Intelligent Speculation for Pipelined Multithreading

Neil Amar Vachharajani

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
Advisor: David I. August

November 2008

© Copyright by Neil Amar Vachharajani, 2008.

All Rights Reserved

Abstract

In recent years, microprocessor manufacturers have shifted their focus from single-core to multi-core processors. Since many of today's applications are single-threaded and since it is likely that many of tomorrow's applications will have far fewer threads than there will be processor cores, automatic thread extraction is an essential tool for effectively leveraging today's multi-core and tomorrow's many-core processors. A recently proposed technique, Decoupled Software Pipelining (DSWP), has demonstrated promise by partitioning loops into long-running threads organized into a pipeline. Using a pipeline organization and execution decoupled by inter-core communication queues, DSWP offers increased execution efficiency that is largely independent of inter-core communication latency and variability in intra-thread performance.

This dissertation extends the pipelined parallelism paradigm with speculation. Using speculation, dependences that manifest infrequently or are easily predictable can be safely ignored by the compiler allowing it to carve more, and better balanced, thread-based pipeline stages from a single thread of execution. Prior speculative threading proposals were obligated to speculate most, if not all, loop-carried dependences to squeeze the code segment under consideration into the mold required by the parallelization paradigm. Unlike those techniques, this dissertation demonstrates that speculation need only break the longest few dependence *cycles* to enhance the applicability and scalability of the pipelined multi-threading paradigm. By speculatively breaking these cycles, instructions that were formerly restricted to a single thread to ensure decoupling are now free to span multiple threads. To demonstrate the effectiveness of speculative pipelined multi-threading, this dissertation presents the design and experimental evaluation of our fully automatic compiler transformation, *Speculative Decoupled Software Pipelining*, a speculative extension to DSWP.

This dissertation additionally introduces multi-threaded transactional memories to support speculative pipelined multi-threading. Similar to past speculative parallelization ap-

proaches, speculative pipelined multi-threading relies on runtime-system support to buffer speculative modifications to memory. However, this dissertation demonstrates that existing proposals to buffer speculative memory state, *transactional memories*, are insufficient for speculative pipelined multi-threading because the speculative buffers are restricted to a single thread. Further, this dissertation demonstrates that this limitation leads to modularity and composability problems even for transactional programming, thus limiting the potential of that approach also. To surmount these limitations, this thesis introduces multi-threaded transactional memories and presents an initial hardware implementation.

Acknowledgments

First, I thank my advisor David August for his support throughout graduate school. Its tough to believe that I began working with him more than 7 years ago as an undergraduate. His charisma and enthusiasm about research kept me at Princeton as a graduate student and often inspired me through the long PhD process. I am grateful that he was able to insulate me from things such as funding, while simultaneously giving me the opportunity to explore many research topics and exposing to me all attributes of being a researcher. Throughout my tenure in graduate school, he ensured that my work was visible. Whether it was pushing for publication or providing the opportunity to present our group's research at funding meetings, faculty summits, or industrial visits, David worked hard to make me visible in the community. Finally, I would like to thank him for the countless hours of interesting debate, sometimes related to research, but often not. The university environment is fundamentally about open discussion, and our discussions about computer architecture, compilers, voting, politics, renewable energy, etc. were always stimulating.

I would like to thank Sharad Malik and Scott Mahlke for reading this dissertation and providing insightful feedback. Additionally, I thank Andrew Appel and Li-Shiuan Peh for serving on my thesis committee and for their support and feedback that helped make this work what it is today. I would also like to thank the National Science Foundation for supporting my work with a Graduate Research Fellowship.

This dissertation would not have been possible without the Liberty Research Group. The members of group have been great friends and excellent collaborators. This work has significantly benefitted from the infrastructure, the Liberty Simulation Environment and the VELOCITY research compiler, built collectively by the group. It is impossible to imagine how this work would have been possible without the group's dedication to building robust infrastructure. I would also like to thank the "founding" members of the the group, Manish Vachharajani, Spyros Triantafyllis, David Penry, Ram Rangan, and Jason Blome, for creating the close-knit team environment in the group. This environment and the cama-

raderie may have been the most rewarding part of the graduate school experience. I would also like to specifically thank my closest collaborators, Matt Bridges and Ram Rangan. Over the years, they have been great sounding boards and teammates. Many of the ideas described in this dissertation were born and refined through long discussions with them.

I would also like to thank the many friends I've met over the years. Matt Bridges, George Reis, Jonathan Chang, Ram Rangan, Jason Blome, Dan Dantas, Chris Sadler, Adam Stoler, Manny Stockman, Chris Singh, and many more were all instrumental in keeping me sane through the years. All of you made the countless hours in lab entertaining and the precious few outside of lab truly enjoyable. I am grateful to you for reminding me that there was life outside of work, even when a paper deadline (or graduation) was looming. I will certainly miss the regular games of squash, midday excursions to Wild Oats, and hours of Photo Hunt at Winberie's.

