raised to 5. And when the sixth `produce` instruction passes through the REG stage, it gets tagged with 0 as the physical slot number (physical slot number is mapping minus predicate-off count modulo queue size). The subtractor logic will happen in parallel with scoreboarding and register file access of the REG stage. So, it will not prolong the cycle time for the REG stage. Note, the subtraction operation is also performed in the modulo-N space, where N is the number of entries in a queue. Since predicate values are known by the end of the REG stage, the POT logic updates the counters corresponding to any predicated off `produce` or `consume` instructions at the end of the REG stage, as instructions make their way into the REG-EXE stage latch. Also, note that the subtractor logic need only be $\lceil log_2 N \rceil$ bits wide and therefore, is unlikely to affect the processor's critical path timing for reasonable queue sizes (32, 64, etc.).

To summarize, the POT logic tags communication instructions with their absolute physical slot number by performing a simple subtraction operation during the cycle, in parallel with regular REG stage operations and it updates the POT counters themselves as instructions enter the REG-EXE latch. Note that the cycle time arguments presented here are based on intuition about the relative timing of the various pieces of logic. Low-level timing measurements have to be carried out before any of these can be incorporated into real designs.

Implementing Acquire and Release Semantics

As mentioned before, the `produce.rel` and `consume.acq` instructions are used to guarantee correct semantics when synchronizing for memory dependences. Scoreboarding for both types of instructions is done in the REG stage of the pipeline. The REG stage maintains two counters, one to track the number of outstanding store instructions and the other to track the number of outstanding `consume.acq` instructions.

The store count is incremented when a true store instruction (i.e. not predicated off) is issued. If a true store is canceled due to misspeculation, the store count is decremented. Similarly, if a true store commits in the memory subsystem, the REG stage is informed and the outstanding store count is decremented. A `produce.rel` instruction cannot issue from REG unless the outstanding store count is zero. The `consume.acq` count is also incremented when a true `consume.acq` instruction (i.e. not predicated off) is issued. The count is decremented if a true `consume.acq` instruction is canceled or if a true `consume.acq` instruction commits. A load instruction cannot issue so long as the outstanding `consume.acq` count is not zero. With these two counters and relevant datapaths to update them, produce release and consume acquire semantics can be added easily to the Itanium® 2 microarchitecture.

3.4 The Snoop-Based Synchronization technique

This is a low-cost design that attempts to combine the positive aspects of a dedicated hardware technique like the synchronization array with the low hardware and OS cost advantages of shared-memory based software-only techniques. In effect, it provides an efficient message passing implementation atop a shared-memory CMP. Others have proposed mechanisms to implement efficient message passing in shared-memory multiprocessors [7, 15, 21, 28, 48]. However, message setup overhead in these designs make them in appropriate for PMT applications with high-frequency streaming. It improves Mukherjee et al. [37]'s cachable queue design in that snoop-based synchronization includes logic to update head and tail pointers (kept in both the producer and the consumer cores) based on the read/write messages snooped from the bus. This avoids unnecessary bus traffic for occupancy pointer updates. By piggybacking queue pointer-updates on cache line write forwarding, snoop-based synchronization is able to achieve efficient data transfers as well as queue pointer updates.

In the snoop-based synchronization technique, `produce` and `consume` instructions are dynamically renamed to unique memory addresses. Special microarchitectural stream address logic assigns consecutive stream addresses (in a modulo space) for all accesses to a queue. Per-queue hardware occupancy counters maintained at the L2 controller provide synchronization. Although the proposed implementation assumes `produce` and `consume` instructions, the same behavior can be obtained with conventional store and load instructions as well. One possible implementation is to for the compiler to make stores and loads to a particular queue to always generate the same unique effective address. Prior to this, during program loading time, the TLB must be initialized with certain virtual memory areas as being “streaming pages”. Dynamically, when a store or a load to a streaming page is encountered, it is handled differently from conventional stores and loads. Streaming stores and loads go through the stream address logic and special synchronization before accessing the correct memory location. This design has not been evaluated, but is mentioned here as another potential design option.

A producer (consumer) core updates its occupancy counters after successfully executing a `produce` (`consume`) instruction or after snooping occupancy updates from the bus. A `produce` (`consume`) instruction is allowed to access the L2 cache if and only if the occupancy counter corresponding to its stream does not indicate a full (empty) queue. Write-forward messages are used by the consumer core to update its occupancy counters. When the last queue item (“last” depends on the queue layout) from a given streaming line is read by a `consume` instruction, the consumer core sends out a message on the bus to inform the producer’s occupancy-tracker of how many `consume` instructions were serviced from that particular line. When a streaming cache line is evicted from an L2 cache, then the cache once again puts out on the bus the number of queue items produced into (or consumed from) the line for its counterpart to update its occupancy counters. When the producer thread wraps around, it is stalled until all queue items from the corresponding line have been consumed by the consumer, to avoid damaging spatial locality in the consumer. Finally, since no write-forward messages will be sent when a stream terminates midway through a cache line, `consume` requests initiate an L3 access after a time-out to elicit a writeback from the producer core, to obtain the remaining queue items and avoid deadlock. Although the proposed implementation is a bus-based one, through simple modifications to the occupancy update protocol, this can be adapted to network-based interconnects of future CMPs.

In order to cope with increased inter-thread streaming traffic due to multiple threads or more complex communication patterns (for example, scatter-gather), the memory network arbiter can be modified to favor application memory requests over inter-thread operand traffic (a simple way to do this is to just look at the memory area being accessed). While application memory performance remains unaffected, pipelined inter-thread communication helps tolerate delays due to increased contention.

In the case where the producer thread races ahead of the consumer thread, has wrapped around and is trying to write to the same cache line that the consumer is still trying to read from, it will steal the cache line from the consumer to do the write, which could in turn adversely impact the performance of an already slow consumer. The solution is to stall all `produce` requests to a cache line until all queue items have been consumed from that line in the remote core. So basically, with a queue layout unit of 8, a 32-sized queue takes up four cache lines. To start with, all cache lines are empty and the producer can immediately produce 32 data items, 8 each to lines 0, 1, 2, and 3. However, in order to continue producing further, the producer has to wait until the occupancy counter for the queue reaches 24, which indicates that the consumer is done consuming all data items in line 0. The producer can then go ahead and write to line 0 and so on. The queue layout unit (QLU) should be chosen carefully by taking into account these considerations as well as the overall memory system performance desired, since very low QLUs mean queues take up more cache lines and hence more lines have to be transferred between caches. Relying solely on the write-forward messages for occupancy counter updates has the risk of leading to deadlocks if stream termination is not dealt with carefully. A time-out based forced-writeback mechanism is used to deal with such situations. No write-forwarding happens if a stream terminates midway through a cache line, short of the segment of the line corresponding to the write-forward fraction. Consequently, the consumer side will never update its occupancy counters. In such situations, in order to ensure correctness and forward progress, the waiting requests on the consumer side will time out and try to go to the L3. This will elicit a writeback from the producer core (along with the occupancy), which the consumer core then uses to update its occupancy counters and satisfy the pending `consume` requests.

3.5 Summary

This chapter identified four orthogonal aspects of any streaming communication design and discussed the pros and cons of several design points for each of these basic components in a detailed design space characterization. Based on this characterization, this chapter then presented two techniques - the synchronization array and the snoop-based synchronization technique. While the synchronization array is optimized for performance at a steep hardware and OS cost, the snoop-based synchronization design minimizes hardware and OS cost by using shared memory based queues.

Latency tolerance and scalability of DSWP are first evaluated with the best-performing communication support, the synchronization array, in Chapter 5. Following that, a quantitative evaluation of various communication designs is presented in Chapter 6.

Chapter 4

Evaluation Methodology

Before taking a look at the performance evaluation in Chapters 5 and 6, it is important to understand the experimental environment. This chapter presents details about the applications, compiler, simulator and performance analysis methodology used in this dissertation.

4.1 Benchmarks and Tools

All quantitative evaluation presented in this dissertation used code produced by the VELOCITY [67] compiler framework. A diverse set of applications drawn from several publicly available benchmark suites is used for evaluating the various techniques. The benchmarks studied include `art`, `mcf`, `equake`, `ammp`, and `bzip2` from the SPEC-CPU2000 benchmark suite, `epicdec` and `adpcmdec` from the MediaBench [31] suite, `mst`, `treeadd`, `em3d`, `perimeter`, and `bh` from the Olden suite, `ks` from the Pointer-Intensive benchmark suite, and the Unix utility `wc`. A key loop in each of these applications is targeted for DSWP. A short description of each application and details about the loop chosen from each benchmark are provided in Table 4.1. All the Olden benchmarks except for `em3d`, which were originally recursive implementations, were rewritten to be iterative procedures, since the DSWP compiler can handle only regular loops at the time of this writing. The different code versions for all benchmarks were generated with all the

Benchmark	Function	% Exec. Time	Benchmark Description
`mst`	`BlueRule`	100%	Minimal spanning tree
`treeadd`	`TreeAdd`	100%	Binary tree addition
`perimeter`	`perimeter`	100%	Quad tree addition
`bh`	`walksub`	100%	Barnes-Hut N-body simulation
`em3d`	`traverse_nodes`	100%	3D electromagnetic problem solver
`wc`	`cnt`	100%	Word count utility
`ks`	`FindMaxGpAndSwap`	99%	Kernighan-Lin graph partitioning
`adpcmdec`	`adpcm_decoder`	98%	Adaptive differential PCM sound decoder
`equake`	`smvp`	68%	Earthquake simulation
`ammp`	`mm_fv_update_nonbon`	57%	Molecular mechanics simulation
`mcf`	`refresh_potential`	30%	Combinatorial optimization
`epicdec`	`read_and_huffman_decode`	21%	Image decoder using wavelet transforms and Huffman tree based compression
`art`	`match`	20%	Neural networks based image recognition
`bzip2`	`getAndMoveToFrontDecode`	17%	Burrows-Wheeler compression

Table 4.1: Loop Information

classical optimizations turned on. In all cases, instruction scheduling for control blocks was done both before and after register allocation.

The generated codes were then run on a cycle-accurate multi-core performance simulator constructed with the Liberty Simulation Environment [70, 71]. The multi-core simulator was derived from a validated core model, which was shown to be within 6% of the performance of native Itanium® 2 hardware [42]. This framework does not model a hardware or a software thread scheduler. Therefore, an N-core configuration can run at most N threads.

Details of the baseline in-order model are given in Table 4.2. The out-of-order processor model is identical to the in-order model for the most part. The REG stage of the in-order model is augmented with a register update unit (RUU) [60] to allow out-of-order scheduling. The out-of-order model has an additional pipeline stage ahead of the REG stage for RUU dispatch. The baseline out-of-order model had 256 RUU entries, and twice the number of issue ports, cache ports, and cache lines compared to the baseline in-order issue model. The baseline out-of-order model's parameters are given in Table 4.3. Note, the baseline out-of-order simulator was deliberately configured to model a very aggressive, albeit optimistic and perhaps impractical OOO design. As will be seen in Chapter 5, DSWP on less aggressive, more practical designs, is able to outperform even such highly aggressive designs.

Core	Functional Units - 6-issue, 6 ALU, 4 Memory, 2 FP, 3 Branch
	Misprediction pipeline - 7 stages
	L1I Cache - 1 cycle, 16 KB, 4-way, 64B lines
	L1D Cache - 1 cycle, 16 KB, 4-way, 64B lines, Write-through
	L2 Cache - 5,7,9 cycles, 256KB, 8-way, 128B lines, Write-back
	Maximum Outstanding Loads - 16
Shared L3 Cache	$>$ 12 cycles, 1.5 MB, 12-way, 128B lines, Write-back
Main Memory latency	141 cycles
Coherence	Snoop-based, write-invalidate protocol
L3 Bus	16-byte, 1-cycle, 3-stage pipelined, split-transaction bus with round robin arbitration

Table 4.2: Baseline Simulator.

Core	Functional Units - 12-issue, 12 ALU, 8 Memory, 4 FP, 6 Branch
	Misprediction pipeline - 8 stages
	RUU Entries - 256
	L1I Cache - 1 cycle, 32 KB, 4-way, 64B lines
	L1D Cache - 1 cycle, 32 KB, 4-way, 64B lines, Write-through
	L2 Cache - 5,7,9 cycles, 512KB, 8-way, 128B lines, Write-back
	Maximum Outstanding Loads - 128
L3, memory and coherence model	Same as in-order configuration (refer Table 4.2)

Table 4.3: Baseline Out-Of-Order Simulator.

Tables 4.2 and 4.3 only provide details of the core model and the memory subsystem. The actual number of cores used varied depending on the experiment performed. This will become obvious from the context.

The raw performance in terms of the effective instructions per cycle (IPC) of the baseline single-threaded code for the above benchmark loops on the in-order and out-of-order simulator models used in this dissertation is given in Figure 4.1. Note, the effective IPC calculation uses only useful application instruction counts and excludes no-ops and predicated-off instructions.

4.2 Performance Measurement

As will be seen in the remainder of the dissertation, several different combinations of code types and microarchitectural features are evaluated in tandem and in isolation. Since aggregate statistics like instructions per cycle (IPC) or clocks per instruction (CPI) will not suffice to capture performance improvements across code changes, execution time has to be used to compare performance over different configurations (across code and microarchitecture changes). However, pure execution time does not yield any insight into the run-time

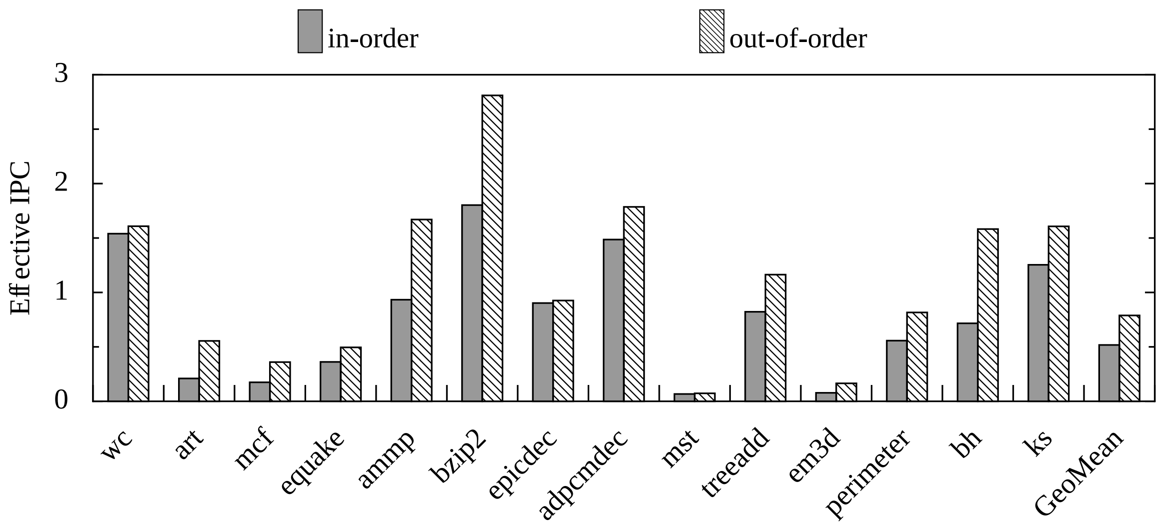


Figure 4.1: Raw performance of single-threaded code on baseline models

behavior of codes and analysis becomes difficult.

To overcome the limitations of plain execution time measurements, this work uses a simple yet powerful *bottleneck analysis* methodology to determine *instruction bottlenecks* as well as *microarchitectural bottlenecks*. By determining the overall percentage contribution of individual instructions to the total execution time, it is possible to identify instruction bottlenecks. This knowledge can then be used to efficiently partition instructions among threads so as to ease the bottlenecks identified. Similarly, identifying microarchitectural bottlenecks, parts of the processor pipeline wherein the most of the critical path time is spent, is equally important, so that the architect can try and avoid any spurious stalls or devise microarchitectural optimizations to reduce stalls.

The analysis works as follows. In the simulator, every instruction flowing through the processor pipeline is annotated with the timestamp of when the instruction leaves a particular pipeline stage. The analysis then aims to attribute every cycle of execution to stalls experienced by the oldest dynamic instruction, program instruction in terms of the number of stall cycles experienced by that instruction in various processor pipeline stages.

The analysis maintains a *last accounted cycle (LAC)* variable, which, as the name suggests, tracks the last execution cycle that has been accounted for. When an instruction commits, the analysis first computes the number of bubbles introduced in each stage of the pipeline. The number of bubbles introduced in a particular stage S_i is computed by subtracting the timestamp of S_i from the timestamp of the previous stage $S_{(i-1)}$. Since any stage is expected to take 1 cycle by default, 1 cycle is subtracted from the difference obtained above to obtain the number of bubbles introduced. Next, the different timestamps accumulated by the committing instruction are processed in pipeline order. For each stage S_i, its timestamp is compared with the *lastAccountedCycle* variable and if found to be greater than *lastAccountedCycle*, the difference between the timestamp and the *lastAccountedCycle* value are new cycles that need to be accounted for somehow. The analysis accounts for these new cycles by scanning the $bubble$ array backwards, starting at the current stage. If the number of bubbles introduced in S_i is greater than the number of new cycles that need to be accounted for, the analysis subtracts new cycles from $bubbles[i]$. Otherwise, the analysis decrements the cycles to be accounted for by $bubbles[i]$, sets $bubbles[i]$ to 0, and moves on to stage $S_{(i-1)}$. This is repeated until the cycles to be accounted for becomes 0. At that point, the analysis advances *lastAccountedCycle* to the timestamp of S_i and is done processing the current instruction.

The non-overlappable stalls determined by the above procedure are organized into a two-dimensional matrix with instructions as rows and pipeline stages as columns. A row represents the non-overlappable stall contribution of an individual instruction to the overall execution time and can help identify instruction bottlenecks. A column represents the non-overlappable time spent in a given pipeline stage during the entire execution and is useful to identify microarchitectural bottlenecks. The underlying philosophy of this analysis approach is to provide the automatic partitioner (in the case of instruction bottlenecks) or the architect (in the case of microarchitectural bottlenecks) feedback about only those stalls that actually end up contributing to the overall execution time and avoid red-flagging instructions or pipeline parts whose stalls are completely overlapped by other, more critical stalls.

4.3 Sampling Methodology

Application of DSWP to a program loop leaves the pre-loop and post-loop code almost untouched[1]. As a result, their performance is the same across all single-threaded and multi-threaded executions for a given hardware configuration (modulo minor differences due to cache state differences). Therefore, detailed simulation and performance measurement is done only for DSWPed loops across various threading versions.

However, highly detailed modeling of core as well as memory architecture and large input set sizes of benchmarks, preclude the possibility of simulating all iterations of each and every invocation of a given loop in a reasonable time. Popular strategies to reduce simulation time include fast-forwarding (functional simulation) a few billion instructions into the program and then simulating in detail for several million instructions, running applications to completion albeit with reduced input sets [26] and sampling only select program regions [57]. In recent years, sampling techniques based on statistics theory such as SMARTS [74] have demonstrated that it is possible to estimate whole program behavior to desired confidence levels by doing detailed simulation for only a small number of discrete chunks chosen from across the entire dynamic execution trace and doing only functional simulation for the rest of the execution. The TurboSMARTS [73] methodology drives down simulation time even further by checkpointing architectural and select microarchitectural (caches, branch predictors, etc.) state through functional simulation at select points of execution and initiating SMARTS-style sampling in parallel from all the checkpoints.

This dissertation uses a similar methodology, albeit with some key variations. First, the above sampled simulation techniques work fine only across microarchitectural changes, since they rely on sampling a specific number of instructions across various simulator configurations. That, clearly, is unsuitable for evaluating both code and microarchitecture changes, when going from single-threaded execution to DSWPed execution. The strategy

[1]The only modification to pre-loop and post-loop code in multithreaded code versions are a few additional instructions to communicate loop live-in information to auxiliary threads and to communicate loop live-outs from auxiliary threads back to the main thread

used in this dissertation is as follows. Given the manner in which DSWP partitions a loop, the only thing that is constant across single-threaded and multi-threaded versions of a loop is the total work done per loop iteration and the number of loop iterations executed. Thus, instead of an instruction being used as the smallest logical unit of work during sampling, a *loop iteration* is used as the smallest unit. Several discretely chosen chunks from the loop iteration space are sampled for a particular invocation of the loop. Like in SMARTS, statistical sampling, including warmup, is done at specific periodicity for a given loop. Second, TurboSMARTS-style checkpoints of architectural and microarchitectural state are collected at the beginning of randomly chosen loop invocations and the loop iteration granular SMARTS sampling is initiated from these checkpoints.

While some loops have high trip counts (i.e. invocation counts) but execute only few iterations per invocation, others have low trip counts but execute many many iterations for each of those invocations. Given the varied nature of benchmark behavior, there is no clear strategy to determine the sizing of the iteration chunks for sampling, the probability for collecting checkpoints and the number of such checkpoints. For this work, the above numbers were arrived at on a trial-and-error basis. A general rule of thumb that was used to set the checkpoint probability and the statistical sampling interval was to make sure that a total of at least 10000 loop iterations are simulated across all checkpoints. For example, for high trip count loops, checkpoint collection probability was decreased to obtain wider coverage across the entire program execution. However, if a high trip count loop has low iteration count per invocation, then in order to sample 10000 loop iterations, a large number of checkpoints were collected. Benchmarks `188.ammp`, and `ks` are the only exceptions to this rule. Both these benchmarks had an outermost loop DSWPed. Unfortunately, the outermost loop does not execute sufficiently many times, resulting in smaller sample sets. The 10000 figure is not large enough to estimate the performance of an individual code-model configuration (for example, single-thread execution on baseline machine model) to a desired confidence level with a narrow enough confidence interval. However, performance *comparison* metrics (e.g. speedup of one technique relative to another) for a given application across code/architecture changes tend to demonstrate strong correlation during various phases of program execution and have been shown to require much lower number of samples to yield tight confidence intervals [35]. The performance comparisons in this dissertation are given at a 95% confidence level and the accompanying error bars are shown in all speedup graphs. The accompanying error bars, computed as per Luo and John's formula [35], confirm that the sample set size chosen is sufficient to yield narrow enough confidence intervals.