As far back as I can remember, my parents have given me unconditional love and support. I'm also grateful that, from a very young age, they emphasized the importance of education and taught me the value of hard work and dedication. Over the years, I have benefitted from their many sacrifices from rearranging their work schedules to make sure I was taken care of as a child to driving my car from New Jersey to Oregon because I had a scheduling conflict. However, I have only seen the obvious sacrifices. My success and happiness were always their priority, and I cannot imagine all the sacrifices they have made to allow me to be where I am today. Any success I have achieved, or will ever achieve, is truly theirs.

Finally, I'd like to thank my brother Manish. This dissertation, and most other things in my life, would not have been possible without his love, support, and leadership. I've always benefitted from watching him blaze a trail ahead of me, allowing me to learn from his successes, and learn even more from his failures. From childhood to adolescence and now in adulthood, he has always been a mentor and a role model. I have never witnessed an ounce of jealousy; he has always selflessly shared with me his experiences and wisdom.

Sir Isaac Newton wrote, “”If I have seen further than other men, it is because I have stood on the shoulders of giants.” Manish has undoubtedly been my giant. I can only hope to live up to his example.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

For years, steadily increasing clock speeds and uniprocessor microarchitectural improvements reliably enhanced performance for a wide range of applications. Figure 1.1 illustrates this historical trend in microprocessor performance by graphing SPECint® [67] scores for all reported configurations across 3 generations of the benchmark suite (SPEC® CPU92, CPU95, and CPU2000).[1] In the figure, the y-axis measures performance (larger numbers indicate faster processors) and is logarithmic. The x-axis denotes time and each point on the graph reflects the SPECint® score for a particular machine configuration that was reported in the given year. As the graph shows, processor performance consistently doubled approximately every 18 months between the years 1970 and 2004.

However, as the also graph shows, processor performance began to stagnate (or at least improve much more slowly) in 2004. Among other reasons, the microprocessor industry fell off past trends due to increasingly unmanageable design complexity; increased variability in manufacturing processes; power, energy, and temperature as first class design issues; and diminishing returns from extensions to past microarchitectural optimizations.

Despite this stall in processor performance improvements, Figure 1.2 shows Moore's Law still remains in effect. Consistent with historic trends, the semiconductor industry

[1] Since scores between succeeding generations of the suite are not directly comparable, the data has been normalized using a machine configuration that had scores reported for two or more generations of the benchmark suite.

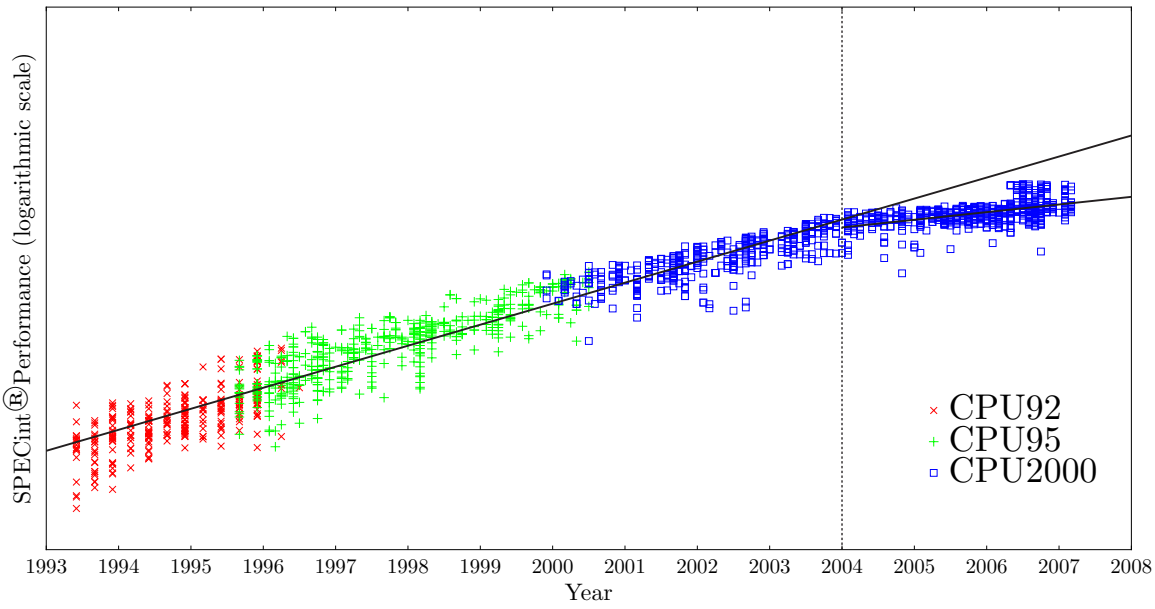


Figure 1.1: Normalized SPECint® scores for all reported configurations between 1993 and 2007.

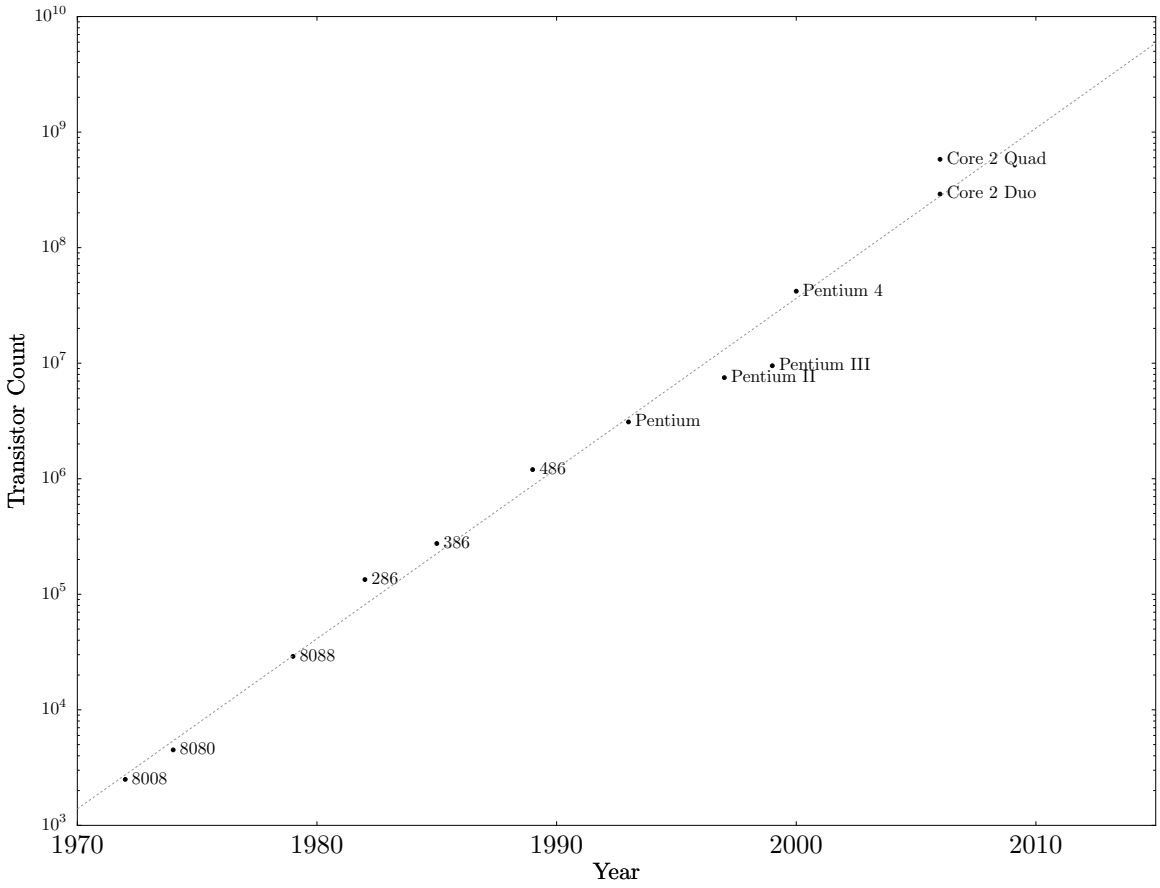


Figure 1.2: Number of transistors integrated per die for Intel x86 processors.

continues to double the number of transistors integrated onto a single die every two years. Since conventional approaches to improving performance with these transistors has faltered, microprocessor manufacturers have begun to use the extra transistors to produce *multi-core* processors. For example, in the general purpose processor space, Intel has released the Core™2 Duo and Core™2 Quad processors incorporating two and four cores per die respectively, AMD has released the Opteron™ processor in two and four core configurations, IBM has released the POWER6 processor featuring two cores per die, and Sun has released the UltraSPARC T1 and T2 processors[2] each with eight cores. These multi-core processors can improve system throughput and potentially speed up multi-threaded applications, but the latency of any single-thread of execution, at best, remains unchanged. Consequently, to exploit multi-core processors, applications must be multi-threaded, and they must be designed to efficiently use the resources (i.e., cache memory, off-chip bandwidth, etc.) provided by the processor.

1.1 Shortcomings of Existing Approaches

The task of producing efficient multi-threaded code could be left to the programmer, however, there are several disadvantages to this approach.

First, writing multi-threaded codes is inherently more difficult than writing single-threaded codes. To ensure correctness, programmers must reason about concurrent accesses to shared data and insert sufficient synchronization to ensure data accesses are ordered correctly. Simultaneously, programmers must ensure that too much synchronization is not inserted otherwise synchronization overhead and loss of parallelism due to synchronization cause the multi-threaded program to perform no better or *worse* than its single-threaded counterpart. Active research in automatic tools to identify deadlock, livelock, race conditions, and performance bottlenecks [22, 24, 25, 47, 63] in multi-threaded programs is

[2]Code named Niagara and Niagara 2, respectively.

a testament to the difficulty of achieving this balance.