4.4 Summary

This chapter presented a detailed description of the experimental methodology used in this dissertation, including information on the benchmarks used for evaluation, sampling methodology, baseline simulation models, and performance analysis methodology. The next chapter presents results and analysis from a detailed evaluation of the latency tolerance and scalability characteristics of DSWP.

Chapter 5

Performance Evaluation of DSWP

The first part of this chapter focuses on understanding the run-time behavior of 2-thread DSWP and presents empirical data that highlights the variable latency tolerance property of DSWP. The second part presents a performance scalability study of the automatic DSWP implementation in the VELOCITY compiler and analyzes the performance and bottlenecks for DSWP when moving from 2 threads to 4, 6, and 8 threads. This chapter uses the high performance synchronization array communication support presented in Chapter 3 to evaluate the latency tolerance and scalability aspects of DSWP. Chapter 6 is devoted to evaluating the performance of different types of communication support for DSWPed codes.

5.1 Performance of 2-thread DSWP

The speedup of automatically generated 2-thread DSWP codes over single-threaded execution on the baseline in-order processor is shown in Figure 5.1. The geometric mean loop speedup achieved with the automatic partitioner is 1.20X. Performance improvement is achieved due to two main reasons - multi-threaded scheduling exposes more parallelism and better latency tolerance is achieve through decoupled execution.

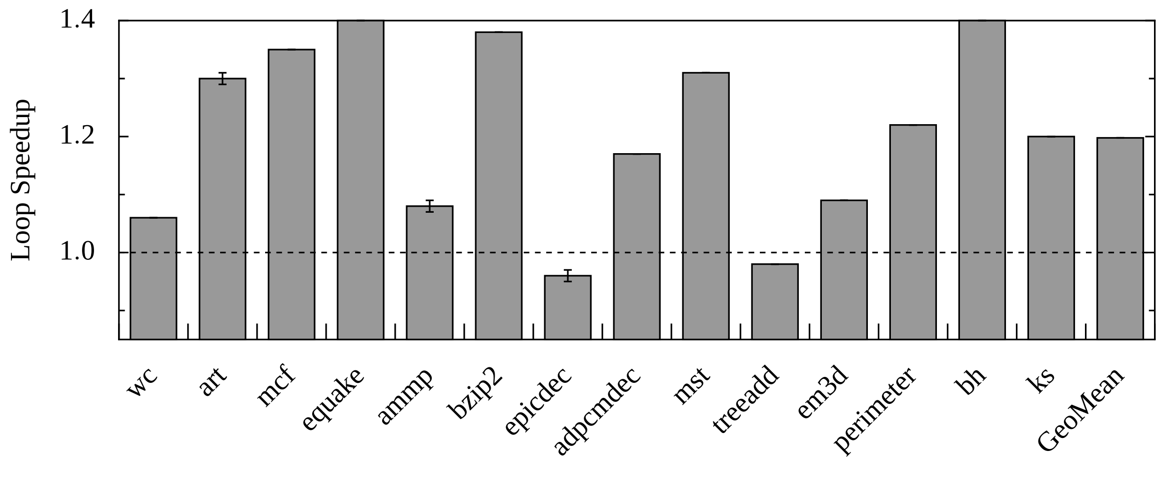


Figure 5.1: Speedup of fully automatic 2-thread DSWP with 32-entry inter-thread queues.

5.1.1 Balancing Threads Better

Since the speedup achievable from pipelined multithreading is directly correlated to the amount of overlap achieved among concurrently executing threads, the dynamic execution weights of the individual threads must be as well-balanced as possible to achieve the best overlap and hence the best speedup. In order to determine the balance among the automatically generated partitions, an experiment that simulates ideal communication behavior[1] is set up to measure the relative performance of the individual partitions. With non-blocking communication operations, the performance of an individual thread is determined solely[2] by the execution latency of its operations and is not influenced by the execution of other threads.

In theory, the optimal partitioning of the DAG_{SCC} can be demonstrated to be NP-complete through a reduction from *bin packing* [40]. In this section, a straightforward feedback-driven iterative approach is used to re-balance instructions among automatically generated DSWP threads to obtain better load balance and also to understand the efficacy of the compiler's partitioning heuristic. This approach is a heuristic; it does not methodi-

[1]Communication operations are made non-blocking by simulating queues with 10000 elements. This size is large enough to allow non-blocking `produce` and `consume` instructions.

[2]Well, this is not exactly true. Cache interference behavior is changed in such ideal communication experiments, since memory accesses from temporally far apart iterations could now happen simultaneously leading to different memory subsystem behavior than simulations with more realistic queue sizes.

cally explore the entire space of possibilities, nor is it provably optimal. It comprises the following steps:

- First, the iteration completion rates of individual threads, under ideal communication behavior, are used to determine the slowest thread. In 2-thread DSWP, a better balance can be obtained by moving operations from the slowest thread (source thread) to the other thread (destination thread).

- Second, the execution latency (L_{target}) for ideal DSWP performance is computed as the average of the execution latencies of all the individual partitions. The goal, then, is reduced to determining the operations that need to be moved from the source thread to the destination thread to strike the desired balance i.e. to move the execution latency of the individual threads as close as possible to the ideal execution latency, L_{target}.

- Third, using the bottleneck analysis presented in Section 4.2, key bottleneck instructions and their contributions to the overall execution time (after taking into account all possible intra-thread stall overlaps) are identified. However, only a subset, S, of these instructions are moved from the source thread to the destination thread, such that the absolute difference in the execution latency of the source thread and the sum of the latency contributions of the individual instructions in the set, S, is as close to L_{target} as possible. The difference should also be smaller than the absolute difference between the execution latency of the source (or the destination) thread and L_{target}.

- Fourth, the compiler is fed the new partition (source thread minus instructions in S, and destination thread with instructions in S). If the compiler finds the new partition untenable (due to the creation of cyclic inter-thread dependences), then the procedure stops. Otherwise, the above steps are repeated with the newly generated partition, since the run-time effects of the new partition may be different from the old partition

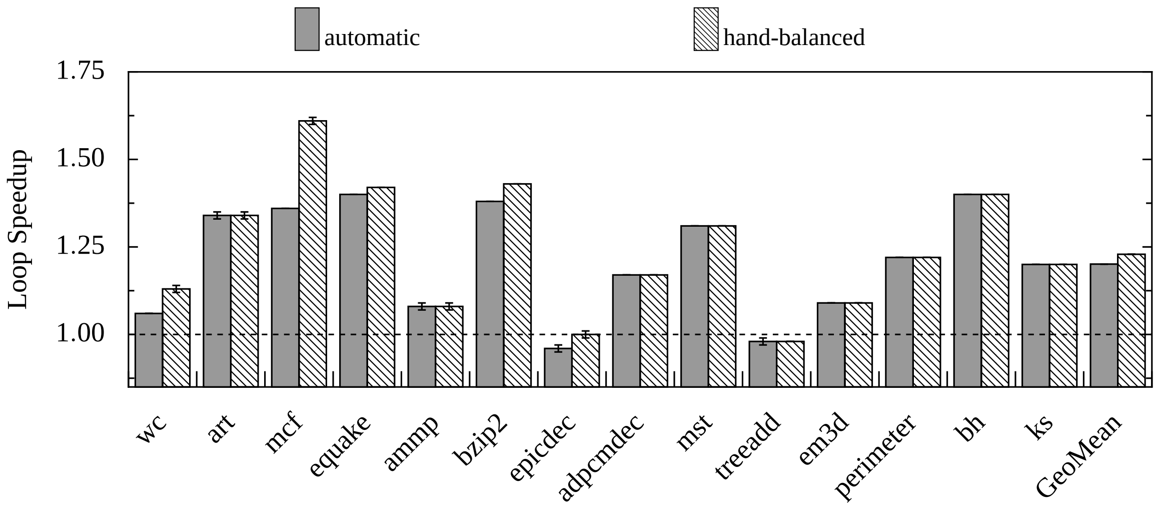


Figure 5.2: Performance improvement from feedback-driven load rebalancing.

and hence it becomes necessary to rerun the ideal communication experiments to determine the new bottleneck instructions.

For each benchmark, the above steps were applied iteratively. All but `mcf` required exactly 1 iteration to attain their peak performance. While the compiler was not able to repartition threads for benchmarks `treeadd`, `perimeter`, and `bh`, since the suggested partition was untenable, it was able to successfully repartition threads for all the other benchmarks and deliver peak performance. `mcf`'s performance continued to improve through 5 successive repartitionings before stabilizing at 1.61X speedup. Its original performance speedup was 1.36X. Similarly, the performance of benchmarks `wc`, `equake`, and `bzip2` improves from 1.06X to 1.13X from 1.4X to 1.42X and from 1.38X to 1.43X respectively. Benchmark `epicdec`'s DSWP performance improves from a slowdown to a zero speedup situation. The baseline was single-threaded execution running on an in-order issue processor configuration. Figure 5.2 shows the performance improvement from manual repartitioning for all benchmarks. The results indicate that while rebalancing does help improve performance by a lot in a few cases, for a majority of the cases the compiler's heuristic suffices to strike the correct balance among partitions. These partitions will henceforth be called *rebalanced partitions*.

```
 1  while( node != root ) {
 2    while( node ) {
 3      if( node->orientation == UP )
 4        node->potential = node->basic_arc->cost +
 5                          node->pred->potential;
 6      else {
 7        node->potential = node->pred->potential -
 8                          node->basic_arc->cost;
 9        checksum++;
10      }
11      tmp = node;
12      node = node->child;
13    }
14
15    node = tmp;
16    while( node->pred ) {
17      tmp = node->sibling;
18      if( tmp ) {
19        node = tmp;
20        break;
21      } else
22        node = node->pred;
23    }
24  }
```

Figure 5.3: Traversal loop in `mcf`

As can be seen from the graph, 2-thread DSWP with rebalanced partitions provides speedups ranging from 6% to 61% over single-threaded execution. These results are obtained by achieving an out-of-order scheduling effect on an in-order processor.

For example, consider the traversal loop for `mcf` shown in Figure 5.3. The calculation to identify the next node to visit in the tree often involves traversing not just a child pointer (line 12) in the innermost computation loop, but also sibling and predecessor pointers (lines 17 and 22 respectively) in a loop following the computation loop. Each iteration of the innermost computation loop performs two pairs of dependent loads (lines 4-5 and 7-8). Cache misses in these instructions followed by subsequent uses force the in-order processor to delay the execution of the sibling pointer traversal in the loop following the computation loop. Subsequent misses in the sibling traversal are taken *after* the load-use stalls, rather than being overlapped with the stalls. Static scheduling techniques fail to allow the sibling traversal loads to be overlapped with the computation loads because the

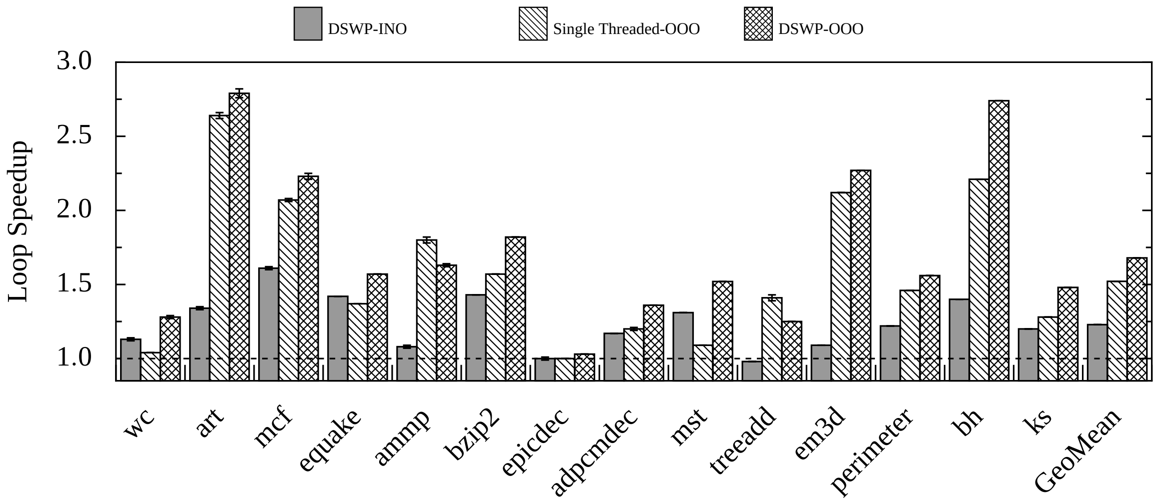


Figure 5.4: Speedup from DSWP execution on in-order and out-of-order processors.

Number of Cores	2
Core	Functional Units - 6-issue, 6 ALU, 4 Memory, 2 FP, 3 Branch
	RUU Entries - 128
	L1I Cache - 1 cycle, 16 KB, 4-way, 64B lines
	L1D Cache - 1 cycle, 16 KB, 4-way, 64B lines, Write-through
	L2 Cache - 5,7,9 cycles, 256KB, 8-way, 128B lines, Write-back
	Maximum Outstanding Loads - 64
L3, memory and coherence model	Same as in-order configuration (refer Table 4.2)

Table 5.1: Dual-Core Out-Of-Order Issue Configuration.

loads exist in two independent loops. In the multithreaded implementation, the child *and* predecessor/sibling traversal loads are moved into the traversal thread. Stalls incurred in the computation loop do *not* affect the traversal thread, allowing the misses incurred in both loops to be overlapped. This is similar to what would occur in an out-of-order processor. Independent instructions that occur logically after the stalled instruction can be executed while stalled instructions wait for their source operands.

5.1.2 Latency tolerance through decoupling

Since DSWP achieves an out-of-order effect on in-order processors, it is only logical to compare this technique with out-of-order execution. Figure 5.4 compares the performance of single-threaded execution on an out-of-order core to the execution of rebalanced DSWP partitions on dual-core CMPs with in-order and out-of-order cores. The graph indicates that

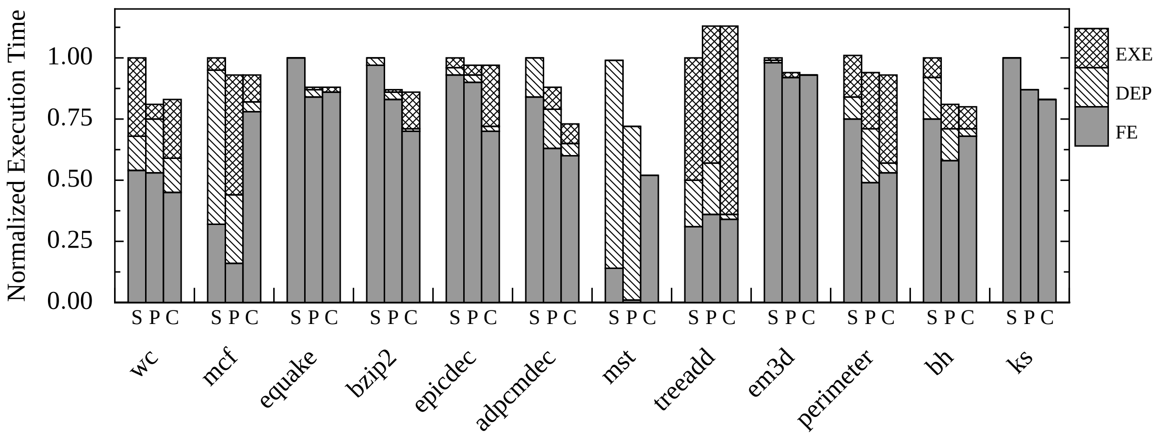


Figure 5.5: Distribution of iteration time. S=single-threaded, P=producer, C=consumer.

in some benchmarks (`wc`, `equake`, and `mst`) the out-of-order scheduling effect of DSWP on in-order cores is able to expose more coarse-grained thread level parallelism (TLP) from far apart loop iterations than traditional single-threaded execution on an aggressive out-of-order core, which can only expose local instruction level parallelism (ILP). However, as the rightmost bar in the figure shows, for all benchmarks except `treeadd`, DSWPed execution on dual-core CMPs, with out-of-order cores, can take advantage of aggressive cores to expose local ILP in addition to exposing coarse-grained TLP to yield an additive performance improvement over both DSWPed execution on in-order cores as well as single-threaded execution on aggressive uniprocessor out-of-order cores. The aggressive uni-processor out-of-order core (Table 4.3) had twice the resources (including functional units, load-store units, RUU entries, cache size, and cache ports) compared to a single core of the dual-core CMP configuration (Table 5.1) used to execute DSWPed codes.

To understand how DSWPed execution on out-of-order cores exposes more parallelism than single-threaded execution on a more aggressive out-of-order core, it is important to understand what happens to the program critical path in both cases. Figure 5.5 presents a bottleneck analysis of the critical path for the DSWP and single-threaded out-of-order executions for all benchmarks except for loops from `art` and `ammp`, which did not have clearly identifiable critical paths. One or more critical path instructions are identified in each benchmark loop and the time between two consecutive executions of critical path instructions is accounted for as front-end stall *FE*, (includes non-overlapped cycles when the second critical path instruction has not even entered the pipeline or is stuck in the front-end of the pipeline), dependence stall *DEP* (accounts for non-overlapped cycles the second instruction spends waiting in the RUU for its dependences to be resolved) and execution stall *EXE* (cycles spent by the second instruction in execution that are non-overlappable with the lifetime of a preceding critical path instruction).

As can be seen, *FE* stalls account for a major portion of the iteration time in single-threaded execution in most of the benchmarks. In comparison, the producer thread in the DSWPed version spent far less time in the front end of the machine despite the fact that its width and instruction window size are *half* that of the single-threaded core. This occurs for reasons described in Section 2.3. DSWPed execution dramatically reduces the *FE* stalls on the critical path which directly contributes to the improved performance in almost all the benchmarks. In `wc`, the *FE* stalls reduce only marginally due to the very small size of the loop as well as the fact that the byte load instruction that reads the input character stream almost always hits in the cache leading to fewer misses and hence the latency does not vary by much through the loop's entire execution. In `treeadd`, the *FE* stalls actually *increase* leading to a slowdown over single-threaded out-of-order execution. The reason for `treeadd`'s poor *FE* behavior in DSWP execution is that the partitioner ends up putting the majority of the original single-threaded loop's instructions in the producer thread. However, to supply control flow information to the consumer thread, the partitioner must also insert quite a few `produce` instructions. Dynamically, the producer thread must only fetch about 90% of the number of instructions the single-threaded program has to fetch. Since the producer thread has only half the width, this leads to increased *FE* stalls. Overall, this analysis demonstrates how prioritized execution of critical path instructions allows for more rapid completion of the loop's critical path by avoiding stalls due to limited resource availability.

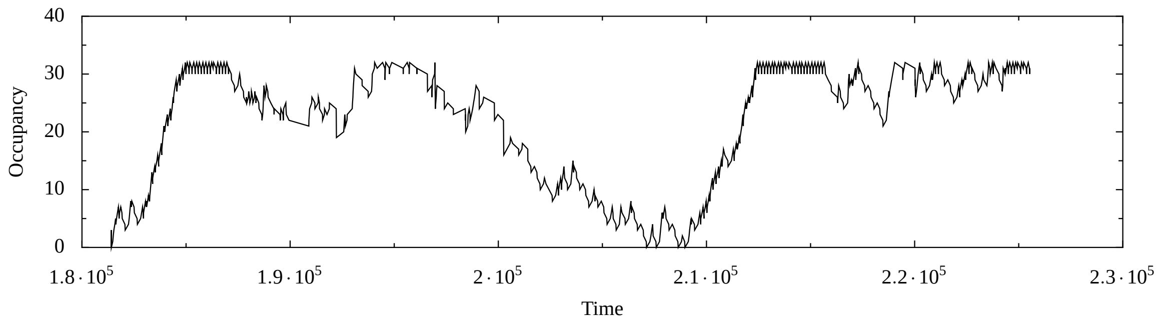


Figure 5.6: Synchronization array occupancy of `mcf` illustrates importance of decoupling

For example, consider the stall breakdown of `mcf`. Decoupled execution of critical path and off-critical path instructions means that the *FE* stalls experienced by the pointer-chasing load in the producer thread and the `consume` instruction that receives the pointer value in the consumer thread are cut in half when compared to the execution of pointer-chasing load in single-threaded execution. Misses in the off-critical path arising from the multiple loads required to compute `node->potential` (refer to Figure 5.3) delay initiation of pointer-chasing loads in single-threaded execution, whereas they do not affect pointer-chasing load initiation in DSWPed execution.

In order to illustrate the effect of decoupling more clearly, the changes in the inter-thread queue occupancies over time for a randomly chosen window of execution of the `refresh_potential` function in `mcf` is shown in Figure 5.6. A ramp-up indicates a producer thread run-ahead while a ramp-down indicates a consumer thread catch-up, either due to a stalled producer or because the producer thread has completed all its loop iterations. Negative occupancy in the graph indicates that consumer instructions have issued prior to their corresponding produce instructions and are waiting for data. Data buffered in the synchronization array decouples the behavior of the critical path thread and the off-critical path thread, allowing misses in each thread to be overlapped with work in the other thread. Note from the figure that there are periods of time when the occupancy of the synchronization array dips by as many as 8 entries. This drop indicates that the consumer was able to execute more iterations than the producer. To avoid stalling in this situation, it is necessary that at least 8 instances of the pointer-chasing load have completed ahead of the computation code. Since in a single-threaded processor only one copy of the pointer-chasing load appears in the instruction window per iteration of the loop, 8 entire copies of the loop would have to appear in the instruction window to avoid a stall. This loop in `mcf` contains 71 instructions, which means that avoiding this stall would require an out-of-order processor to be able to hold at least 568 instructions in its reorder buffer/RUU. To put this in perspective, the microarchitecture of the Intel Pentium® 4 processor, one of the most aggressive out-of-order processors built to date, can support only 126 in-flight micro-ops [22]. To conclude, these experiments have demonstrated that DSWP can provide the same amount of latency tolerance as aggressive out-of-order execution on processor configurations that are very difficult to build in practice.