Second, there are many legacy applications that are single-threaded. Even if the source code for these applications were available, it would take enormous programming effort to translate these programs into well-performing parallel versions.

Finally, even if efficient multi-threaded applications could be written for a particular multi-core system, these applications may not perform well on other multi-core systems. The performance of multi-threaded applications is very sensitive to the particular system for which it was optimized due to, among other factors, the relation between synchronization overhead and memory subsystem implementation (e.g., size of caches, number of caches, what caches are shared, coherence implementation, memory consistency model, etc.) and the relation between number of application threads and available hardware parallelism (e.g., number of cores, number of threads per core, cost of context switch, etc.). Writing an application that is portable across multiple processors (even multiple generations in the same family of processors) would prove extremely challenging.

A promising alternative approach for producing multi-threaded codes is to let the compiler automatically convert single-threaded applications into multi-threaded ones. This approach is attractive as it takes the burden of writing multi-threaded code off the programmer. Additionally, it allows the compiler to automatically adjust the amount and type of parallelism extracted based on the underlying architecture, just as instruction-level parallelism (ILP) optimizations relieved programmers of the burden of targeting their applications to complex single-threaded architectures.

In scientific and numerical computing, compilers are routinely used to extract thread-level parallelism (TLP) with good results. Techniques such as DOALL and to a lesser extent DOACROSS, are used in this domain to extract tens to hundreds of threads [8, 72]. These techniques assign loop iterations to threads to execute a loop in parallel. For DOALL, all the iterations are independent so no communication or synchronization is necessary. For DOACROSS, loop carried dependences must be communicated between the threads.

Consequently, these techniques perform well on counted loops manipulating very regular and analyzable structures consisting mostly of predictable array accesses. This reliance on regular structure, however, makes these techniques ill-suited for general purpose applications. These techniques are unable to extract significant parallelism from codes with unpredictable control flow or data access patterns that are the norm for general-purpose applications.

Unpredictable control flow and data access patterns force compilers to assume conservatively that dependences exist between instructions even when no such dependence exists or when such dependences manifest only infrequently at run time. Because excess dependences tend to be the limiting factor in extracting parallelism, speculative techniques, loosely classified as thread-level speculation (TLS), have, in recent years, dominated the literature [7,13,34,38–40,49,66,68,73,80]. Speculating dependences that prohibit DOALL parallelization or that dramatically restrict DOACROSS parallelization increases the amount of parallelism that can be extracted. Unfortunately, speculating enough dependences to create DOALL parallelism often leads to excessive misspeculation, dramatically reducing the benefits of parallelism. Consequently, TLS techniques often produce speculative DOACROSS parallelizations. Unfortunately, to overcome the serializing effects of inter-thread communication latency, even speculative DOACROSS requires significant speculation leading to high misspeculation rates and only small amounts of parallelism.

A more recent technique, Decoupled Software Pipelining (DSWP) [56,61], approaches the problem differently. Rather than partitioning a loop by placing distinct iterations in different threads, DSWP partitions a loop's body into a pipeline of threads, ensuring that critical path dependences are kept thread-local. This approach creates parallel code which is tolerant of both variable latency within each thread and long communication latencies between threads. Consequently, the approach gets better applicability than DOALL parallelization since it gracefully handles cross-iteration dependences, and can often out perform DOACROSS parallelization by removing inter-thread communication latencies from

the critical path. However, since the existing DSWP transformation is non-speculative, it must respect all dependences in the loop. Unfortunately, excess dependences due to static and conservative dependence analysis dramatically limit the applicability and scalability of the non-speculative DSWP transformation.

1.2 Contributions

This dissertation extends the pipelined multi-threading (PMT) paradigm used by DSWP with speculation. Speculation can increase the applicability and scalability of DSWP, just as it increased the applicability and scalability of DOALL and DOACROSS parallelization. More specifically, this dissertation presents *Speculative Decoupled Software Pipelining* (SpecDSWP) [74] and an initial *automatic* compiler implementation of it. SpecDSWP leverages the latency-tolerant pipeline of threads characteristic of DSWP and combines it with the power of speculation to break dependence cycles that inhibit DSWP parallelization. Like DSWP, SpecDSWP exploits the latent, fine-grained pipeline parallelism present in many applications to extract long-running, concurrently executing threads, but can extract more threads on more loops than its non-speculative counterpart.