5.2 Performance scalability of DSWP

This section examines the performance scalability of DSWP when moving from 2 threads to 4, 6 and 8 threads. Automatically generated partitions from the compiler were used for these experiments (i.e. no feedback-driven repartitioning was performed). The compiler created the requested number of threads, N, only if it could heuristically determine that it was profitable to partition the loop's SCCs among N threads, taking into account computation and communication costs. These experiments used N-way multi-core simulator models with in-order Itanium® 2 like cores to run the respective multithreaded code versions i.e. applications with 2, 4, 6, and 8 threads were simulated on multi-core simulator configurations with the corresponding number of cores. All simulation parameters were maintained the same as in the baseline model described in Chapter 4. Notably, the bus latency was fixed at 1 cycle for all the experiments. The synchronization array access latency for `produce` and `consume` instructions was 1 cycle. A pipelined bus interconnect carries traffic from the cores to the synchronization array and back. Cache coherence was

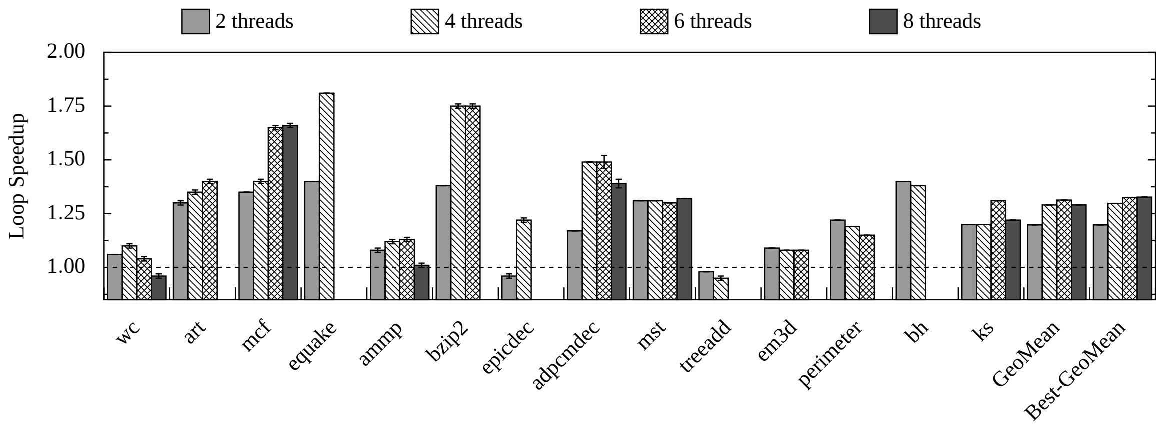


Figure 5.7: $DSWP_{Q_{32}}$: DSWP performance when moving from 2 to 4, 6, and 8 threads with 32-entry inter-thread queues and a bus interconnect. Note, a missing bar for a particular number of threads for a benchmark means the compiler was not able to partition the chosen loop for that benchmark into that many number of threads.

provided by bus-based baseline snoopy mechanism for all multi-core configurations.

Figure 5.7 shows the speedup provided by automatically generated DSWP threads relative to single-threaded in-order execution, when moving from 2 threads to 4, 6, and 8 threads. This performance graph will be referred to as $DSWP_{Q_{32}}$. Note that the graph shows two geometric means - a plain geometric mean (denoted “GeoMean” in the graph) and a best geometric mean (denoted “Best-GeoMean” in the graph). When calculating the plain geometric mean for N-thread code versions, if a certain benchmark did not have an N-thread code version (for example, `equake` does not have 6 and 8-thread versions), then for that benchmark, the speedup of the version with the next highest number of threads is used. For example, when calculating the *plain geometric mean* across all 8-thread versions, the speedup of `equake` from the 4-thread version is used since it does not have a code version with more than 4 threads. On the other hand, the *best geometric mean* for an N thread version represents the mean of the best speedups across all benchmarks for all code versions with number of threads fewer than or equal to N. It represents the speedup that can be achieved with code generated by an intelligent compiler that will generate the best multithreaded version for each benchmark for a given number of cores, even if it means generating fewer threads than available cores. The analysis presented here primarily uses the plain geometric mean to compare DSWP's performance in the presence and absence of bottlenecks. The best geometric mean is also provided to highlight the maximum speedup achievable through careful selection of multithreaded code versions.

To understand the performance of $DSWP_{Q_{32}}$, recall that the *autoDSWP* technique partitions the DAG_{SCC} such that there are no backward dependences among partitions. Since the performance of pipelined multithreading is limited by the performance of the slowest running thread, the maximum performance can be achieved by placing the "heaviest" SCC in a partition of its own and by making sure that no other partition is heavier than the partition with the heaviest SCC. This can be done by either load-balancing the remaining partitions in such a way so as to not exceed the weight of the heaviest SCC or if that's not possible, then, each SCC can be placed in its own thread. The heaviest thread is called the *bottleneck thread*. Often times, application loops contain a few large SCCs and many small, mostly single-instruction SCCs. Once the heaviest SCC has been placed in a thread of its own and no other partition is heavier (including ones with more than one SCC), the heaviest SCC thread becomes the bottleneck thread and it is no longer possible to obtain more performance improvement by partitioning the remaining SCCs among more threads. This trend is clearly seen in Figure 5.7, which shows that even for benchmarks that yield more than 2 threads, no performance improvement is seen beyond 6 threads.

On the contrary, a performance *slowdown* is seen for some application loops when moving to more threads, which is somewhat counter-intuitive. The theoretical performance improvement expected when moving to more threads no longer holds. Since *autoDSWP* virtually does no code duplication, the above slowdown, when moving to more threads, cannot be due to differences in the amount of computation. While the total computation remains constant across the four different multithreaded partitionings, the amount of communication varies. Figure 5.8 provides the normalized execution time breakdown of each thread for the different multithreaded partitionings for all benchmarks. The figure shows how the execution time of each benchmark for each thread configuration is spent in dif-

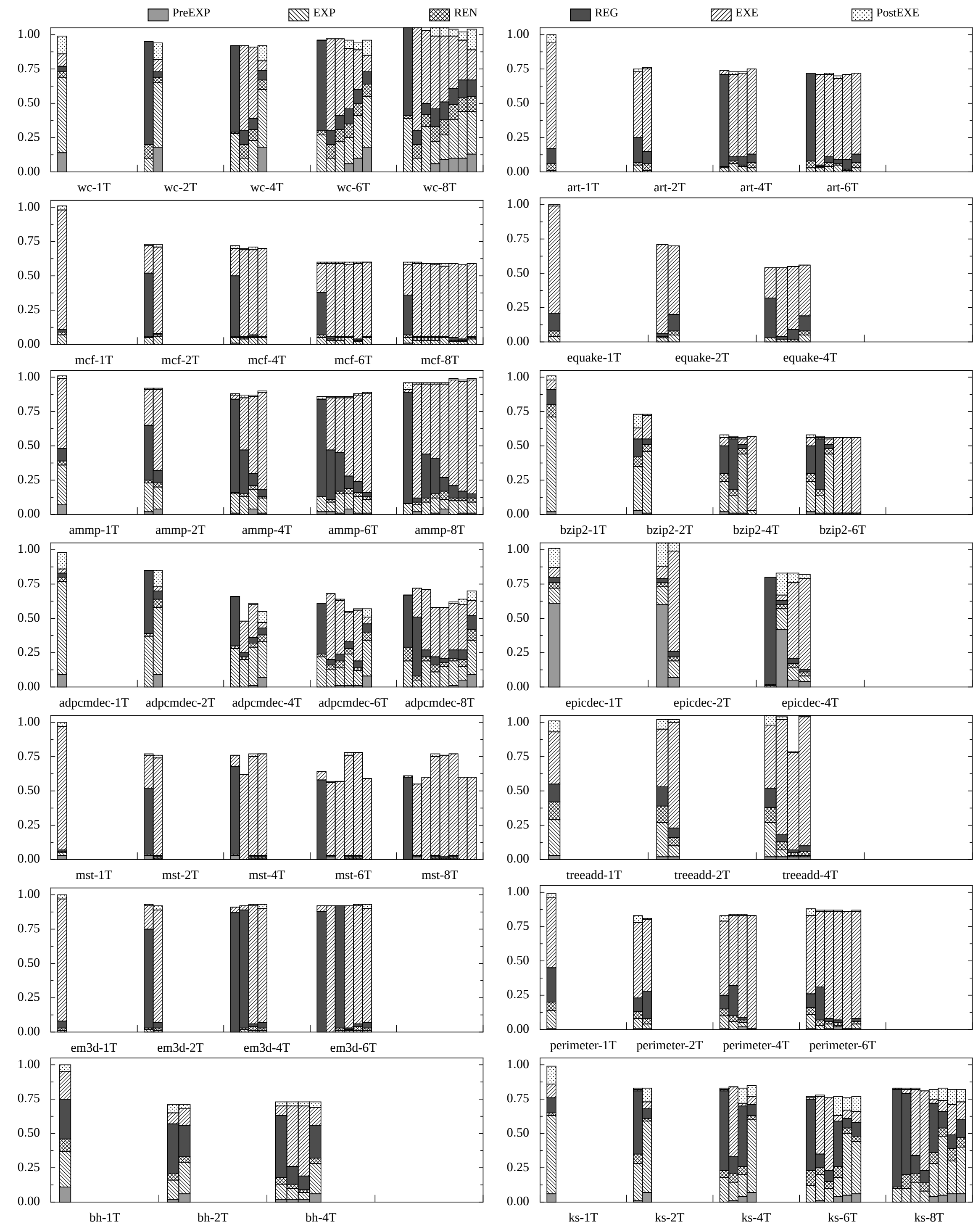
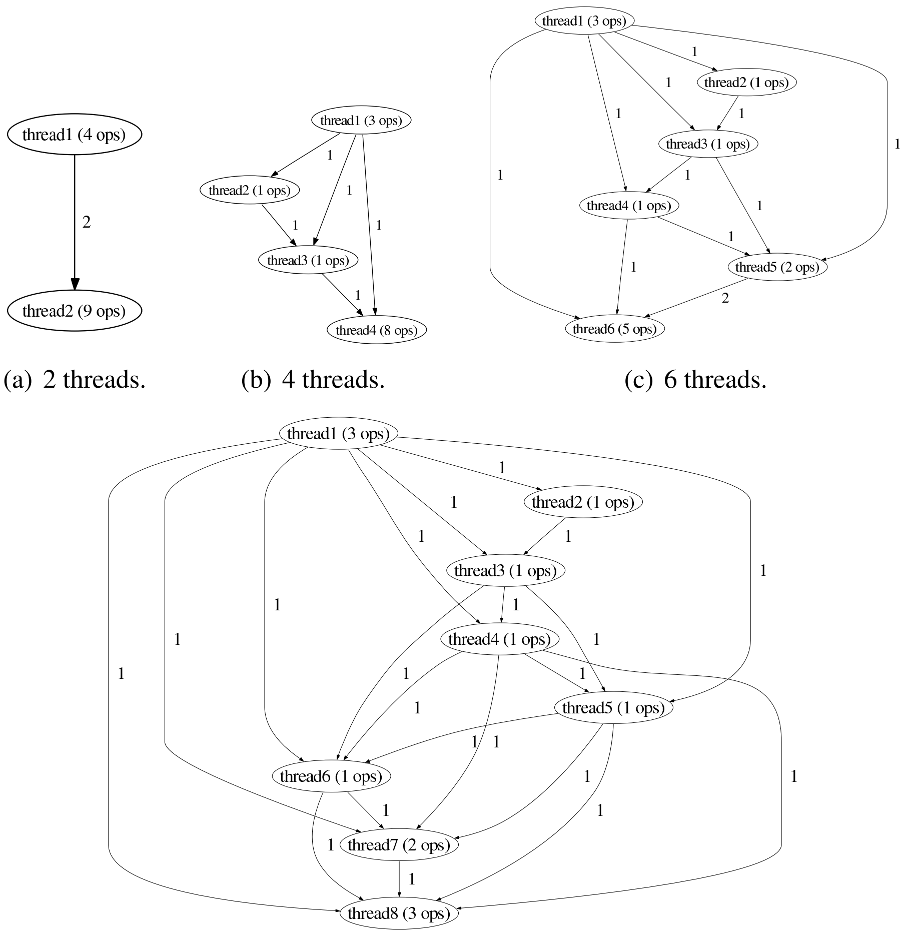


Figure 5.8: Normalized execution time breakdown of individual benchmarks when moving to more threads with 32-entry queues and bus interconnect

ferent stages of the processor pipeline. For this breakdown, the detailed stall breakdown provided by the analysis described in Section 4.2 is collapsed into the following six different aggregate stall groups - *PreEXP* (comprises stalls in the instruction fetch stages of the Itanium® 2 pipeline), *EXP* (stalls in the decode stage), *REN* (stalls in the register renaming stage), *REG* (stalls in the scoreboarding and register access stage and synchronization array renamer), *EXE* (stalls in the execution stage which accounts for all execution time including memory access, synchronization array access, etc.), and *PostEXE* (comprises stalls in the DET and WRB stages of the Itanium® 2 pipeline). The x-axis of each graph in Figure 5.8 represents the various code partitionings - $bench$-1T, $bench$-2T, $bench$-4T, $bench$-6T and $bench$-8T - of each benchmark $bench$. Within a cluster, for example $bench$-6T, the normalized execution time breakdown of each thread of that configuration is shown. Absence of a cluster for a benchmark means that the particular benchmark was not partitionable into the corresponding number of threads. Within a cluster, the bars corresponding to "upstream" threads appear to the left and the bars corresponding to "downstream" threads appear to the right. Benchmark `epicdec` shows a high *PreEXP* component in all the code configurations. This is because the stall analysis is made to account for stalls only in the loop being optimized. As a result, any time spent executing instructions outside the loop (for e.g. floating point library calls made my `epicdec`) is reported as *PreEXP* stall time.

The breakdowns show that when moving to more threads, the *EXE* component increases dramatically compared to the 1 or 2 thread configurations. A fine-grained breakdown of the stalls on a per instruction basis revealed that `consume` instructions were the main reason that led to the increased *EXE* component. For example, in the graph for `wc`, note that, while the first thread has a large *REG* component, due to frequent stalls by `produce` instructions in the synchronization array renamer due to queue full conditions, the second and third threads have large *EXE* components, due to frequent stalls by `consume` instructions on queue empty conditions. Given the classical notion of pipelined execution, this is very counter-intuitive. To explain this performance anomaly when moving to more threads, it is



(a) 2 threads. (b) 4 threads. (c) 6 threads.

(d) 8 threads.

Figure 5.9: Thread dependence graphs for loop from `wc`.

important to understand the actual communication pattern among threads and their run-time behavior.

5.2.1 Linear and non-linear thread pipelines

Given a linear chain of producer-consumer threads (i.e. thread 2 consuming from thread 1, thread 3 consuming from thread 2 and so on), the communication rate in the chain will be determined by the slowest thread and all threads will produce and consume at the exact same rate as the slowest thread. Such thread pipelines will be called *linear pipelines*. As mentioned before, the maximum performance attainable by such thread pipelines is $\frac{S}{D_H}$, where S is the single-threaded execution time, D_i is the execution time of thread i of the pipeline and H is the slowest thread in the pipeline. This expression does not say anything about the communication requirements of such pipelines. In particular, if a linear pipeline had insufficient queue buffering, then the factor D_H will increase to include the time the slowest thread spends waiting for data arrival, thereby adding inter-core communication delays to the overall thread execution time. But, such a situation can be avoided if inter-thread queues are sized appropriately. In particular, if the time taken to communicate a data item or a synchronization token from one thread to another is C cycles, it takes a total of $2 \times C$ cycles for a producer thread to communicate a value to a consumer thread and for the consumer to communicate its acknowledgment to the producer. This round-trip communication will be called a *synchronization cycle*. Since all threads in the pipeline need only communicate at the same rate as the slowest thread H i.e. once every D_H cycles, all inter-thread queues need only be as big as the queues leading into and out of thread H. If the synchronization cycle delay, $2 \times C$, is less than D_H, then D_H is the limiting factor and only one entry is needed in all inter-thread queues (no buffering is needed if communication happens instantaneously, i.e. C equals 0). On the other hand, if the round-trip time is greater than D_H, then the number of loop iterations the slowest thread can execute in that time is $\frac{2 \times C}{D_H}$. Consequently, there needs to be at least these many queue slots to keep the slowest thread continuously busy. Thus, the minimum[3] queue size necessary to prevent inter-core communication delays from being added to thread execution times is given by $\lceil \frac{2 \times C}{D_H} \rceil$.[4]

As long as the inter-core communication latency is less than or equal to the computation time, queue sizes of 1 or 2 will suffice to provide peak throughput. Increase in computation time will only reduce the demand for more queue entries. This is a very desirable property of linear pipelines as it helps place reasonable bounds on inter-thread queue sizes, enabling optimal communication support design.

[3]This is the minimum queue size without accounting for variability in data production and consumption rates.

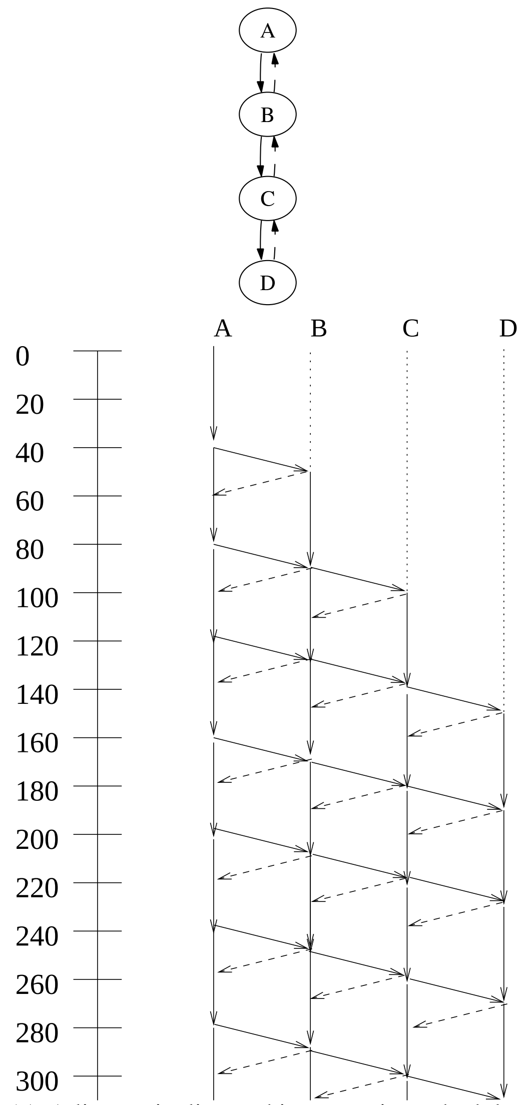
[4]This can be easily augmented to account for different communication costs between different pairs of threads.

However, in practice, general-purpose applications, often do not yield linear pipelines. As SCCs are partitioned among more threads, more inter-thread dependences are created amongst threads, since previously local (*inter*-SCC but *intra*-thread) dependences may now need to be communicated between threads. The communication pattern among constituent threads is quite varied and the partitioner ends up creating dependences between *almost* every pair of upstream and downstream threads. Such thread pipelines will be referred to as *non-linear pipelines*. Consider the example of `wc` shown in Figure 5.9. The figure shows the thread dependence graphs of different partitionings of the `wc` loop. It also shows the number of operations (compiler intermediate representation operations, not machine operations) in each thread and the label on each edge indicates the number of queues running between a pair of threads. Except for the 2 threads case (Figure 5.9a), the dependence graphs for the other cases (4, 6 and 8 thread cases in Figure 5.9b, 5.9c and 5.9d respectively) do not turn out to be linear pipelines. Such non-linear thread pipelines, while still providing PMT parallelism, experience certain communication bottlenecks that lead to below par performance.

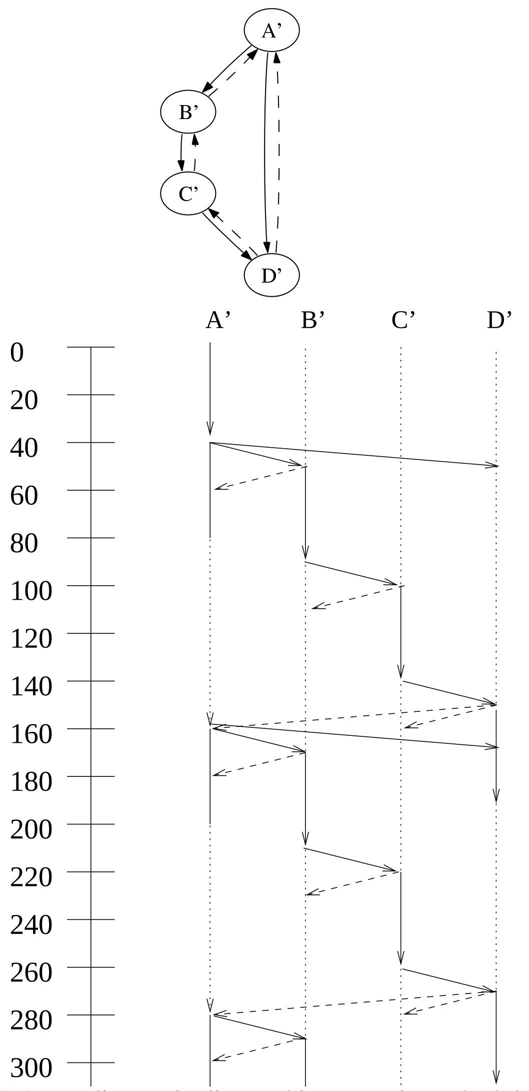
To understand the communication bottlenecks in such pipelines, consider the thread dependence graphs and the execution schedules of a 4-thread linear pipeline, *ABCD*, and a 4-thread non-linear pipeline, *A'B'C'D'*, in Figures 5.10a and 5.10b respectively.

The dashed arcs in the backward direction in the thread dependence graphs represent synchronization dependences going from consumer threads back to their producers. These arcs indicate to the producer when it is permissible to write to a particular queue location. When a consumer thread is slower than the corresponding producer thread, the latter has to block after filling up the queue, until the consumer thread frees up a queue slot, to produce the next data item. These arcs become relevant when inter-thread queue buffering is not sufficient to tolerate inter-core communication delays or the variability in data production and consumption rates.

For illustration purposes, suppose one iteration of the original single-threaded loop



(a) A linear pipeline and its execution schedule.



(b) A non-linear pipeline and its execution schedule.

Figure 5.10: Linear and non-linear thread pipeline execution with inter-core delay of 10 cycles and per-thread iteration computation time of 40 cycles.

takes 120 cycles and the individual threads each take 40 cycles, in both *ABCD* as well as *A'B'C'D'*, for executing one iteration of the loop in question. Let the inter-thread communication latency be 10 cycles. Even though, in practice, the compiler is free to schedule the communication instructions anywhere in a thread, for this example, assume that all `produce` instructions are executed at the end of thread's loop iteration and all `consume` instructions, at the beginning. Finally, for simplicity, all `produce` instructions in a thread will be blocked if any one `produce` blocks. A similar all-or-none behavior will be assumed for `consume` instructions as well. In the execution schedules shown in the above figure, a solid inter-thread arrow means a data value communication from the producer thread to its consumer. A dashed inter-thread arrow in the reverse direction denotes an acknowledgment signal from a consumer to a producer, indicating a queue entry is free to be reused. Dotted straight lines in a thread's schedule indicate periods of no activity in the thread, because it is blocked on a produce or a consume operation.

As the execution schedule in Figure 5.10a shows, the linear pipeline *ABCD* is able to finish a loop iteration once every 40 cycles. Notice that because the per-iteration time is 40 cycles and the synchronization cycle delay (round-trip time) is only 20 cycles, the acknowledgment for a queue entry arrives well before a producer thread is ready to produce the next data item. Therefore, a producer will never block on a queue full condition and the performance of the linear pipeline attains the theoretical maximum speedup of $\frac{S(120cycles)}{D_H(40cycles)}$ i.e. 3X. The important point to note here is that the linearity of the pipeline enables it to achieve this speedup with just 1 entry per inter-thread queue.

Now, consider the non-linear pipeline *A'B'C'D'*'s execution schedule in Figure 5.10b. This schedule has been drawn assuming 1-entry inter-thread queues to compare and contrast its performance with the linear pipeline from above. Notice that *A'B'C'D'* is able to complete only 1 loop iteration every 120 cycles (the first iteration of thread D' completes in cycle 190, and the second iteration in cycle 310) resulting in no speedup at all over single-threaded execution. The reason for this abysmal performance is due to inadequate queue sizing that leads to prolonged stalls, as can be seen by the long dotted lines in all threads in Figure 5.10b. So, why do single-entry queues, which were adequate to deliver peak throughput in the linear pipeline above, create performance bottlenecks here?

To answer that, observe that in Figure 5.10b, thread A' sends values to both threads B' and D'. Consequently, in any given iteration, before producing a data item, thread A' has to ensure that it can produce into the queues leading into both thread B' and thread D' before initiating both data sends (per assumptions stated above). In other words, whenever it is blocked on queue full condition, thread A' has to wait for acknowledgments from both threads. This requirement leads to communication bottlenecks, thereby slowing down multithreaded performance. In the execution schedule, notice that even though thread A' is ready to produce data after its second iteration as early as cycle 80, it has, by that point in time, received acknowledgment only from thread B'. It has to wait a further 80 cycles before it receives acknowledgment from thread D', at which point, it proceeds to produce the data to both threads B' and D'. And since thread A' is the head of the pipeline, the rest of the pipeline also stalls waiting for data from upstream threads. The fundamental problem here is that queue sizes for executing such non-linear pipelines cannot be determined solely from the inter-thread communication delay and the per-iteration computation time of the slowest thread.

For non-linear pipelines, the synchronization cycle expands to include the computation time of all intermediate threads as well as the one-way communication delays between the intermediate threads. For example, for thread A' above, the synchronization cycle comprises the communication delay from thread A' to thread B', the computation time in thread B', the communication delay from thread B' to thread C', the computation time of thread C', the communication delay from thread C' to thread D' and finally, the delay for the acknowledgment to go from thread D' to thread A'. More generically, the round-trip communication delay of the new longer synchronization cycle can be expressed as $2\times C+\sum_{i=1}^{N_s}(D_i+C)$, where N_s is the number of threads in the synchronization cycle s. By a similar reasoning as from before, the queues should be large enough to tolerate this round-trip delay, but only to the point of sustaining the maximum throughput. Thus, for non-linear pipelines, the minimum queue size needed to provide maximum PMT performance is

$$\lceil \frac{2 \times C + \max_{s,\forall s \in S}(\sum_{i=1}^{N_s}(D_i + C))}{D_H} \rceil$$

where S is the set of all synchronization cycles in a given thread dependence graph. The second term in the numerator causes non-linear pipelines to require longer queues to deliver peak throughput. This term also makes non-linear pipelines unwieldy for communication support design, since it is impossible to place an upper bound on the size of the inter-thread queues. Synchronization cycles can be made arbitrarily long due to the computation costs of intermediate threads, making it very difficult to design bottleneck-free communication support. This explains the anomaly in Figure 5.8, wherein threads experienced increased *EXE* stalls, due to the creation of non-linear thread pipelines, when moving to more threads.

5.2.2 Performance under ideal communication behavior

The 32-entry queue sizing was insufficient to tolerate longer synchronization cycles created by non-linear thread pipelines. This phenomenon is aggravating, especially when moving to 8 threads. To remedy the situation and evaluate the performance scalability potential of DSWP when moving to more threads, a second set of simulations (labeled $DSWP_{Q_\infty}$) were run, once again on different multi-core configurations, with enough cores to match the number of application threads. The only difference was that the queue sizes were set to infinity [5]. Figure 5.11 presents the speedup obtained from the different multithreaded configurations relative to both single-threaded in-order execution (at the bottom) as well as $DSWP_{Q_{32}}$ (at the top). Figure 5.12 shows the execution time breakdown of each thread in the different multithreaded code versions with infinite queue sizes normalized to single-threaded in-order execution.

[5]A size of 10000 was sufficient to ensure that no queue full/empty stalls occurred.

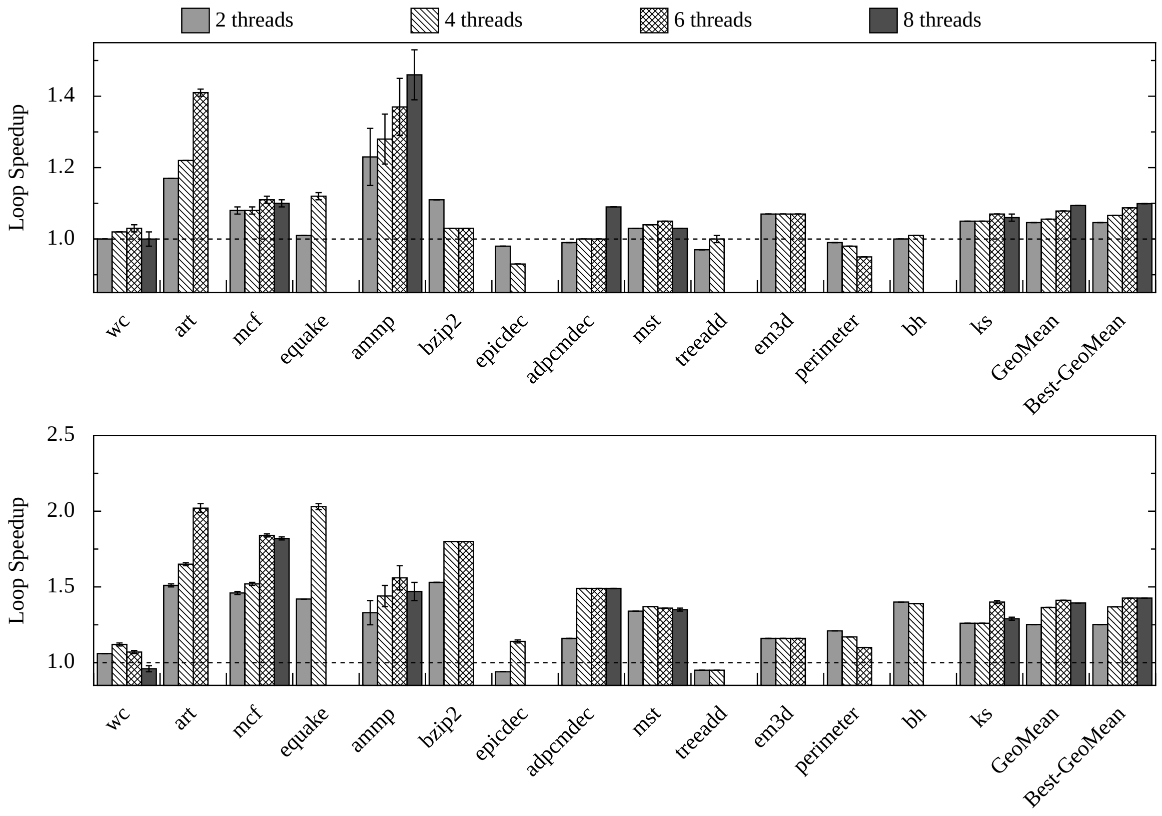


Figure 5.11: $DSWP_{Q_\infty}$: Performance of DSWP with 2, 4, 6, and 8 threads with infinitely long queues and a bus interconnect, relative to $DSWP_{Q_{32}}$ (top) and single-threaded in-order execution (bottom).

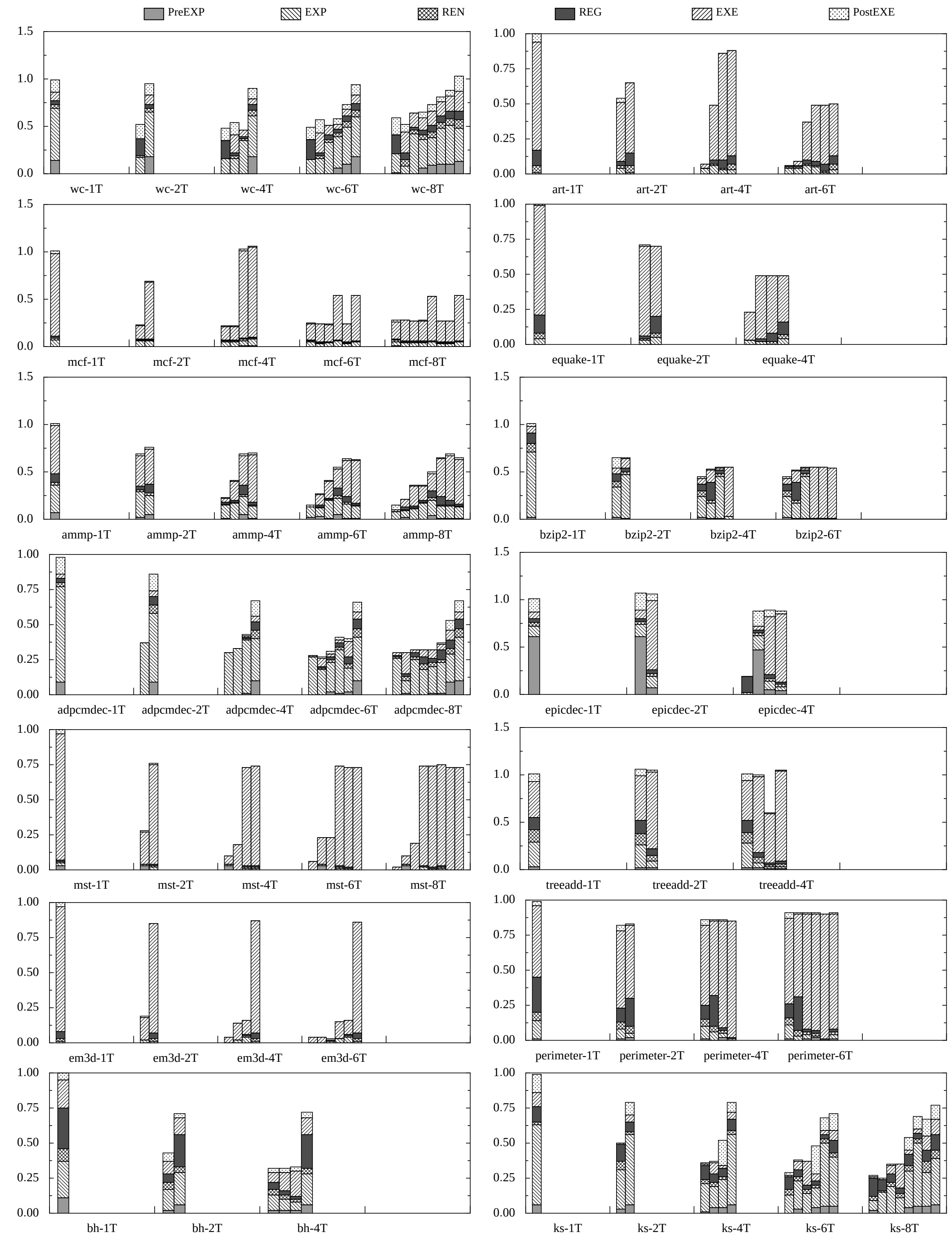


Figure 5.12: Normalized execution time breakdown of individual benchmarks when moving to more threads with infinitely long queues and bus interconnect.

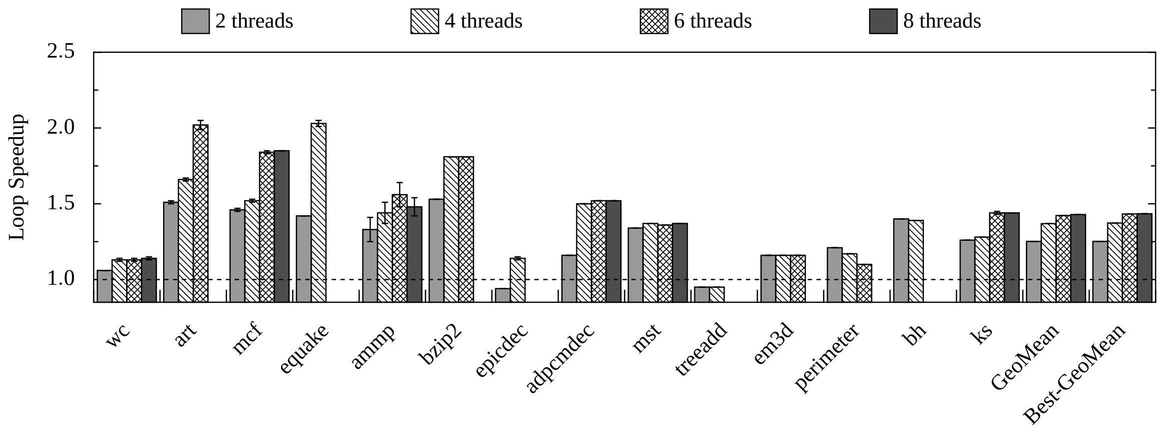


Figure 5.13: $DSWP_{Q_\infty+BW_\infty}$: Performance of DSWP with 2, 4, 6, and 8 threads with infinitely long queues and infinite communication bandwidth relative to single-threaded in-order execution.

Several observations are in order. As expected, easing the queue size limitation does alleviate the communication bottleneck imposed by non-linear pipelines and improves DSWP performance for most benchmarks when moving to more threads. The geometric mean speedups of $DSWP_{Q_\infty}$ with 2, 4, 6, and 8 threads are 1.25X, 1.36X, 1.41X, and 1.39X respectively, whereas, the geometric mean speedups of $DSWP_{Q_{32}}$ with 2, 4, 6, and 8 threads from Figure 5.7 were 1.20X, 1.29X, 1.31X, and 1.29X respectively.

Despite the overall improvement, there are several notable exceptions. Benchmarks `wc`, `mcf`, `ammp`, `perimeter`, and `ks` continue to see a performance degradation when moving to more threads. A closer look at the execution revealed that the arbitration policy of the bus interconnect carrying synchronization array traffic, always favored earlier threads. This caused threads later in the pipeline to suffer arbitration stalls in 6 and 8 thread scenarios. The removal of the bottleneck due to pipeline non-linearity with infinitely long queues resulted in producer threads earlier in the pipeline being greatly sped up, causing instructions from threads later in the pipeline to suffer interconnect contention stalls. In particular, when these stalls hit `consume` instructions, which were at the head of dependence chains, of downstream threads, performance slowdown was inevitable. This slowdown in downstream threads eventually slowed down the upstream producer threads as well.

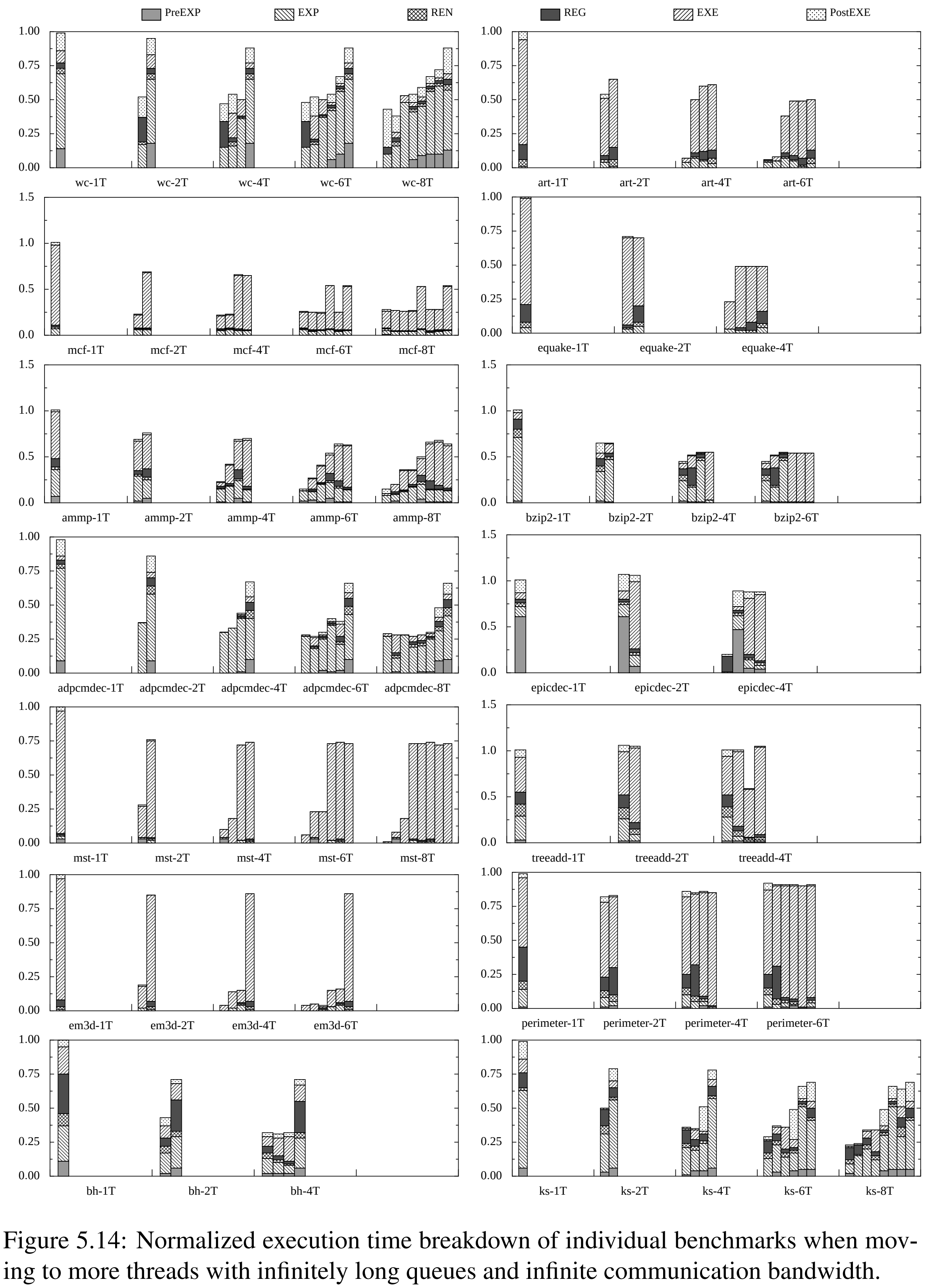
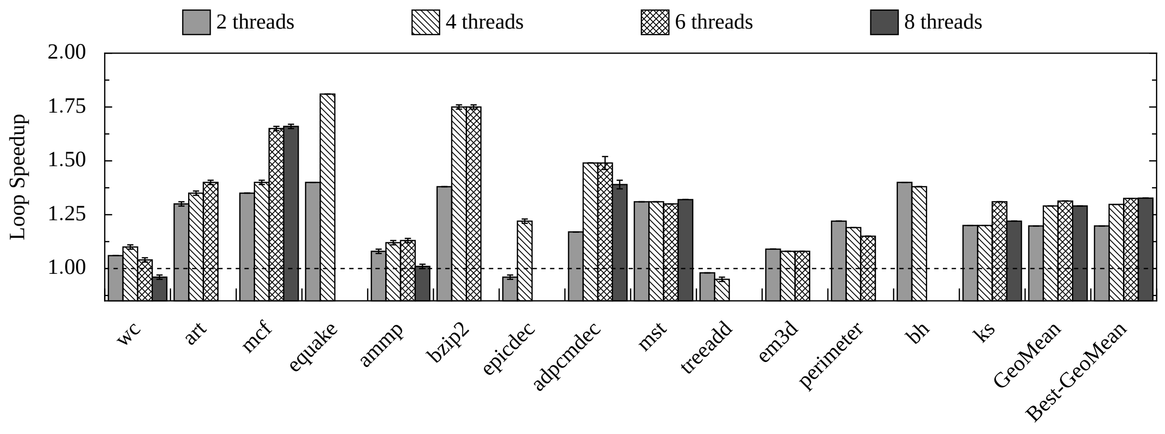
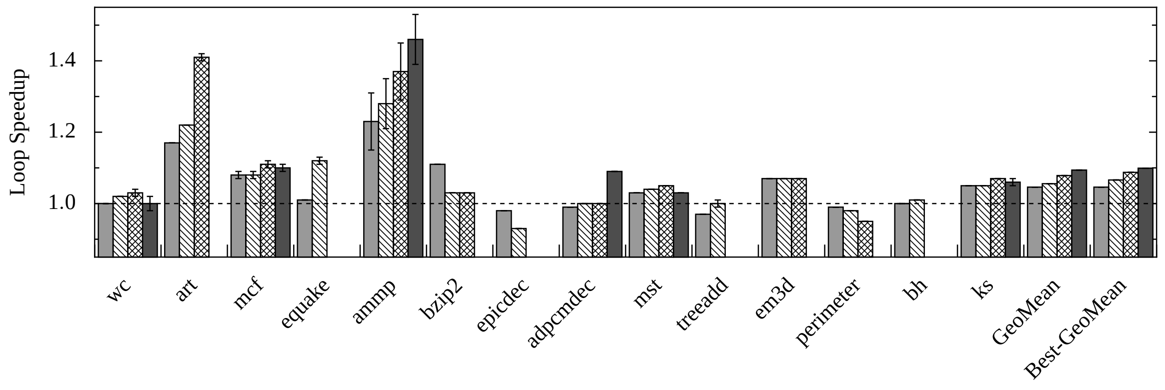