Unlike DOALL and DOACROSS parallelization, PMT techniques, such as DSWP, do not presuppose that each thread extracted is responsible for a subset of loop iterations. Rather, PMT techniques let the code guide the parallelization. Consequently, rather than speculating many dependences to force code to fit a predetermined mold as in TLS, speculative PMT in general, and SpecDSWP in particular, can *judiciously* speculate only those dependences which improve execution efficiency and that cause minimal misspeculation at runtime. Identifying the ideal set of dependences to speculate involves solving an NP-hard problem, so this dissertation presents a novel heuristic algorithm which is demonstrated to work well in practice.

This dissertation also explores the runtime support necessary to enable SpecDSWP.

Like TLS, SpecDSWP must buffer speculatively state generated from each loop iteration until all speculated dependences have been resolved. However, the DSWP execution model creates several challenges for buffering speculative memory state since the code comprising a single loop iteration has been distributed across many threads. Despite the goal of facilitating speculative parallelization, conventional transactional memories and TLS memory subsystems are inapplicable because they limit transactions to a *single* thread. SpecDSWP requires data for a single loop iteration, which is generated across a multitude of threads, to be buffered in a single transaction.

To address this problem, this dissertation introduces *multi-threaded transactions* (MTX) and a hardware implementation based on an invalidation-based cache coherence protocol. In addition to supporting Speculative DSWP, MTXs can be used to enable *nested parallelism* in transactional programs; individual tasks in a transactional program can be further parallelized. Additionally, they can enable programmers to focus on exposing coarser-grained task-level parallelism, while letting compilers analyze and extract thread-level parallelism within individual tasks. Like its single-threaded counterpart, an MTX represents an atomic set of memory accesses, however, unlike its single-threaded counterpart, an MTX can be initiated by one thread, accessed and modified by any number of threads, and then finally committed by yet another thread. Furthermore, all threads participating in an MTX can see the results of uncommitted speculative stores executed by other threads in the MTX. As this dissertation will show, these properties are essential for enabling SpecDSWP.

Finally, this dissertation evaluates the proposed compiler transformations and runtime support. Using an initial automatic compiler implementation and a simulation infrastructure modeling an MTX-enabled memory system, this dissertation demonstrates that SpecDSWP provides significant performance gains on a multi-core processor running a variety of codes.

1.3 Dissertation Organization

Chapter 2 examines existing non-speculative and speculative parallelization techniques characterizing their applicability and scalability. This discussion will motivate SpecDSWP. Chapter 3 will describe the non-speculative DSWP transformation upon which SpecDSWP is based. Chapters 4, 5, and 6 will then describe how to extend DSWP to support speculation. These chapters will detail the algorithm to select dependences for speculation, the code generation algorithm, and the runtime-system support necessary to support SpecDSWP. Chapter 7 will discuss related work in transactional memory and describe why existing solutions are inadequate for SpecDSWP. Then, it introduces MTXs and describes a cache-coherence based implementation. A quantitative evaluation of SpecDSWP and MTXs is given in Chapter 8. Finally, Chapter 9 describes future avenues of research and summarizes the conclusions of this dissertation.

Chapter 2

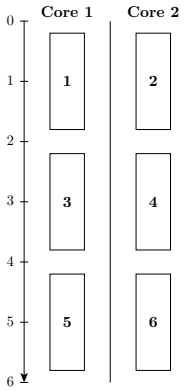
Parallelization Paradigms

The key obstacle in parallelizing applications is dealing with dependences that exist between the program's instructions. Once parallelized, the program's instructions will be distributed across various threads, and to guarantee correct execution, any dependences between instructions allocated to different threads must be synchronized or speculated. Since inter-thread synchronization latency can be large, an application's partitioning into threads must be carefully planned to ensure a high-performance multi-threaded application. Since the cost of misspeculation is also large, synchronization cannot always be eschewed in favor of speculation. Only speculation that is high reward and that will lead to infrequent misspeculation will be profitable in practice.

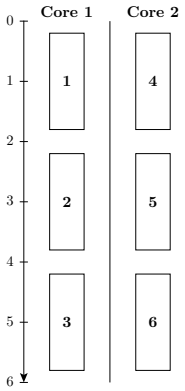
Based on the pattern of communication, multi-threaded applications can be classified into 3 categories: independent multi-threading (IMT), cyclic multi-threading (CMT), and pipelined multi-threading (PMT) [59]. Each of these three paradigms has a non-speculative and speculative variant. This chapter will describe each category using a common parallelization strategy from that paradigm: DOALL [8] for IMT, DOACROSS [23] for CMT, and DSWP [56, 61] for PMT. To understand the benefits, drawbacks, and applicability of speculation in each paradigm, this chapter will develop analytical models describing the performance of each parallelizing transformation in a variety of idealized scenarios. The

```
1  for (i = 0; i < N; i++) {
2    doit(a[i]);
3  }
```

(a) DOALL loop



(b) Naïve Schedule



(c) Optimized Schedule

Figure 2.1: The loop shown in (a) has no loop carried dependences and can be parallelized using DOALL. Two possible schedules for the parallelization are shown in (b) and (c).

virtues of pipelined multi-threading, combined with the absence of a speculative pipelined multi-threading optimization, motivate Speculative Decoupled Software Pipelining, the topic of the remainder of this dissertation.