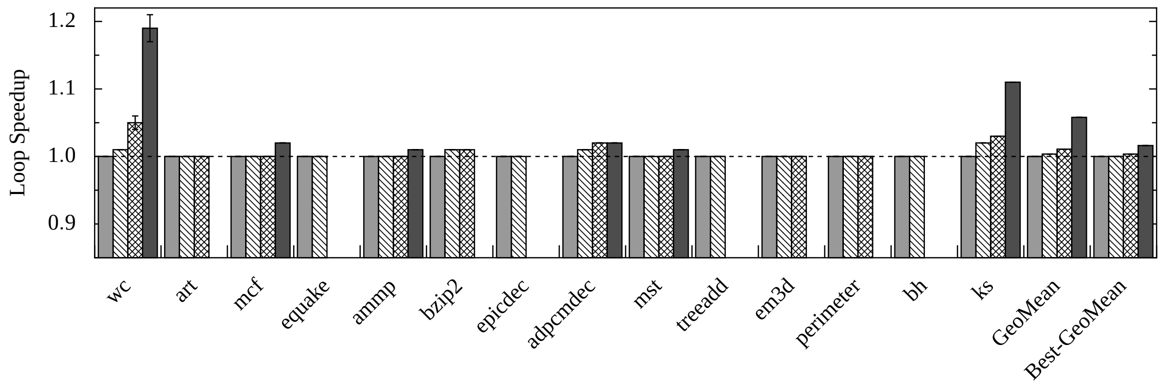
Figure 5.14: Normalized execution time breakdown of individual benchmarks when moving to more threads with infinitely long queues and infinite communication bandwidth.



(a) Speedup with 32-entry queues with interconnect and port contention (baseline: single-thread).



(b) Speedup with infinitely long queues with interconnect and port contention (baseline: $DSWP_{Q_{32}}$).



(c) Speedup with infinitely long queues and infinite communication bandwidth (baseline: $DSWP_{Q_{\infty}}$).

Figure 5.15: Relative performance of DSWP with 2, 4, 6, and 8 threads under different communication scenarios.

In order to alleviate finite bandwidth limitations to the synchronization array, the bus interconnect was replaced with a crossbar interconnect and the synchronization array was allowed to have as many ports as required to cater to requests from all cores every cycle. This configuration with infinite-sized queues and infinite communication bandwidth is labeled $DSWP_{Q_\infty+BW_\infty}$. This idealization made the 8-thread versions of benchmarks `wc`, `mcf` and `ks` perform no worse than the 6-thread versions. From Figure 5.13, which presents the overall speedup obtained by $DSWP_{Q_\infty+BW_\infty}$ relative to single-threaded in-order execution, the geometric mean speedups of $DSWP_{Q_\infty+BW_\infty}$ across 2, 4, 6, and 8 thread versions can be seen to be 1.25X, 1.37X, 1.42X, and 1.43X respectively. Figure 5.14 shows a thread-wise breakdown of the performance of multithreaded DSWP with infinitely long queues and zero port contention while accessing the synchronization array. The main difference between this and Figure 5.12 are the graphs for `wc`. In the latter, `wc` can be seen to have a performance degradation when moving to more threads. The infinite inter-thread queue (i.e. synchronization array) bandwidth remedies this situation and causes `wc`'s performance to fall in line with theoretical expectations. Similar improvements can also be seen for benchmarks `mcf`, `adpcmdec`, `mst` and `ks`.

Figure 5.15 presents the relative speedup graphs for all three communication scenarios - $DSWP_{Q_{32}}$, $DSWP_{Q_\infty}$, and $DSWP_{Q_\infty+BW_\infty}$. It highlights the incremental improvement obtained by easing just the queue size bottleneck relative to $DSWP_{Q_{32}}$ (Figure 5.15b) and by easing both the queue size as well as the interconnect contention bottlenecks (Figure 5.15c) relative to $DSWP_{Q_\infty}$.

Notice that `ammp` and `perimeter` continue to experience performance degradation even after elimination of interconnect contention stalls. Detailed analysis revealed that in `perimeter`, the best balance is reached with 2 threads, with thread 1 being the bottleneck thread. Since thread 1 contained only 1 SCC, further parallelization is made possible only by spreading thin the remaining SCCs among more threads. However, this leads to loss of locality in data accesses leading to increased coherence activity, thereby increasing the number of coherence-induced misses in the L2 caches. A similar problem is observed in `ammp` as well when moving from 6 to 8 threads. The best balance is reached with 6 threads. Moving to 8 threads only results in the creation of more coherence traffic leading to performance slowdown.

Finally, note that the best geometric mean speedup, which represents the maximum speedup possible with N cores (with number of threads fewer than or equal to N), for 2, 4, 6, and 8 threads improves from 1.20X, 1.30X, 1.32X, and 1.33X in the $DSWP_{Q_{32}}$ configuration to 1.25X, 1.37X, 1.43X, and 1.43X respectively in the $DSWP_{Q_{\infty}+BW_{\infty}}$ scenario. This overall performance improvement serves to highlight that it is not enough for the compiler to be intelligent enough to pick the best "code version" (i.e. multithreaded configuration), but that it must also strive to eliminate non-linear pipelines during partitioning, in order to achieve the best performance possible.

5.3 Summary

Through detailed simulations and performance analysis, this chapter first demonstrated the latency tolerance property of DSWPed codes. The geometric mean speedup of DSWPed execution on a 2-core CMP with in-order issue is 1.29X compared to single-threaded execution on one core. More saliently, the geometric mean speedup of DSWPed execution on a 2-core CMP with out-of-order issue from a 128-entry instruction window is 1.11X faster than single-threaded execution on an out-of-order processor with a double the RUU entries, cache ports and cache lines. Critical path stall analysis clearly demonstrated that DSWP achieves this improvement by avoiding *FE* stalls through decoupled multithreaded execution.

This chapter also analyzed and highlighted the various communication bottlenecks that must be overcome for DSWP to achieve theoretically predicted performance. Without any communication bottlenecks, DSWP delivers a geometric mean speedup of 1.25X to 1.46X when going from 2 to 8 threads. An important lesson from the performance scalability analysis is the need to make the compiler aware of the communication costs of linear and non-linear thread pipelines so that it can generate code to avoid communication bottlenecks. Performance loss due to false sharing can be avoided by using clever data layout schemes. For example, false sharing due to read and write accesses to different fields of a structure can be avoided by padding the structure with enough dummy bytes so that the relevant fields are moved to different cache lines.

The next chapter presents an evaluation of a variety of mechanisms for pipelined inter-thread communication. It quantitatively demonstrates the effect of each sub-component, of the design space presented in Chapter 3, on DSWP performance and identifies the main bottlenecks in the designs studied.

Chapter 6

Evaluation of Communication Support

This chapter evaluates four important communication support mechanisms that represent design variants ranging from existing commercial processor designs to designs like the synchronization array that leverage heavy-weight dedicated streaming hardware to maximize the performance of streaming codes. All selected design points ensure backward compatibility with legacy software. Using these design points, it is empirically shown that DSWPed codes do, in fact, tolerate transit delay. The results and analysis from this chapter motivate light-weight communication support optimizations described and analyzed in the next. Note that an evaluation of various communication support options for PMT was first undertaken in [50]. The main differences between the evaluation presented in this dissertation and the original work are the use of VELOCITY compiler framework in this dissertation and an improved write-forwarding implementation.

6.1 Systems Studied

The four design points explored were:

EXISTING. This design point is representative of existing commercial CMPs. It is a naïve implementation of shared memory software queues that relies only on the baseline coherence and consistency mechanisms. This design will serve as our baseline for measuring the hardware cost and operating system impact of other designs.

MEMOPTI. This variant will illustrate the efficacy of EXISTING with write-forwarding support, a low-impact memory subsystem optimization. This design requires little additional hardware (cache modifications for write-forwarding). Write-forwarding is limited to a core's L2 cache, to avoid polluting the L1 cache with short-lived inter-thread streaming data.

SYNCOPTI. This variant corresponds to the snoop-based synchronization design presented in Section 3.4 of Chapter 3 and will illustrate the benefits of using dedicated hardware to optimize communication operation sequences and synchronization, while still relying on the memory subsystem for queue backing stores and core-to-core interconnect. The design requires modifying core pipelines to execute `produce` and `consume` instructions, write forwarding logic (with the locality enhancements described in Section 3.2.5) and synchronization counters in the processor caches, and OS support to context switch the synchronization counters. Here too, write-forwarding does not propagate all the way to L1 and is terminated at the destination core's L2.

While this design does introduce more hardware than the previous designs, it remains fairly light weight. This design reuses the L2-L3 memory bus, a critical component of CMP architectures, rather than introducing a new dedicated network like the synchronization array. Such an approach makes optimal use of the available on-chip transistors; the single on-chip network can be provisioned according to the total system bandwidth requirements without regard to how such traffic is generated (application memory requests or inter-thread operand requests). This generality makes the solution appealing since it has potential to support various models of application parallelism. Additionally, use of memory as a backing store avoids introducing new dedicated stores, allows flexible queue sizing, *and* greatly reduces OS context-switch and virtualization costs.

HEAVYWT. This variant represents the performance achievable by hardware-heavy mechanisms such as the FIFOs provided by the scalar operand networks in Raw [65] or the synchronization array (SA). It combines the point along each design axis that should offer greatest performance without regard for hardware cost or OS impact. In addition to core modifications to execute `produce` and `consume` instructions, HEAVYWT introduces additional dedicated distributed on-chip queue backing store and a new interconnect network to connect processor cores to this backing store. The contents of the backing store and in-flight data buffered in the network must be part of a process's context. Consequently, this design variant *also* requires OS modifications to context switch this state. Since this state is concurrently updated by multiple threads belonging to the same process, the OS implications will be more far reaching than for the other design variants.

6.2 Experimental Setup

The benchmarks used for this evaluation are the exact same workloads that were used to evaluate DSWP in Chapter 5. The multithreaded workloads used here correspond to the rebalanced partitions derived in Section 5.1.1. Two types of codes were automatically generated - one with `produce` and `consume` instructions (same as the best-performing code from Section 5.1.1) and the other with code sequences for shared-memory software queue writes and reads.

The code sequences for the shared-memory software queue operations have been highly tuned to contain the minimal number of instructions possible. Despite that, the software overhead for a communication operation was 10 instructions (6, 1 and 3 instructions for synchronization, data transfer and stream address update respectively) with a dependence height of 4. The overhead for the `produce`-`consume` instructions based versions was just the one instruction for data transfer. On an in-order machine such as the Itanium® 2, these overheads tend to contribute significantly towards the overall execution time, especially for really tight loops. Code for software queue implementations was generated such that the spin lock loop, and lock release and queue pointer update were part of existing control

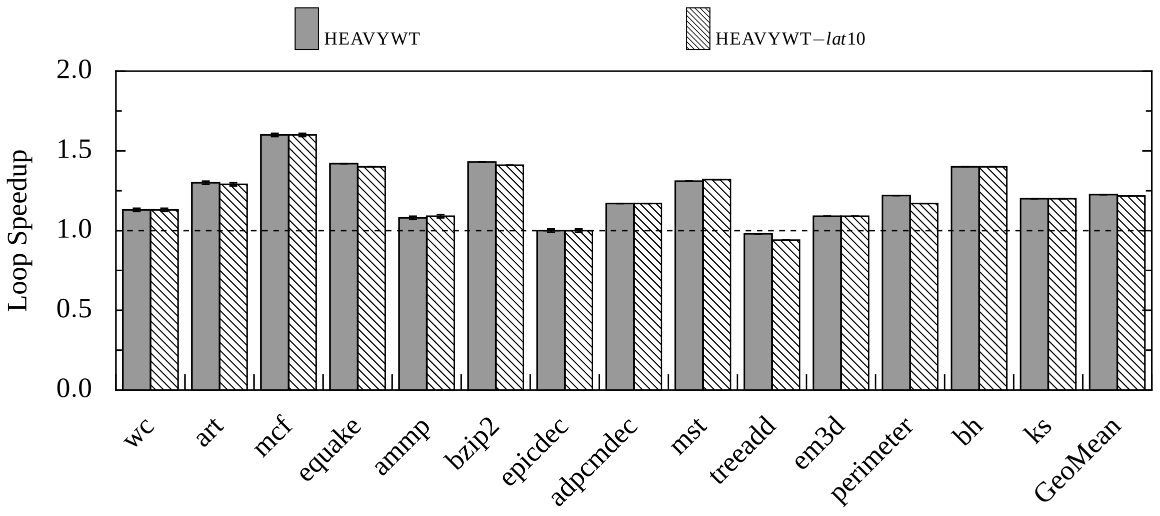


Figure 6.1: Effect of transit delay on streaming codes.

blocks so as to enable more compact scheduling.

All designs used 256 queues of depth 32 unless otherwise mentioned (not all queues were used by each application). For all designs using shared memory backing stores, the queue layout unit was 8. For all designs using write-forwarding, lines were forwarded only to the cores' private L2 caches and forwarding was initiated only after all queue entries on the line had been written. For the lone configuration with a dedicated backing store, HEAVYWT, the backing store was located in the consumer core, but queue synchronization counters were maintained at both the producer and consumer core. The dedicated store could service 4 concurrent operations per cycle and was connected to remote cores by a dedicated pipelined interconnect. Within a consuming core, the consume-to-use latency was 1 cycle. If not otherwise mentioned, the interconnect latency was 1 cycle.

6.3 Results and Analysis

First, to understand the decoupling present in the applications and to see the effect of transit delay on pipelined inter-thread communication, Figure 6.1 presents a performance comparison of three HEAVYWT variants. They differ only in the end-to-end latency of their dedicated pipelined interconnects. The left bar corresponds to a 1-cycle end-to-end latency (default HEAVYWT) and the right bar to a 10-cycle latency. All other parameters are held constant.

As the figure shows, the performance difference across the three design points vary from little to nothing. This is in line with the theoretical expectation that PMT codes tolerate transit delays. That being said, there is a small performance dip for a few benchmarks, namely, `equake`, `bzip2`, `treeadd`, and `perimeter`. These benchmark loops have one or more nested loops inside the outermost loop that is DSWPed. Upon DSWPing such loops, `produce` instructions in the outer loop can execute only after all iterations of its inner loop(s) are done. And, in order for the producer to be done with all iterations of its inner loop, either the queue size must be large enough compared to the number of loop iterations or in the more likely scenario, wait for the consumer to drain the queue so that the producer can keep producing to the queue. In the event the outer loop of the producer fails to build sufficient decoupling (i.e. ensure that the queue occupancy is greater than or equal to one), any `consume` instruction to that queue will be stalled until the queue occupancy becomes greater than zero. Such stalls are exacerbated by the increased 10-cycle delay for transferring a value from the producer core to the consumer core. This leads to the slight performance dip for the above benchmarks.

Next, Figure 6.2 gives a performance comparison all four design points with respect to baseline single-threaded execution. The speedup of each technique with respect to single-threaded execution is given. The normalized execution times of the producer (above) and consumer (below) threads for each of the four different mechanisms is given in Figure 6.3. From left to right, for each benchmark, the bars correspond to single-thread execution, HEAVYWT, SYNCOPTI, EXISTING and MEMOPTI respectively. Note, even though the overall performance of the producer and consumer cores are the same, their breakdowns look different due to different mix of communication and computation instructions and additionally, in the case of HEAVYWT and SYNCOPTI, it is due to the differences in

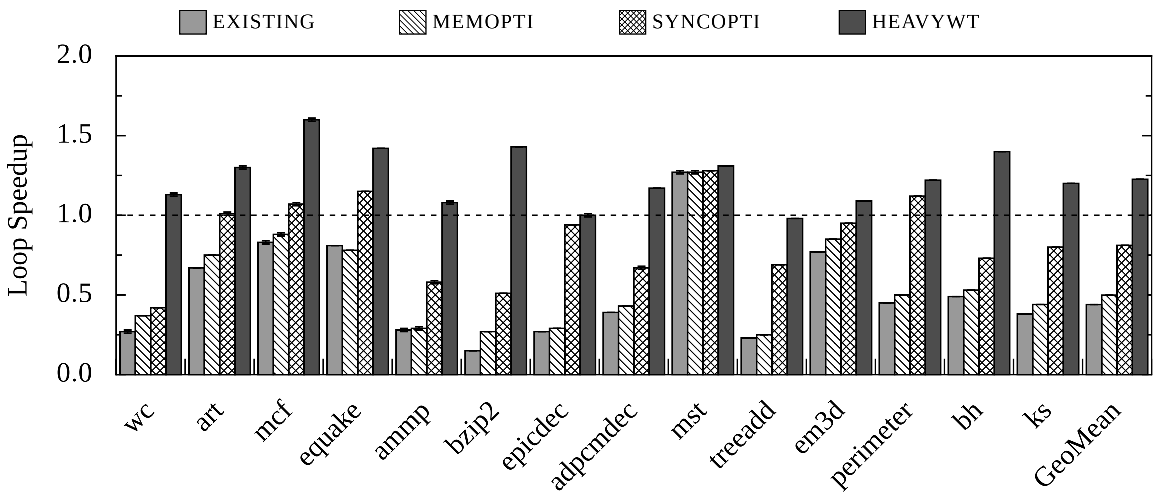
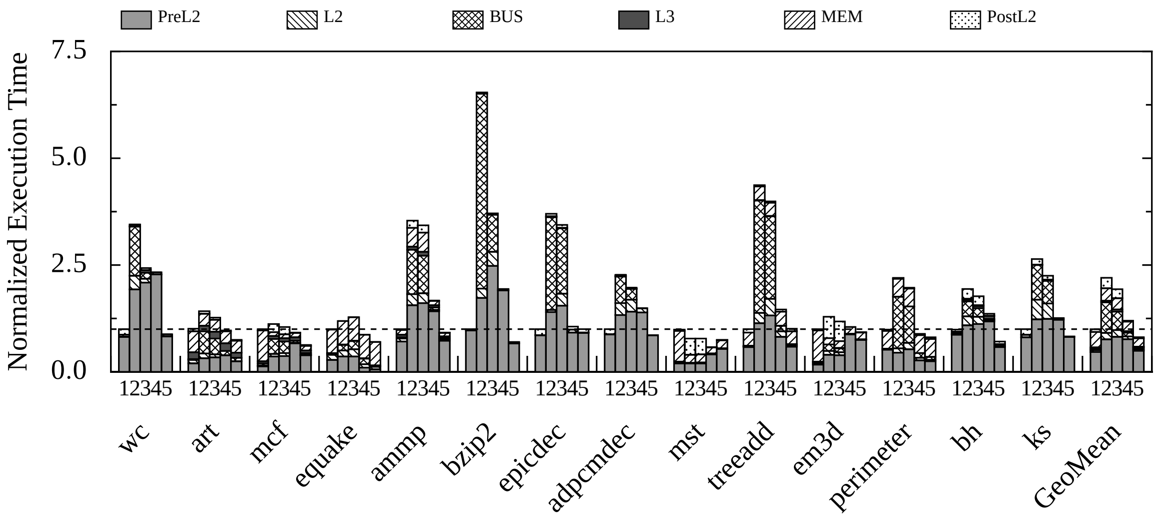


Figure 6.2: Overall performance comparison across all design points.

the stall behavior of `produce` and `consume` instructions.

As mentioned in Chapter 4, the component-wise breakdowns represent non-overlappable stalls that contributed directly to the dynamic schedule height in the respective sections of the machine. Since the study of the various communication mechanisms focuses primarily on the memory subsystem, the component-wise breakdown used in this and next chapters aggregates stalls before and after the memory access stages of the pipeline into *PreL2* and *PostL2* components respectively and zooms into the memory system performance by splitting it into four components - *L2*, *BUS*, *L3* and *MEM*. *L2*, *L3* and *MEM* represent the time spent in the L2, the L3 and the main memory respectively, *BUS* is the total time spent on the shared bus (including arbitration, snoops, requests, and data transfers).

The figures show that HEAVYWT and SYNCOPTI perform better than MEMOPTI and EXISTING for all benchmarks. It is obvious from design why HEAVYWT is the best overall (since it provides the lowest COMM-OP delay). SYNCOPTI closely trails HEAVYWT across all the benchmarks. This is expected since SYNCOPTI and HEAVYWT are identical in all respects, except in their queue backing stores. However, there is still a considerable difference between the SYNCOPTI and HEAVYWT bars across all benchmarks and in fact the difference is pretty significant in `wc`. A careful ex-



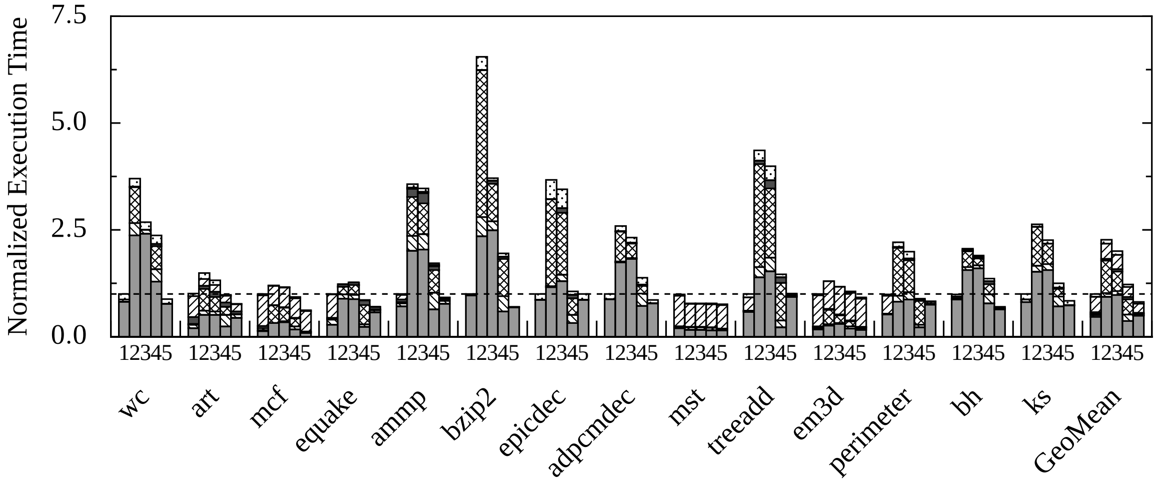


Figure 6.3: Normalized execution time breakdown for producer (above) and consumer (below) threads for each design point.

(1 = SINGLE-THREADED, 2 = EXISTING, 3 = MEMOPTI, 4 = SYNCOPTI, 5 = HEAVYWT)

amination of the pipeline behavior of these benchmarks revealed the main reason for the slowdown. The average `consume`-to-use latency in SYNCOPTI is at least 6 cycles (since synchronization happens in L2 following a 2-cycle stream address generation), whereas it is 1 cycle in HEAVYWT. The higher COMM-OP delay results in the consumer performing slower in SYNCOPTI than in HEAVYWT. This in turn delays freeing up of queue slots, thereby ultimately slowing down the producer thread. For `wc`, the reason why SYNCOPTI is almost twice as slow as HEAVYWT is because the streaming loop is very tight. With three consume operations per loop iteration, the overhead turns out to be a significant factor.

While HEAVYWT incurs no memory system overhead (by design), SYNCOPTI does equally well too, as can be seen by the *L2* and *BUS* components. Since synchronization counters are efficiently maintained and updated in a distributed fashion, SYNCOPTI avoids unnecessary cache line ping-ponging between cores; only the producer side writes to a cache line, while the consumer reads the cache line. The only extra memory traffic stems from uni-directional queue line transfers and bulk acknowledge notifications for synchronization counter updates. However, since MEMOPTI and EXISTING have to explicitly modify condition variables and communicate them in both directions, their memory system performances are significantly poorer. Since SYNCOPTI, MEMOPTI and EXISTING all effect data transfers through the memory subsystem, one might expect their breakdowns to be somewhat similar. However, that is not the case, because, in MEMOPTI and EXISTING, instructions recirculate through the OzQ[1] when they cannot issue because of port contention or to respect memory fence semantics. Further, when a produce operation tries to produce into a full queue, the spin lock instructions keep flowing through the pipeline till the produce happens. Whereas, in SYNCOPTI, a `produce` instruction takes up one OzQ slot and remains dormant till it goes past the synchronization phase in its state machine. Often, this causes the OzQ to fill up leading to back-pressure in the pipeline, resulting in a

[1]An ordered queue of outstanding transactions, in the Itanium 2's L2 controller, whose entries also serve as miss status holding registers (MSHRs).

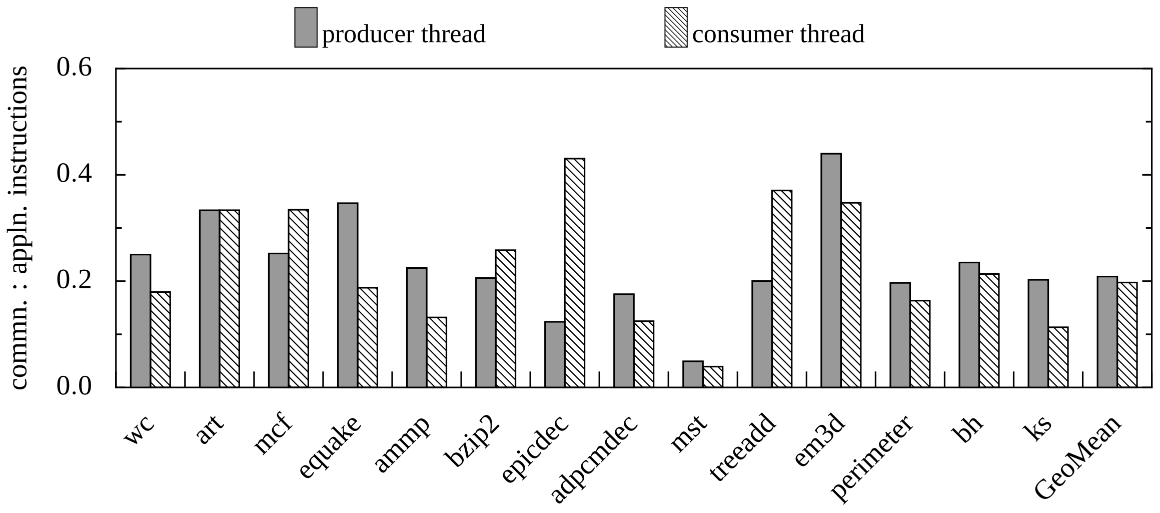


Figure 6.4: Ratio of the # of dynamic communication instructions to application instructions for producer and consumer threads.

larger *preL2* component. Finally, the greater intrinsic schedule height for software queues, causes MEMOPTI and EXISTING to have larger *postL2* components than SYNCOPTI since fewer instructions execute and writeback in SYNCOPTI. Hence the differences in the breakdowns.

MEMOPTI performs better than EXISTING overall due to timely transfer of cache lines from one core to another. This can be seen from the reduced *BUS* components for all benchmarks for MEMOPTI, since the pre-emptive forwarding in MEMOPTI minimizes the number of critical path cycles spent on bus transfers, as was the case with the on-demand bus transfers in EXISTING.[2]

Overall, a major factor contributing to the improved performance of HEAVYWT and SYNCOPTI over MEMOPTI and EXISTING is the *postL2* component. MEMOPTI and EXISTING simply commit many more instructions due to the software overhead for synchronization and address generation and this directly causes them to perform worse than HEAVYWT and SYNCOPTI. Figure 6.4, which has a plot of the ratios of the dynamic counts of communication and synchronization instructions to application instructions for

[2]This result is slightly at odds with the result reported in [50], wherein, for a lot of benchmarks, EXISTING performed better than MEMOPTI, due to differences in the implementation of write-forwarding.

both the producer and consumer threads for codes with `produce-consume` instructions, shows that on the average, a communication is required once every 5 to 20 dynamic application instructions. Given this high communication frequency, the 10 instruction software queue sequence, required per communication, proves to be a significant overhead and detrimentally affects software queue performance. The performance of SYNCOPTI is in between that of the HEAVYWT mechanism and the EXISTING and MEMOPTI mechanisms. As seen in Figure 6.2, SYNCOPTIhas 1.56X and 2.1X speedup over MEMOPTI and EXISTING respectively and a relatively modest 38% slowdown relative to HEAVYWT. This, however, is not good enough since the speedup of HEAVYWT over single-threaded codes for the optimized loops is 25%. This means the communication overhead for the other mechanisms actually *negates* parallelization benefits and causes multithreaded execution to perform *worse* than single-threaded execution. These basic experiments demonstrate the importance of efficient communication support for high-frequency pipelined inter-thread communication.

6.4 Sensitivity Study

In order to evaluate the sensitivity of the four techniques to increased wire delays of future CMPs, the basic performance experiments were repeated with a bus latency equal to 4 CPU cycles. For HEAVYWT, the end-to-end latency of the dedicated interconnect was increased to 4 cycles as well. The execution time breakdown is presented in Figure 6.5 and is normalized to single-threaded execution with the same 4 cycle bus latency. EXISTING, MEMOPTI and SYNCOPTI are all affected by the increased bus latency and have significantly larger *BUS* components. This is not a surprise, since these communication mechanisms use the bus to transfer inter-thread streaming data and any increase in bus latency will consequently have a negative impact on the overall performance. Given that it takes 8 ($\frac{linesize(128)}{buswidth(16)}$) bus cycles for a line to be transferred on the bus, with a bus latency

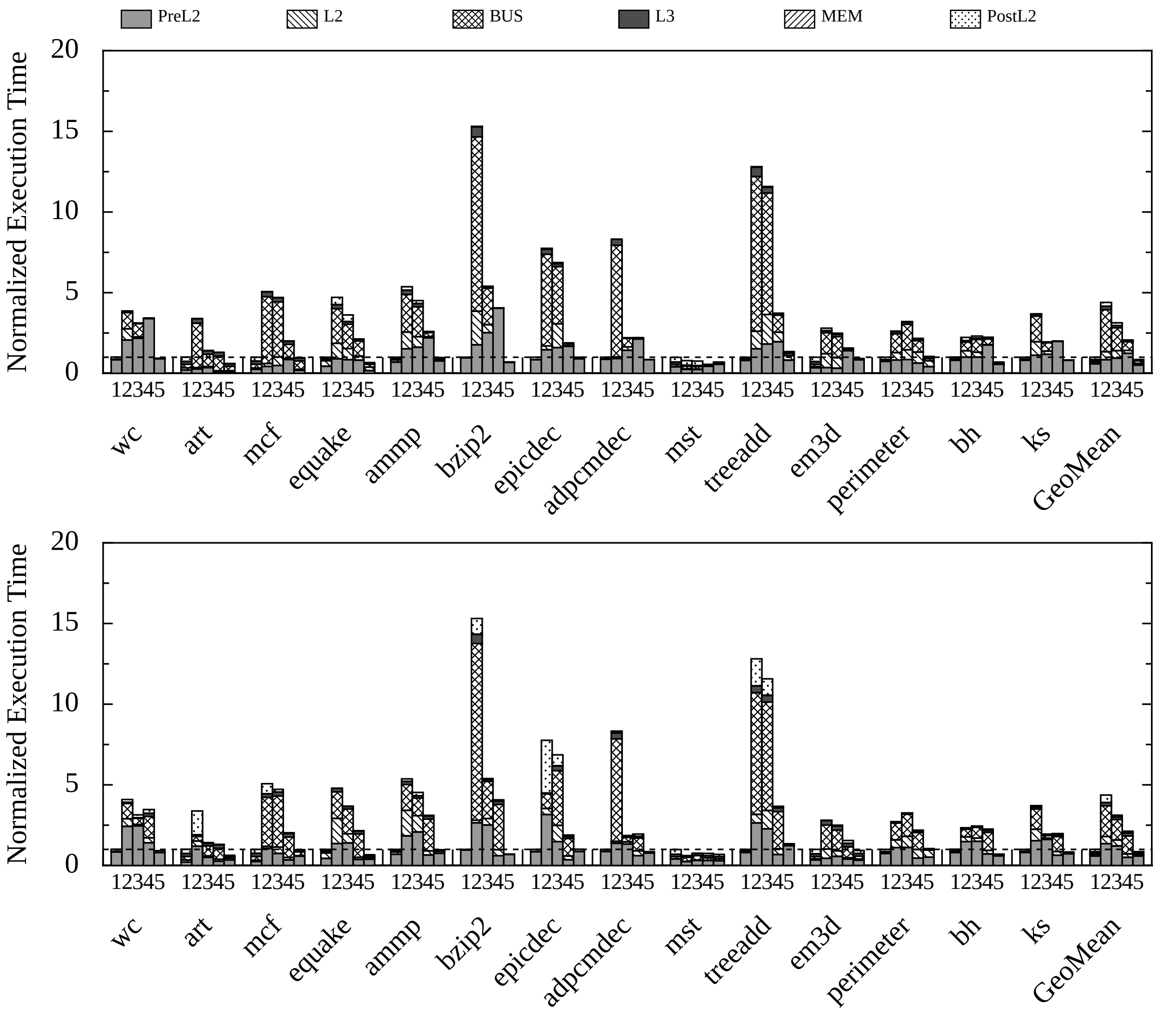


Figure 6.5: Normalized execution time breakdown with transit delay = 4 cycles.

(1 = SINGLE-THREADED, 2 = EXISTING, 3 = MEMOPTI, 4 = SYNCOPTI, 5 = HEAVYWT)

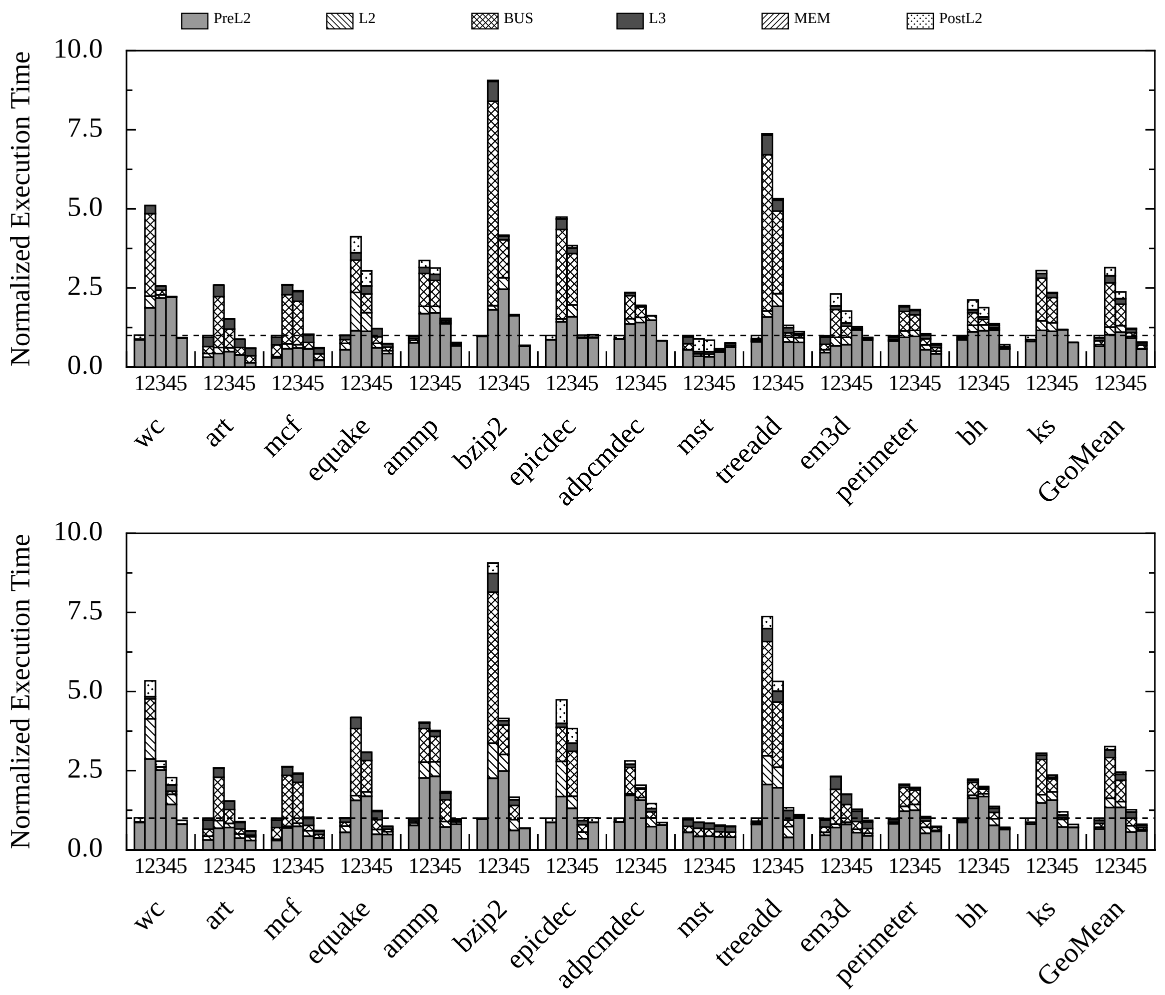


Figure 6.6: Effect of increased increased interconnect bandwidth (transit delay = 4 cycles, bus width = 128 bytes).

(1 = SINGLE-THREADED, 2 = EXISTING, 3 = MEMOPTI, 4 = SYNCOPTI, 5 = HEAVYWT)

of 4 CPU cycles, it takes 32 CPU cycles for line transfers. This not only causes increased time for line transfers but also causes requests to backlog leading to large arbitration delays within cores. Further, benchmarks like `mcf` and `equake` are memory-intensive applications and tend to access the L3 cache frequently, making them sensitive to bus delays.

To see if interconnect bandwidth is the problem, another set of experiments was run with a bus width of 128 bytes (equal to cache line size) holding the latency at 4 CPU cycles (peak bandwidth of 32 bytes per cycle). This change significantly eases contention leading to lower arbitration delays as seen from the *BUS* components in Figure 6.6. This highlights the importance of *interconnect bandwidth* for high-frequency streaming. Although building a 128-byte-wide interconnect can be expensive, the same benefits can be had by using a pipelined interconnect with equal bandwidth.

6.5 Summary

To conclude, the key observations from the performance evaluation in this chapter are:

- The main source of performance loss for software queues is the huge instruction overhead required to implement synchronization and queue pointer update in software for every communication operation. This is chiefly responsible for the poor performance of EXISTING and MEMOPTI with respect to SYNCOPTI and HEAVYWT.
- Even though SYNCOPTI performs better than EXISTING and MEMOPTI, HEAVYWT performs better than SYNCOPTI because of its smaller `consume`-to-use delays.

The next chapter will present alternate designs that will seek to alleviate the bottlenecks identified in this chapter.

Chapter 7

Communication Support Optimizations

This chapter develops optimizations that seek to address the main reasons for performance loss identified in Chapter 6. Two types of optimizations are proposed and evaluated.

The first is a software-only optimization that minimizes synchronization and queue pointer update overhead for shared memory software queue implementations. The second comprises simple hardware enhancements to snoop-based synchronization (i.e. SYNCOPTI) that enable it to almost equal the performance of the synchronization array (i.e HEAVYWT) by reducing `consume`-to-use latency, albeit without the associated hardware and OS costs.

7.1 Amortizing Overhead Costs for Software Queues

As discussed in Section 3.2, software queues obviate the need for special hardware support for inter-thread communication. Further, no OS modification is necessary. The default memory consistency and cache coherence implementation of any machine provide the complete hardware support needed for software queue based communication. However, they come with a heavy performance cost. As shown in the previous chapter and summarized at the beginning of this chapter, software queue implementations suffer from heavy instruction overhead required to implement synchronization and queue pointer update functionality. Even highly tuned code sequences have significant recurring intra-thread overhead and tend to negate any benefits from PMT parallelization. While synchronization and queue pointer update are necessary evils which cannot be wished away entirely, the optimization presented in this section seeks to coalesce such overhead instructions into one per *group* of queue accesses instead of one per *individual* queue access, so as to amortize the synchronization and queue pointer update overhead across multiple queue accesses. The optimization comprises of a compiler analysis to automatically identify queue operations for which synchronization and queue pointer update can be coalesced and a subsequent code generation pass to use analysis information to actually perform the coalescing. The analysis is implemented as a pass after *autoDSWP* in the VELOCITY compiler framework. The analysis was originally presented in [49] without any run-time evaluation results. In addition to presenting run-time results for the original algorithm, this section also presents a new algorithm to perform coalescing across larger code regions.

AutoDSWP implementations can either use a unique queue to handle each inter-thread dependence or may choose to merge two or more dependencies into a single queue. Each has its pros and cons. While the former leads to queue addressability concerns, the latter constrains the scheduler by requiring the order of queue accesses be strictly the same in all communicating threads. The analysis described in this section assumes a queue per dependence implementation in this chapter. Further, it also assumes finite-sized queues with equal number of entries in each queue.

Variants of the above approaches often use multiple queue buffers or coarser-grained signaling to minimize cache line ping-ponging or amount of storage used for synchronization or both. Regardless, the main drawback of software queues is that the code sequences to produce and consume a single datum are quite lengthy. A naïve use of any of the above software queue implementations to handle communication in DSWP by reproducing the entire code sequence (comprising synchronization, data transfer and queue pointer update instructions) for every single communication operation to each queue will naturally lead to performance inefficiencies. However, in code regions with two or more accesses to "parallel" queues, the overhead arising from synchronization and queue pointer update instructions can be amortized across multiple accesses. Coalescing synchronization leads to an increase in critical section size, which may be unacceptable in most conventional scenarios. However, since DSWP pipelines communication and synchronization, increasing the critical section size will at worst manifest itself as a *one-time* increased pipeline fill cost. The next subsection expands upon this intuition and presents an analysis to automatically identify parallel queue accesses for which synchronizations and queue pointer updates can be coalesced.