2.1 Paradigm Descriptions

This section will introduce each of the paradigms, briefly describe the representative automatic parallelization, and derive an analytical performance model for each parallelizing transformation assuming application loop iterations exhibit no variability in latency and inter-thread communication is no more expensive than intra-thread communication. Later sections will expand these analytical models by relaxing these assumptions.

2.1.1 Independent Multi-threading and DOALL

Description In independent multi-threading (IMT), there exist no cross-thread dependences, and thus, the threads are *independent*. DOALL parallelization is the quintessential IMT loop parallelization. DOALL parallelization can be applied to a loop where each iteration is independent of all other loop iterations. Figure 2.1(a) illustrates such a DOALL

loop. DOALL loops can be parallelized by allocating sets of loop iterations to different threads. Figure 2.1(b) shows a naïve implementation where iterations are allocated round-robin to each thread. Figure 2.1(c) shows a more aggressive implementation where, to promote cache locality, larger contiguous chunks of the iteration space are allocated to threads, similar to strip mining in vectorization [46].

While many loops in scientific applications are DOALL loops, for those that are not, a significant body of research exists dedicated to transforming loop nests into a form suitable for DOALL parallelization. Essentially, these techniques apply affine transformations to loop induction variables to transform the multi-dimensional loop *iteration-space* into one where slices of the space along one dimension are independent of one another. These slices can then be run in parallel using DOALL parallelization [12].

Analytical Model Under idealized conditions, DOALL parallelization yields speedup proportional to the number of processor cores available to run the loop. This assumes that the length of each iteration is constant and is independent of where and when other loop iterations execute. Specifically, this ignores all microarchitectural affects such as branch prediction accuracy and cache locality and implies complete homogeneity among loop iterations. With these assumptions, the speedup of DOALL parallelization over sequential execution is given by:

$$\text{Speedup}_{\text{DOALL}} = \frac{NL}{\frac{N}{T}L} = T \tag{2.1}$$

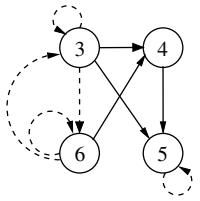
where N is the number of loop iterations, L is the length of each iteration, and T is the number of threads used in the parallelization.

2.1.2 Cyclic Multi-threading and DOACROSS

Unfortunately, despite the many transformations to coax loop nests into a form suitable for DOALL parallelization, for many general purpose loops DOALL is still inapplicable. For example, consider the loop shown in Figure 2.2(a). Figure 2.2(b) shows the program

```
1  cost=0;
2  node=list->head;
3  while(node) {
4    ncost=doit(node);
5    cost += ncost;
6    node=node->next;
7  }
```

(a) Loop with loop-carried dependences.

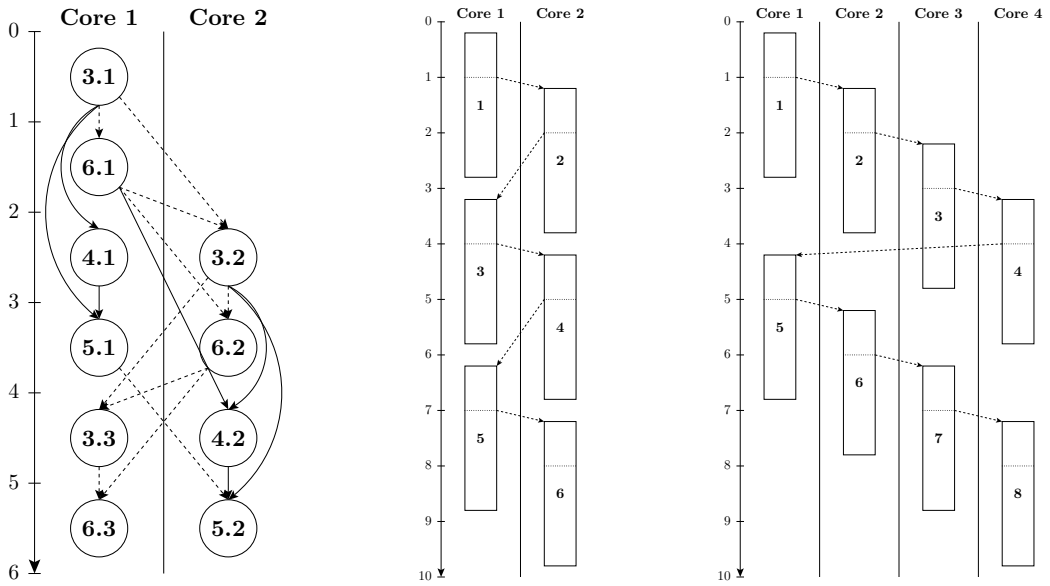


(b) PDG

Figure 2.2: The loop shown in (a) has loop carried dependences and cannot be parallelized with DOALL. It can be parallelized using DOACROSS or DSWP. The PDG for the loop is shown in 2.2(b).

dependence graph (PDG) [27] corresponding to the code. In the PDG, edges that participate in *dependence cycles* are shown as dashed lines. Since the statements on lines 3 and 6 and the statement on line 5 each form a dependence cycle, each iteration is dependent on the previous one. Since the loop is not counted, techniques to transform the iteration space into one that is amenable to DOALL are inapplicable. Even if such techniques were applicable to pointer-chasing loops such as the one shown in the figure, no transformation could unlock DOALL parallelism from this loop since all iterations are directly or transitively dependent on all earlier iterations. To circumvent this, some researchers have proposed splitting such loops into two loops, one that traverses the linked list and another which operates on each node. Unfortunately, after the split, the first loop is executed sequentially, and only the second is executed in parallel using DOALL [79].

The entire loop however can be parallelized using DOACROSS parallelization, an instance of cyclic multi-threading (CMT). In cyclic multi-threading, cross-thread dependences exist, and these dependences form a cycle (i.e., thread 1 feeds thread 2, and thread 2 also feeds thread 1). Figure 2.3(a) illustrates how DOACROSS parallelization works using the loop from Figure 2.2(a). Similar to the naïve implementation of DOALL, loop iterations are allocated round-robin to threads participating in the parallelization. However, since each iteration is dependent on the previous one, later iterations synchronize with earlier ones waiting for the cross-iteration dependences to be satisfied. The code between



(a) Loop Schedule (b) Thread Bound Schedule (c) Dependence Bound Schedule

Figure 2.3: This figure illustrates how DOACROSS parallelism works. The 2-thread DOACROSS schedule for the code in Figure 2.2(a) is shown in (a). Figure (b) illustrates a DOACROSS schedule that is performance limited by the number of available threads. Figure (c) illustrates a DOACROSS schedule that is performance limited by the dependences in the loop.

synchronizations can run in parallel with code executing in other threads. Since, for all threads $t < T$, thread t feeds thread $t+1$, and thread T feeds thread 0, the synchronization between threads forms a cycle making DOACROSS a form of CMT.

Analytical Model With a small finite number of threads the performance of DOACROSS execution closely resembles the performance of DOALL execution. Figure 2.3(b) shows the general execution schedule of DOACROSS when the number of threads (cores) executing the loop bounds the performance. Assuming the time between when subsequent iterations start is δ, the performance of DOACROSS is identical to DOALL, with the exception of the initial synchronization delay $(T-1)\delta$. With T threads and N iterations taking L cycles per iteration, the speedup is:

$$\text{Speedup}_{\text{DOACROSS,T}} = \frac{NL}{\frac{N}{T}L + (T-1)\delta} \approx T \quad \textit{(for large } N\textit{)} \tag{2.2}$$

For large N the parallelism scales linearly with the number of threads, just as in DOALL.

However, unlike DOALL parallelism, after a certain number of threads, T_0, the performance of DOACROSS is limited by the dependence patterns present in the loop. Figure 2.3(c) illustrates this scenario. Here, the delay between when each iteration starts, δ, bounds the performance. Assuming T_0 or more threads (cores) are available to run the loop, the speed up of DOACROSS parallelization over sequential execution is:

$$\text{Speedup}_{\text{DOACROSS},\infty} = \frac{NL}{(N-1)\delta + L} \approx \frac{L}{\delta} \quad \textit{(for large } N\textit{)} \tag{2.3}$$

Assuming inter-thread communication is just as fast as intra-thread communication, loop iterations exhibit no variable latency (i.e., control flow, cache behavior, branch prediction, etc. do not cause different iterations to take different amounts of time), and that sufficient instruction-level parallelism (ILP) resources exist to execute dependence cycles optimally, the constant δ can be bound by considering the dependence height of all depen-

dence cycles in the loop. An instruction i in iteration n (denoted i_n) can only execute after all of the instructions that feed it have completed. This means the earliest that i_{n+1} can execute is τ_c cycles after i_n if instruction i participates in a dependence cycle with dependence height τ_c. Since any dependence cycle for any instruction can delay the execution of an iteration, δ is greater than or equal to the maximum latency of all dependence cycles for all instructions in the loop.