7.1.1 Analysis

First, an intuitive algorithm is described to determine which queue accesses to coalesce synchronization for. This algorithm is progressively refined to take into account various correctness constraints. The notation `ACQUIRE n` and `RELEASE n` will denote acquire and release operations for a condition variable n. The term *synchronization number* will be used to abstractly refer to a condition variable. The analysis operates on a machine-independent intermediate representation (IR). While the exact code sequence for an `ACQUIRE n` or `RELEASE n` operation may vary slightly depending on whether it synchronizes a produce or a consume operation, for the discussion below, it suffices to know that `ACQUIRE n` is a spinlock loop which will prevent the enclosing thread from making forward progress until the spinlock succeeds. Since coalescing synchronization is inherently more complex than coalescing queue pointer updates, the analysis is driven from a synchronization standpoint. Section 7.1.2 will explain how queue pointer update coalescing can be piggybacked on synchronization coalescing during code-generation.

A *synchronization equivalence group (SEG)* is defined as a group of queues for which synchronization can be coalesced. To start with, only innermost loops in 2-thread DSWP will be considered. For pipelined communication between two innermost loops, it can be intuitively seen that by acquiring and releasing synchronization at the beginning and end of the loop body respectively, all communication operations in the loop body can be correctly synchronized. However, in a two-deep loop nest that has been DSWP'ed such that there are communication operations in both the outer and inner loops, the above simple strategy will no longer work. All synchronization cannot be coalesced at the outer loop boundaries as that will leave communication operations in the inner loop without any synchronization. In Figure 7.2, notice that there is no `ACQUIRE n` or `RELEASE n` operation for queue accesses in the inner loop. This can cause produce operations to run over the allotted buffer space or consume operations to prematurely read stale data. Thus, upon coalescing, `ACQUIRE` and `RELEASE` operations for a queue access can move to a less restrictive control flow condition (like outside an "if" statement), but cannot be hoisted out of their original loops. This condition ensures that every dynamic queue access operation is guarded by at least one `ACQUIRE n` and one `RELEASE n` operation.

But this condition is *not* sufficient. For example, in Figure 7.3, synchronization is acquired and released at the beginning and end of each loop nest level. Even though this satisfies the condition stated above, it fails to provide correct synchronization because, assuming a queue to contain 32 entries, the producer thread's `ACQUIRE 1` will spinlock trying to produce beyond 32 queue items. However, the consumer thread will spinlock in `ACQUIRE 0` in its outer loop, since the producer's outer loop will never get a chance to execute its synchronization release, `RELEASE 0`. Since the consumer is spinlocking in `ACQUIRE 0`, it will never reach its inner loop, thus leading to a deadlock.

Figure 7.4 highlights another potential pitfall if `ACQUIRE` and `RELEASE` operations for a synchronization number are not control-equivalent. In this example, if the loop were to proceed down the "if" path, the `RELEASE` operation for synchronization number 0 would never execute. This in turn would cause the consumer thread to spin loop in its `ACQUIRE` operation and prevent it from making any progress.

To summarize, the necessary and sufficient conditions for correct synchronization coalescing are:

1. For each synchronization number n, dynamically, there be a many-to-one or one-to-one mapping of synchronization operations (`ACQUIRE`s and `RELEASE`s) to queue accesses, for each queue in its $SEG(n)$[1].

2. There be no circular inter-thread dependence among overlapping critical sections in any thread.

3. For each synchronization number, dynamically, there be a one-to-one correspondence between `ACQUIRE`s and `RELEASE`s.

To satisfy all the above conditions, a single-entry single-exit acyclic region with control equivalent entry and exit points called a *loop region* is defined. `ACQUIRE` and `RELEASE` operations for all queue accesses in this region can be coalesced at the region entry and exit points. The *acyclic* clause satisfies conditions 1 and 2 and the *control equivalence* clause satisfies condition 3.

The first step of the analysis is to form loop regions. Initial loop region entries are defined as points in a loop's static CFG where control flow is transferred *into* the loop, including from inner loops and fall-through from loopback branches. Similarly, initial loop region exits are points in a loop's static CFG where control flow is transferred *out of* the loop, including to inner loops. The source of a loop backedge is a special loop region exit and likewise a loop header is a special loop region entry.

The algorithm starts by marking initial loop region entries and exits as defined above as entry and exit nodes respectively in the given CFG. Then, for every node in the CFG it computes its latest post-dominator and earliest dominator, taking into account the new entry and exit nodes. All latest post-dominators are marked as exit nodes and all earliest dominators are marked as entry nodes. Source nodes of in-edges into earliest dominators

[1]Note, to take into account coarse-grained signaling implementations, the condition can be modified slightly to ensure a many-to-one or one-to-one mapping with every kth queue access, where k is the signaling granularity.

```
1: Mark Initial Entries and Exits
2: repeat
3:   Find each node's latest post-dominator $L$ and earliest dominator $E$
4:   Mark each $L$ as an exit, mark each $E$ as an entry.
5:   Mark D as entry, $\forall$ $D$, $\forall$ $L$, such that $\overrightarrow{LD}$ is an edge and $L$ is a latest post-dominator
6:   Mark S as exit, $\forall$ $S$, $\forall$ $E$, such that $\overrightarrow{SE}$ is an edge and $E$ is an earliest dominator
7: until (entries and exits do not change for any node)
```

Figure 7.1: Algorithm to find maximal single-entry single-exit loop regions

are marked as exit nodes and destination nodes of out-edges from latest post-dominators are marked as entry nodes. The algorithm iterates till no new entry and exits nodes are marked on the CFG. This procedure yields maximal single-entry single-exit loop regions. The algorithm is summarized in Figure 7.1. The set of queues accessed in each loop region forms an SEG. The analysis also remembers the directionality (i.e. produce or consume) of each SEG. If both produce and consume queue accesses occur in a particular loop region, the analysis groups them into two different SEGs.

Now, all queue operations in a given loop region can theoretically have their `ACQUIRE` and `RELEASE` operations coalesced at the region's entry and exit nodes respectively. However, due to the inter-thread nature of synchronization, it is important to make sure corresponding queue accesses in other threads also fall into the same loop region in those threads. The second step of the analysis compares SEGs across all communicating threads and ensures that for each queue, the SEG it is a member of is exactly the same across all threads accessing that queue. If this is not the case, the corresponding SEGs are split until the condition is true. Once this is done, each globally unique SEG is assigned a globally unique synchronization number.

For example, consider a three-way partitioning of a loop (with no inner loops) wherein thread 1 produces into queues 1, 2, 3 and 4, thread 2 consumes from queues 3 and 4 and produces to queues 5 and 6 and thread 3 consumes from queues 1, 2, 5, and 6. The first step of the analysis creates SEG [1,2,3,4] for thread 1, SEGs [3,4] and [5,6] for thread 2, and SEG [1,2,5,6] for thread 3. The second step of the analysis will split the SEGs such that

```
        Producer Thread              Consumer Thread

Outer:  ACQUIRE  0           Outer:  ACQUIRE 0
        produce [4] = r5             consume r5 = [4]
Inner:  produce [7] = r6     Inner:  consume r6 = [7]
        r6 = r6 + 1                  r6 = r6 + 1
        br r6 < 100, Inner           br r6 < 100, Inner
        r5 = r5 + 1                  r5 = r5 + 1
        produce [9] = r5             consume r5 = [9]
        RELEASE 0                    RELEASE 0
        br r5 < 100, Outer           br r5 < 100, Outer
```

Figure 7.2: Coalescing at the outer loop.

```
        Producer Thread              Consumer Thread

Outer:  ACQUIRE  0           Outer:  ACQUIRE 0
        produce [4] = r5             consume r5 = [4]
Inner:  ACQUIRE  1           Inner:  ACQUIRE 1
        produce [7] = r6             consume r6 = [7]
        r6 = r6 + 1                  r6 = r6 + 1
        RELEASE 1                    RELEASE 1
        br r6 < 100, Inner           br r6 < 100, Inner
        r5 = r5 + 1                  r5 = r5 + 1
        produce [9] = r5             consume r5 = [9]
        RELEASE 0                    RELEASE 0
        br r5 < 100, Outer           br r5 < 100, Outer
```

Figure 7.3: Coalescing at loop entry and exits.

thread 1 has [1,2] and [3,4], thread 2 has [3,4] and [5,6], and thread 3 has [1,2] and [5,6]. Now, there are three globally unique SEGs - [1,2], [3,4], and [5,6] and they are assigned synchronization numbers 0, 1, and 2 respectively.

The first step of the analysis is a local analysis. The second step has to make a pass over all threads to determine the correct mapping from each synchronization number to its SEG and is a global analysis. For each procedure, the analysis outputs the synchronization numbers used in the procedure, the direction of synchronization (i.e. produce or consume), the `ACQUIRE` and `RELEASE` points for each synchronization number and its SEG.

```
        Producer Thread               Consumer Thread

Loop:   ACQUIRE  0           Loop:    ACQUIRE 0
        br r4 == r5, If               br r4 == r5, If
        produce [4] = r5              consume r5 = [4]
        RELEASE  0                    RELEASE  0
        br Loop                       br Loop
If:     br r5 > 0, Loop      If:      br r5 > 0, Loop
```

Figure 7.4: Control inequivalent ACQUIRE and RELEASE.

7.1.2 Code Generation

The code generation phase first creates memory locations for all synchronization numbers handed out. Then, for each procedure, for each synchronization number used in that procedure, it uses analysis information to insert `ACQUIRE`s and `RELEASE`s for that synchronization number at the specified points. It uses direction information for each synchronization number to determine the exact condition variable values (or occupancy counter operations) to use while generating the `ACQUIRE`s and `RELEASE`s. Either concurrently or as a later pass, `produce` and `consume` instructions can be converted into `store` and `load` instructions to memory locations.

A condition variable based synchronization scheme is used as the baseline software queue implementation in this dissertation. The data layout of these per-entry condition variables were similar to queue data layouts (i.e. each condition variable also took up 8 bytes and had 64 entries corresponding to each queue entry). This layout enabled the code generator to coalesce queue pointer updates for all SEGs, by doing queue pointer updates only for the condition variable queue corresponding to each synchronization number, and using that offset as the offset for operand queue accesses as well.

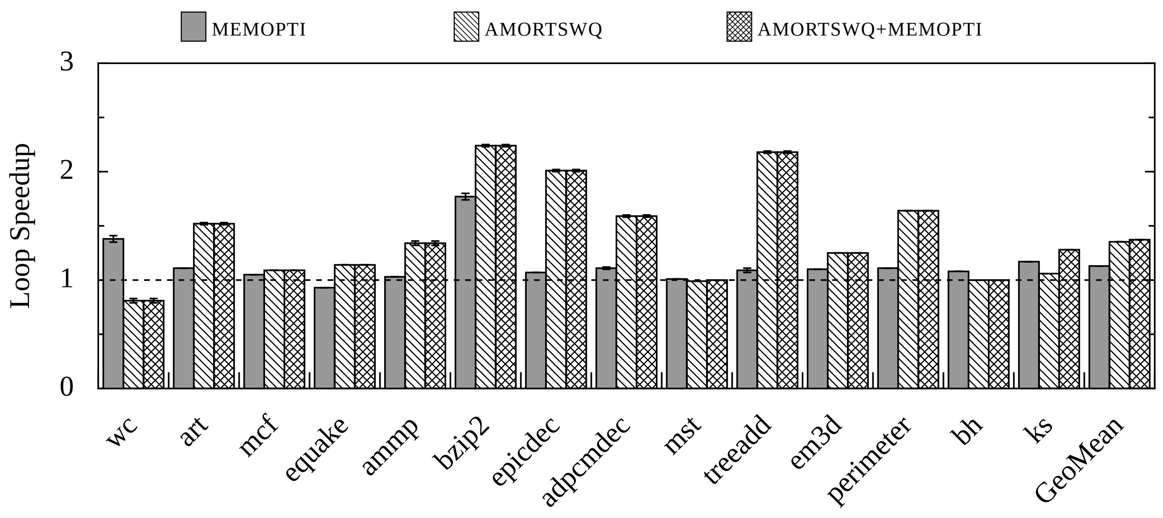


Figure 7.5: Performance of software queues before and after overhead amortization, with and without write-forwarding support.

7.1.3 Evaluation

The performance of software queues with synchronization and queue pointer update coalescing relative to naïvely implemented software queues is shown in Figure 7.5. The baseline for this comparison is EXISTING. The geometric mean speedup of amortized software queues over EXISTING is 38%. With write-forwarding support, the difference between the two techniques narrows down to 26%, as can be seen from a comparison of the performance of AMORTSWQ+MEMOPTI and MEMOPTI.

7.1.4 Discussion

As can be seen from the performance results, synchronization and queue pointer update coalescing succeed in amortizing the instruction overhead of software queues. That being said, there is still a lot of room for performance improvement through overhead amortization. First, the current implementation always tries to coalesce synchronization and queue pointer updates at the least restrictive control flow condition level, so as to cover as many communication operations as possible. However, if a communication operation executes only in a more restrictive condition and there are very few communication operations, then

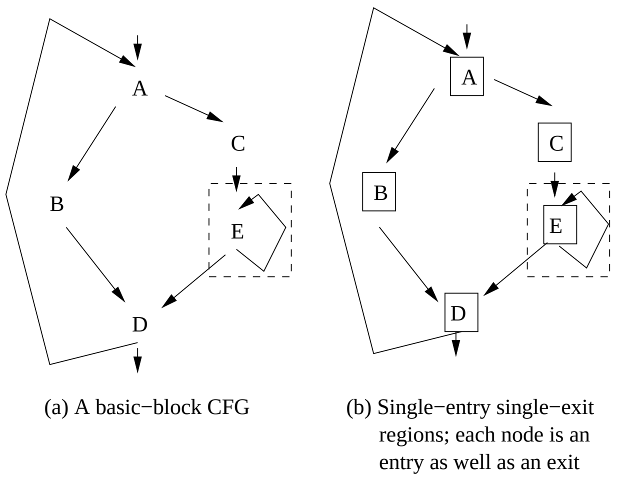

A
B
C
D
E
(a) A basic−block CFG
(b) Single−entry single−exit regions; each node is an entry as well as an exit

Figure 7.6: Example illustrating how the single-entry single-exit algorithm fractures acyclic regions in the presence of side-exits and side-entrances.

the above algorithm may actually cause the program to execute more instructions by acquiring and releasing synchronization for loop iterations when the communication operations themselves would not have executed. In such cases, the naïve software queue implementation would execute fewer instructions and perform better. Therefore, the technique presented above, is at best, a heuristic. More work is necessary to make the analysis aware of such issues and perform coalescing only when it is absolutely profitable.

Another drawback of the analysis technique presented above is that it does not guarantee maximal loop regions for overhead amortization. It only guarantees maximal *single-entry single-exit* loop regions. The presence of side-exits causes the analysis to quickly degenerate to a much simpler basic block level coalescing algorithm. For example, applying the above analysis to the loop, whose basic block control flow graph is shown in Figure 7.6a will result in the loop regions shown in Figure 7.6b. The loop header, and hence an initial entry, is node A. The loop exits through the conditional branch in node D and hence D is an initial exit. The loop regions are indicated by solid bounding boxes. The dashed box around the inner loop E indicates that sources of any incoming arcs are initial exits and destination nodes of any outgoing arcs are initial entries. Thus, D starts off as an entry node along side A, thanks to the $\overrightarrow{ED}$ edge. Likewise, node C starts off as an

```
1: Mark Initial Entries and Exits
2: $\forall\, n$, $Entry[n] = \begin{cases} n & \text{if } n \text{ is an initial entry} \\ \cup & \text{otherwise, where } \cup \text{ is the universal set} \end{cases}$
3: **repeat**
4:     $\forall\, n$, $Gen[n] = \begin{cases} n & \text{if } n \text{ is an entry} \\ \{\} & \text{otherwise} \end{cases}$
5:     $\forall\, n$, $Kill[n] = \begin{cases} n & \text{if } n \text{ is an exit} \\ \{\} & \text{otherwise} \end{cases}$
6:     $\forall\, n$, $Entry[n]$ = $(Gen[n] - Kill[n]) \wedge \bigwedge_S$, $\forall\, n$, where $S$ = $Entry[i]$, $\forall\, i$ such that $\overrightarrow{in}$ is an
       edge and $\wedge$ is the set intersection operator
7:     **if** $Entry[n]$ is $\{\}$ **then**
8:         Mark $n$ as a new loop region entry
9:     **end if**
10: **until** ($Entry[n]$ does not change for any node $n$)
11: Group all nodes $n$ into a loop region with entry $E$ such that, $Entry[n]$ equals $E$
12: Mark $\overrightarrow{nd}$ as loop region exit edge for a region with entry $E$, if $\overrightarrow{nd}$ is an edge such that $Entry[d]$
    is not equal to $E$.
```

Figure 7.7: Algorithm to find maximal single-entry multiple-exit loop regions

initial exit in addition to node D. This causes node A to become its latest post-dominator. Next, B, having an incoming edge from a latest post-dominator is marked as an entry node. Further, since node D is its earliest dominator, the sources of its incoming edges $\overrightarrow{BD}$ and $\overrightarrow{ED}$ must be marked as exits. In this way, every node becomes an entry and an exit and will serve as acquire and release points for communication operations, if any, in the respective blocks. In this extreme example, the single-entry single-exit regions detected by the analysis degenerate to basic blocks. This drastically reduces the scope for synchronization coalescing. For example, communication operations in basic blocks A and B cannot be coalesced as each of them are their own loop regions.

Ideally, synchronization should be acquired upon entry into a maximal acyclic region and released when going from one maximal acyclic region to another. A simple dataflow analysis to detect such maximal regions is shown in Figure 7.7. This procedure detects maximal single-entry multiple-exit regions. Note that the region exits are provided in the form of edges. In order to be able to insert synchronization release operations correctly along all region exits, it is necessary to split the region exit edges and add new "exit" control

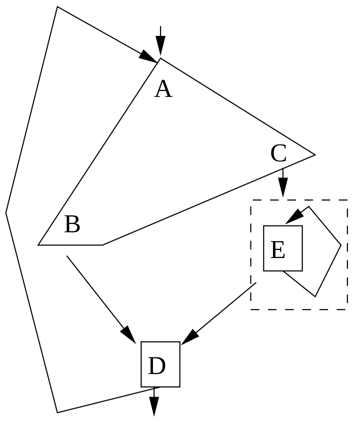


Figure 7.8: Application of the single-entry multi-exit algorithm to the loop from Figure 7.6. There are three single-entry multi-exit regions - region ABC with node A as entry and edges $\overrightarrow{BD}$ and $\overrightarrow{CE}$ as exit edges, region D with node D as entry and the outgoing edge from D as exit edge and finally region E with node E as entry and edges $\overrightarrow{EE}$ and $\overrightarrow{ED}$ as exit edges. The exit edges will need to be split during code generation in order to insert release operations.

blocks along the split edges. While this approach guarantees maximal coalescing, the extra control blocks created due to edge splitting, can detrimentally affect instruction scheduling. Run-time performance evaluation for the single-entry multiple-exit scheme was not performed since static schedule height comparisons showed the single-entry multiple-entry to have longer schedule heights than codes generated with the single-entry single-exit scheme presented earlier in this section, thanks to the new blocks introduced from edge splitting. Figure 7.8 shows the regions formed with the single-entry multi-exit algorithm for the control flow graph shown in Figure 7.6b.

With extra effort, it seems like it would be possible to obtain automatic software queue implementations that deliver high performance by amortizing overhead across multiple queues. Until first class support becomes available for high-frequency streaming inter-core communication, software queues will be the sole means of communicating from one thread to another and will definitely benefit from research along the direction presented in this section. While the performance analysis in this dissertation uses many small application loops, which are responsible for the poor showing of software queues versus other communication mechanisms, for larger loops the overhead due to inter-thread communication may be masked by application instruction execution and techniques such as the one just presented may pave the way for software queues being a really attractive communication mechanism for such applications.

7.2 Hardware Enhancements to Snoop-Based Synchronization

This section describes simple enhancements to SYNCOPTI to bring its performance as close as possible to HEAVYWT. As seen in Section 6.3 of the previous chapter, SYNCOPTI trails the performance of HEAVYWT primarily due to the large `consume`-to-use latency that slowed down the iteration initiation rate of the consumer thread, in turn causing the producer thread to eventually stall. These basic observations motivate two optimizations to SYNCOPTI.

First, in order to avoid frequent producer thread stalls due to queue-full conditions, the queue size is increased to 128 entries (up from 32 entries), and increased the QLU to pack 16 8-byte queue items per cache line (Q128). While this may seem like a straightforward thing to do, it is important to realize that increasing the queue sizes is a luxury that a dedicated hardware technique like HEAVYWT cannot afford due to the sheer hardware costs. Even for SYNCOPTI, one cannot increase the queue size indefinitely. Queue sizes can grow only so long as the bit width of the occupancy counters (used for synchronization) can represent the queue size. However, since the queue size grows exponentially with respect the bit width of the counters, SYNCOPTI can easily scale up to large queue sizes.

Second, in order to reduce the average `consume`-to-use latency, the use of a special fully associative 16KB stream cache (SC) was evaluated. Improving consumer performance indirectly improves producer performance by avoiding frequent queue full stalls. While the stream cache does add additional storage to each processor core, this storage amounts to only 15% of the storage used for the dedicated queue backing store.

The proposed stream cache works as follows. When cache lines mapped to queues are forwarded from the producer's L2 cache to the consumer's L2 cache, after filling the consumer's L2 cache, the memory address is reverse mapped to a queue address (a two-tuple of queue number and queue slot) that is used to fill into the stream cache. By using a different address space, `consume` instructions are now able to access queue data, without going through TLB lookup, memory address generation, etc. Stream cache entries are invalidated by `consume` instructions that hit. If the stream cache is full, then fill requests to the stream cache are ignored. In this modified SYNCOPTI design, `consume` instructions continue to go to the L2, even if they are serviced by the stream cache, to ensure the synchronization counters are updated and the producer core is informed of these updates. If a `consume` instruction misses in the stream cache, then it is handled in the L2, just as it was in the original snoop-based synchronization design.

Note that this optimization requires stream address generation logic in the processor pipeline (akin to HEAVYWT) to rename `consume` instructions to the correct queue addresses to index into the stream cache. However, this is still better than HEAVYWT, since SYNCOPTI shares the L3 bus, while HEAVYWT requires extra interconnects connecting the cores to the synchronization array, which can be expensive [29].

Both these optimizations were evaluated in isolation and together. Figure 7.9 presents the breakdown for the producer (above) and consumer (below) cores. From right to left, SYNCOPTI$_{\mathrm{Q128}}$ improves the producer by reducing stalls (smaller *preL2*) and improves the consumer by providing improved cache locality (smaller *L2*) through a denser queue layout. Next, SYNCOPTI$_{\mathrm{SC}}$ lowers `consume`-to-use latency and improves the performance of both cores. SYNCOPTI$_{\mathrm{SC+Q128}}$ combines the benefit of both by further reducing stalls in the producer and lowering the consumer's *L2* component. It is able to achieve per-

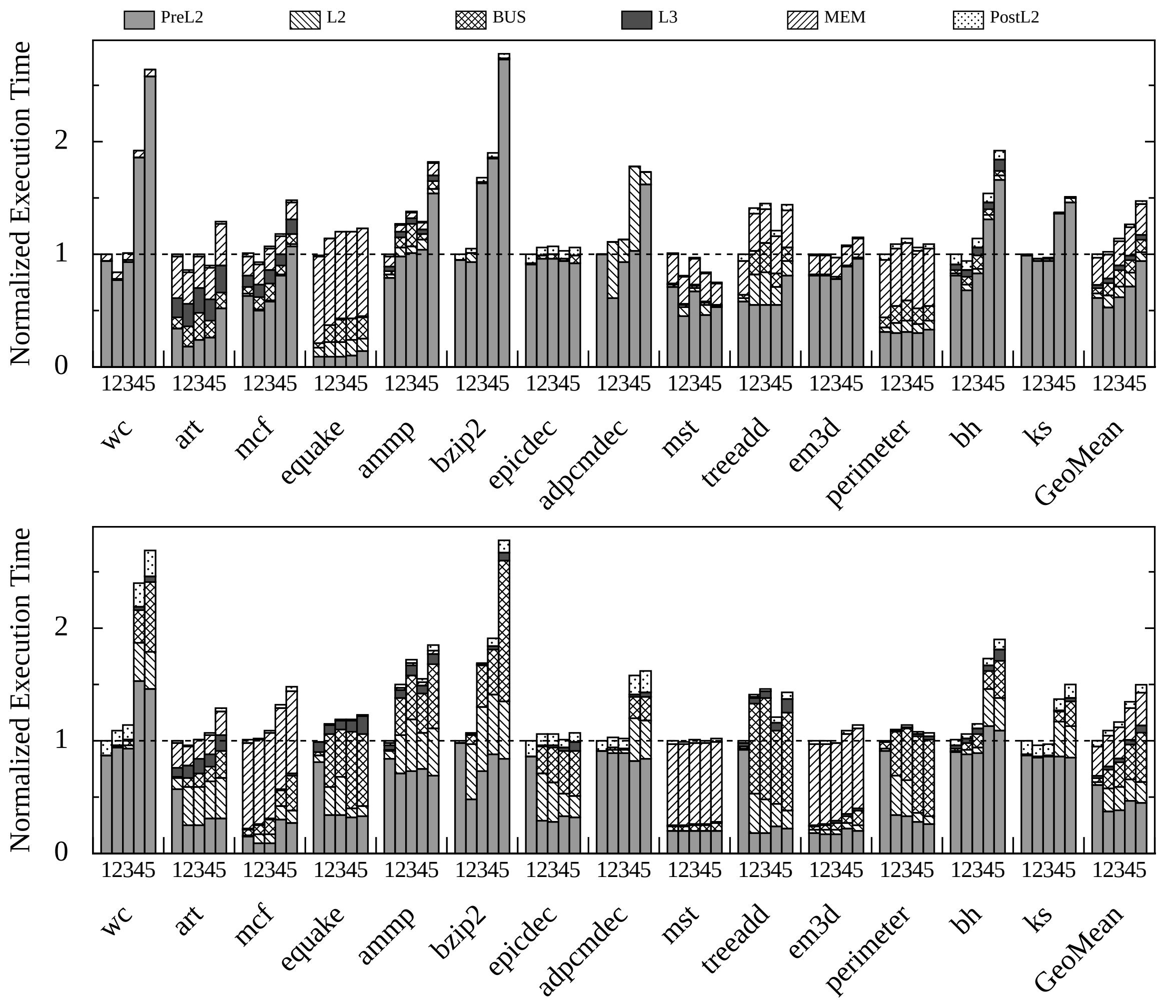


Figure 7.9: Effect of streaming cache and queue size on producer (top) and consumer (bottom).

(1 = HEAVYWT, 2 = SYNCOPTI$_{SC+Q64}$, 3 = SYNCOPTI$_{SC}$, 4 = SYNCOPTI$_{Q64}$, 5 = SYNCOPTI)

formance equaling HEAVYWT at times even performing better, achieving a 2X speedup over EXISTING and MEMOPTI mechanisms and bridging the gap with HEAVYWT to just 8%.

7.3 Summary

This chapter introduced and evaluated two communication support optimizations.

The first is synchronization coalescing, a static analysis and code-generation technique. It amortizes synchronization and queue pointer update overhead of conventional software queues by coalescing these operations across multiple parallel queue accesses. When applied with naïve heuristics, this technique has been shown to outperform existing software queue implementations by 38% on the average. Sophisticated heuristics to carefully apply this technique can yield higher performance.

The second is the stream cache, a simple hardware optimization to reduce `consume`-to-use latency in snoop-based synchronization implementations (SYNCOPTI). The use of longer queue sizes was also evaluated. Longer queues along with the stream cache optimization enabled SYNCOPTI to perform almost as fast as the heavy weight techniques represented by HEAVYWT. In practice, a 16KB stream cache was found to be sufficient to yield high performance. The main benefit with a cache design is that, unlike a dedicated primary storage design (such as the synchronization array), a cache design need not be made large enough to handle the worst-case inter-thread traffic scenario and instead, can be made small enough to meet cycle-time requirements. Using the memory hierarchy as backing store allows more flexible queue sizing than dedicated backing store techniques. Further, it obviates the need to save the contents of the stream cache during context-switches since the data in the memory hierarchy is handled automatically by the operating system.

While the experiments presented in this chapter and Chapter 6 did not require very long queues, long queues may be needed to support DSWP programs with non-linear pipelines (refer to Chapter 5). As mentioned in Chapter 3, accesses to long queues may stride through the entire cache repeatedly and can result in poor locality for non-stream cache accesses. This can degrade overall performance. One solution to this problem is to reserve one "way" of large multi-way caches exclusively for stream accesses. This will entail modifying the cache access logic to ensure that all stream accesses go to the reserved way. By confining stream accesses to one way, this strategy can use the remaining ways of the cache to preserve locality for non-stream accesses.

Chapter 8

Conclusions and Future Directions

Multi-core processors or chip multiprocessors (CMPs) represent the biggest paradigm shift in recent years in the microprocessor industry. While these multi-core processors are great for throughput computing, they do not automatically improve the performance of legacy codes, highly sequential codes and single-threaded codes. Exposing thread-level parallelism (TLP) is critical to obtaining continued performance improvements for these applications on multi-core processors. This dissertation is a step in that direction.

Moving to TLP means one has to efficiently handle rising inter-core communication costs, lest they negate any benefits from parallelization. From this standpoint, this dissertation identifies pipelined multithreading (PMT) as an attractive multithreading strategy. In PMT, programs are partitioned into threads such that there are no cycles in the inter-thread dependence graph. Acyclic dependences among PMT threads allows pipelining of inter-thread communication and thus enables PMT programs to tolerate transit delays efficiently.

This dissertation introduced a non-speculative PMT loop transformation called Decoupled Software Pipelining (DSWP). In its simplest form, DSWP partitions a given single-threaded program loop into a critical path loop and an off-critical path loop, which execute as concurrent threads and communicate through a decoupling queue. The decoupling queue insulates the execution of the critical path thread from stalls in the off-critical path thread and vice-versa. This work demonstrates how decoupled execution in DSWP can be leveraged to effectively tolerate variable latency stalls in an application. With detailed cycle-accurate simulation and analysis, DSWP has been shown to tolerate variable latency stalls better than both in-order and out-of-order issue processors.

The dissertation then studied the performance scalability of DSWP as a general-purpose multithreading strategy. It presented a thorough evaluation and analysis of the performance of automatically generated DSWP codes across 2, 4, 6 and 8 threads. Initial results with 32-entry inter-thread queues turned out to be quite counter-intuitive, since most of the benchmarks showed a performance degradation when moving to more threads. Careful analysis revealed that the cause for this degradation was the non-linearity of thread pipelines when moving to greater number of threads. While linear thread pipelines contain a simple linear chain of dependences from one thread to another, non-linear pipelines can have arbitrary acyclic inter-thread dependences. Small queue sizes lead to communication bottlenecks in non-linear pipelines. It was shown that, for non-linear pipelines, the minimum queue size needed for bottleneck-free communication depends not just on the transit delay, but also on the loop iteration times of the threads in the pipeline. This makes it difficult to pre-design hardware with sufficiently long queues.

Under ideal communication (i.e. no bottlenecks due to insufficient queue sizing or due to bandwidth constraints), the performance scalability of DSWP is in line with theoretical expectations. Except for two benchmarks, wherein increased coherence misses caused performance to go down, the performance for all the other benchmarks uniformly increased until the bottleneck SCC was put in a thread of its own and remained steady at the same performance level upon further partitioning. This analysis concluded that the compiler should strive to avoid generating non-linear thread pipelines.

Work presented in this dissertation and elsewhere [8, 14, 17, 66] have established PMT as a viable technique for parallelizing general-purpose programs. However, its success hinges on efficient underlying communication support. While designers are striving [27] to include new, fast interconnects to add value to future CMPs, these enhancements offer little to latency-agnostic PMT programs, which have a streaming communication behavior. To meet the streaming requirements of PMT programs, this dissertation has presented three novel communication mechanisms with contrasting cost-performance tradeoffs - the synchronization array hardware, SYNCOPTI with stream cache, and synchronization coalescing. The first two techniques require ISA enhancements in the form of `produce` and `consume` instructions. The third technique is a software-only technique to automatically generate software queue implementations with amortized overhead.

Synchronization array performs the best, but it requires dedicated hardware, for inter-thread queue storage and inter-thread operand transfer network. It requires a lot of OS support to save queue state on context switches and queue virtualization. SYNCOPTI with stream cache lowers the hardware cost and OS impact of the synchronization array by using the memory subsystem for inter-thread data transfers. Only synchronization is done with hardware counters. Through the design of SYNCOPTI, the dissertation shows that application properties can be successfully leveraged to implement new low-overhead communication primitives with minimal redesign effort, hardware costs, and OS costs, thus presenting an attractive value proposition to designers of future multi-core processors. Finally, even though synchronization coalescing performs the worst of the three, its performance is still approximately 1.4X better than the performance of existing techniques. It incurs no hardware cost and has no OS impact.

The DSWP transformation, which started as a mechanism to tolerate variable latency stalls [51], has been shown to be a general-purpose multithreaded scheduling technique [40]. Going forward, a lot of exciting research possibilities exist in exploring the interaction of DSWP with other scheduling techniques (for example, CMT or IMT techniques), with and without dependence speculation. Speculative PMT transformations will entail design of novel hardware, software and/or hybrid support for misspeculation detection and recovery. Synergistic interactions with single-threaded performance improvement strategies mentioned in Chapter 2 need to be studied and understood.

Incorporating better communication cost models into the automatic DSWP partitioner will enable it to generate highly optimized codes. Careful code duplication may avoid excessive communication without unduly increasing schedule height. Evolving high performance software queue implementations will play a crucial role in the widespread commercial use of DSWP and other PMT techniques. The synchronization coalescing technique will serve as a starting point for such research. Light-weight hardware mechanisms like SYNCOPTI will need to be adapted to provide efficient inter-core streaming communication in non-bus based memory networks.

Early calculations with simple power models reveal that DSWP can achieve substantial energy-delay-product savings through dynamic voltage and frequency scaling. The initial results are definitely exciting enough to warrant a more detailed exploration.

To conclude, the future is bright for research on improving general-purpose program performance on multi-core processors. While there is no doubt that this is a very hard problem, this dissertation has shown that a purely non-speculative technique like DSWP can record significant performance gains. Hopefully, future research can build on this work and take it to the next level by improving its applicability and scalability.

Bibliography

[1] H. Abdel-Shafi, J. Hall, S. V. Adve, and V. S. Adve. An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, pages 204–215, February 1997.

[2] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society Press, 2003.

[3] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

[4] M. M. Annavaram, J. M. Patel, and E. S. Davidson. Data prefetching by dependence graph precomputation. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 52–61, 2001.

[5] R. D. Barnes, E. M. Nystrom, J. W. Sias, S. J. Patel, N. Navarro, and W. W. Hwu. Beating in-order stalls with 'Flea-Flicker' two-pass pipelining. In *Proceedings of the 36th International Symposium on Microarchitecture*, December 2003.

[6] R. D. Barnes, S. Ryoo, and W. W. Hwu. “Flea-Flicker” multipass pipelining: An alternative to the high-powered out-of-order offense. In *Proceedings of the 38th International Symposium on Microarchitecture*, pages 319–330, December 2005.

[7] G. T. Byrd. *Communication Mechanisms in Shared Memory Multiprocessors*. PhD thesis, Department of Electrical Engineering, Stanford University, Stanford, CA, 1998.

[8] E. Caspi, A. DeHon, and J. Wawrzynek. A streaming multi-threaded model. In *Proceedings of the Third Workshop on Media and Stream Processors*, December 2001.

[9] R. S. Chappel, J. Stark, S. P. Kim, S. K. .Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 186–195, May 1999.

[10] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th International Symposium on Computer Architecture*, July 2001.

[11] Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*, July 1999.

[12] A. Cristal, O. J. Santana, M. Valero, and J. F. Martínez. Toward kilo-instruction processors. *ACM Transactions on Architecture and Code Optimization*, 1(4):389–417, 2004.

[13] R. Cytron. DOACROSS: Beyond vectorization for multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 836–884, 1986.

[14] J. Dai, B. Huang, L. Li, and L. Harrison. Automatically partitioning packet processing applications for pipelined architectures. In *Proceedings of the ACM SIGPLAN*

Conference on Programming Language Design and Implementation, pages 237–248, 2005.

[15] M. I. Frank and M. K. Vernon. A hybrid shared memory/message passing parallel machine. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages 232–236. CRC Press, August 1993.

[16] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, January 1995.

[17] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, 2002.

[18] T. Gross and D. O'Halloron. *iWarp, Anatomy of a Parallel Computing System.* MIT Press, 1998.

[19] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, 2000.

[20] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, September 1997.

[21] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of message passing and shared memory in the Stanford FLASH multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50. ACM Press, 1994.

[22] 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*, 01(2), 2001.

[23] M. S. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. C. Burger, S. W. Keckler, and P. Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 12–24, May 2002.

[24] Intel Corporation. *Intel Itanium 2 Processor Reference Manual: For Software Development and Optimization*. Santa Clara, CA, 2002.

[25] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1991.

[26] A. KleinOsowski, J. Flynn, N. Meares, and D. Lilja. Adapting the SPEC 2000 benchmark suite for simulation-based computer architecture research. In *Proceedings of the International Conference on Computer Design*, September 2000.

[27] M. Kobrinsky, B. Block, J.-F. Zheng, B. Barnett, E. Mohammed, M. Reshotko, F. Roberston, S. List, I. Young, and K. Cadien. On-chip optical interconnects. pages 129–142, May 2004.

[28] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B.-H. Lim. Integrating message-passing and shared-memory: Early experience. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 54–63. ACM Press, May 1993.

[29] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 408–419. IEEE Computer Society, June 2005.

[30] M. S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[31] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, December 1997.

[32] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash multiprocessor. *Computer*, 25(3):63–79, 1992.

[33] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 222–233. ACM Press, 1996.

[34] S. F. Lundstorm and G. H. Barnes. A controllable MIMD architecture. In *Proceedings of the International Conference on Parallel Processing*, pages 19–27, 1980.

[35] Y. Luo and L. K. John. Efficiently evaluating speedup using sampled processor simulation. *Computer Architecture Letters*, September 2004.

[36] S. A. Mahlke. *Exploiting Instruction Level Parallelism in the Presence of Conditional Branches*. PhD thesis, University of Illinois, Urbana, IL, 1995.

[37] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood. Coherent network interfaces for fine-grain communication. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 247–258, 1996.

[38] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the*

9th International Symposium on High Performance Computer Architecture, February 2003.

[39] E. M. Nystrom, R. D. Ju, and W. W. Hwu. Characterization of repeating data access patterns in integer benchmarks. In *Proceedings of the 28th International Symposium on Computer Architecture*, September 2001.

[40] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture*, November 2005.

[41] D. A. Padua. Multiprocessors: Discussion of some theoretical and practical problems. Technical Report UIUCDCS-R-79-990, Department of Computer Science, University of Illinois, Urbana, IL, November 1979.

[42] D. A. Penry, M. Vachharajani, and D. I. August. Rapid development of a flexible validated processor model. In *Proceedings of the 2005 Workshop on Modeling, Benchmarking, and Simulation*, June 2005.

[43] M. Pericàs, R. González, A. Cristal, D. A. Jiménez, and M. Valero. A decoupled kilo-instruction processor. In *Proceedings of the Twelfth International Symposium on High Performance Computer Architecture*, February 2006.

[44] D. Poulsen. *Memory Latency Reduction via Data Prefetching and Data Forwarding in Shared-Memory Multiprocessors*. PhD thesis, University of Illinois, Urbana, IL, 1994.

[45] B. R. Preiss and V. C. Hamacher. A cache-based message passing scheme for a shared-bus multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 358–364. IEEE Computer Society Press, 1988.

[46] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A study of slipstream processors. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 269–280. ACM Press, 2000.

[47] R. Rajwar, A. Kagi, and J. R. Goodman. Inferential queueing and speculative push for reducing critical communication latencies. In *Proceedings of the 17th Annual International Conference on Supercomputing*, pages 273–284. ACM Press, June 2003.

[48] U. Ramachandran, G. Shah, A. Sivasubramaniam, A. Singla, and I. Yanasak. Architectural mechanisms for explicit communication in shared memory multiprocessors. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, page 62. ACM Press, 1995.

[49] R. Rangan and D. I. August. Amortizing software queue overhead for pipelined interthread communication. In *Proceedings of the Workshop on Programming Models for Ubiquitous Parallelism (PMUP)*, pages 1–5, September 2006.

[50] R. Rangan, N. Vachharajani, A. Stoler, G. Ottoni, D. I. August, and G. Z. N. Cai. Support for high-frequency streaming in CMPs. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 259–269, December 2006.

[51] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 177–188, September 2004.

[52] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 63–74, December 1994.

[53] A. Roth, A. Moshovos, and G. S. Sohi. Dependence-based prefetching for linked data structures. In *Proceedings of the 8th International Conference on Architectural Sup-*

port for Programming Languages and Operating Systems, pages 115–126, October 1998.

[54] A. Roth and G. S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[55] A. Roth and G. S. Sohi. Speculative data-driven multithreading. In *Proceedings of the 7th International Symposium on High Performance Computer Achitecture*, January 2001.

[56] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[57] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[58] J. E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th International Symposium on Computer Architecture*, pages 112–119, April 1982.

[59] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22th International Symposium on Computer Architecture*, June 1995.

[60] G. S. Sohi and S. Vajapeyam. Instruction issue logic for high-performance interruptable pipelined processors. In *Proceedings of the 14th Annual Symposium on Computer Architecture*, pages 27–34, June 1987.

[61] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P. N. Strenski, and P. G. Emma. Optimizing pipelines for power and performance. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 333–344. IEEE Computer Society Press, 2002.

[62] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The stampede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, 2005.

[63] M. Takesue. Software queue-based algorithms for pipelined synchronization on multiprocessors. In *Proceedings of the 2003 International Conference on Parallel Processing Workshops*, October 2003.

[64] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J. 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 circuit and general-purpose programs. *IEEE Micro*, 22(2):25–35, March 2002.

[65] M. B. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal. Scalar operand networks. *IEEE Transactions on Parallel and Distributed Systems*, 16(2):145–162, February 2005.

[66] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 12th International Conference on Compiler Construction*, 2002.

[67] S. Triantafyllis, M. J. Bridges, E. Raman, G. Ottoni, and D. I. August. A framework for unrestricted whole-program optimization. In *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 61–71, June 2006.

[68] J.-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew. The superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9):881–902, 1999.

[69] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[70] M. Vachharajani, N. Vachharajani, and D. I. August. The Liberty Structural Specification Language: A high-level modeling language for component reuse. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, pages 195–206, June 2004.

[71] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 271–282, November 2002.

[72] S. Vajapeyam and T. Mitra. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. In *Proceedings of the 24th annual International Symposium on Computer Architecture*, pages 1–12. ACM Press, 1997.

[73] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. TurboSMARTS: Accurate microarchitecture simulation sampling in minutes. Technical Report 2004-003, Computer Architecture Lab at Carnegie Mellon, November 2004.

[74] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 84–97, June 2003.

[75] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proceedings of the 35th annual International Symposium on Microarchitecture*, pages 85–96. IEEE Computer Society Press, 2002.