$$\delta \geq \max_c \tau_c \tag{2.4}$$

If there exist dependence cycles with dependence distances greater than one, $d_c > 1$, (i.e., iteration n feeds iteration $n+d_c$ rather than iteration $n+1$), then δ can be bound as follows:

$$\delta \geq \max_c \frac{\tau_c}{d_c} \tag{2.5}$$

The tightness of this bound on δ depends on how control intensive the loop being parallelized is. The synchronization in DOACROSS follows the producer-consumer paradigm. This means that the produce and consume instructions must have identical conditions of execution. Placing a produce instruction in a deeply nested control structure in one thread requires inserting the consume instruction in a control equivalent region in the consuming thread. However, since different threads are executing different iterations, this control equivalent point does not exist. Consequently, control intensive loops parallelized with DOACROSS typically execute all consume operations at the top of each loop iteration. One or more produce instructions are inserted such that the cumulative condition of execution for the produce instructions is equivalent to the condition of execution of the loop iteration (i.e., produce instructions are inserted along all possible paths through a loop iteration). In such a configuration, δ is equal to the delay between the start of an iteration and the last produce operation. In practice this delay can be considerably greater than the delay imposed by dependence cycles [78].

Returning to the analytical model, the actual speedup obtained by DOACROSS paral-

lelization is:

$$\text{Speedup}_{\text{DOACROSS}} = \min(\text{Speedup}_{\text{DOACROSS},\infty}, \text{Speedup}_{\text{DOACROSS,T}}) \tag{2.6}$$

The number of threads, T_0 at which the dependence delay begins to dominate can be computed by equating the denominators (i.e., the time taken to execute the parallel code) in the speedup formulas (2.2) and (2.3).

$$(N-1)\delta + L = \frac{N}{T_0}L + (T_0 - 1)\delta \tag{2.7}$$

Solving equation (2.7) for T_0 yields $T_0 = \frac{L}{\delta}$.[1] Conceptually, this says the crossover occurs when the time that all cross-thread dependences for iteration $n + T$ are satisfied is precisely equal to the time when thread t finishes processing iteration n.

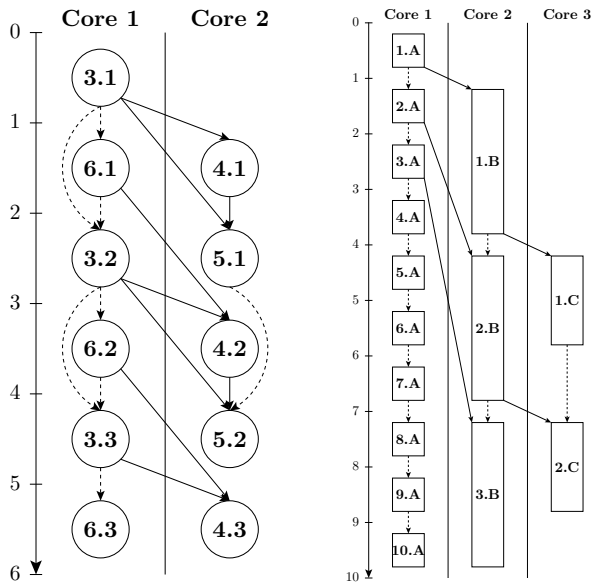
Overall, notice that while DOACROSS exploits iteration-level parallelism, its performance scales with the number of threads *only* up to a fixed number of threads, T_0, after which additional threads offer no more parallelism. Further, T_0 is at best determined by the dependence patterns present in the loop and at worst by the control patterns in the loop that prevent efficient synchronization.

2.1.3 Pipelined Multi-threading and DSWP

Description Like the CMT approach, applications parallelized in the pipelined multi-threading (PMT) paradigm possess cross-thread dependences. However, unlike CMT, the cross-thread dependences form a pipeline (or more specifically, a directed acyclic graph (DAG)) rather than a cycle. Formally, thread t can feed thread any thread $t' > t$. However, thread t cannot feed any thread $t' < t$.

Decoupled Software Pipelining (DSWP) is a loop parallelization transformation pro-

[1] There also exists the trivial solution where $T_0 = N$. This solution is uninteresting because there is one thread per loop iteration.



(a) Loop Schedule (b) General Schedule

Figure 2.4: This figure illustrates how DSWP parallelism works. The 2-thread DSWP schedule for the code in Figure 2.2(a) is shown in (a). Figure (b) illustrates the general schedule of a DSWP parallelization.

ducing pipelined multi-threaded code. Unlike DOALL and DOACROSS parallelization, DSWP does not allocate entire loop iterations to threads. Instead, each thread executes a portion of *all* loop iterations. The pieces are selected such that the threads form a pipeline. Figure 2.4(a) illustrates how DSWP would parallelize the code from Figure 2.2(a). In the figure, thread 1 is responsible for executing statements 3 and 6 for all iterations of the loop, and thread 2 is responsible for executing statements 4 and 5 for all iterations of the loop. Since neither statement 4 nor 5 feed statements 3 and 6, all cross-thread dependences flow from thread 1 to thread 2 forming a thread pipeline.

Analytical Model Since the threads produced by DSWP execute different pieces of each loop iteration, each thread may complete its portion of each loop iteration at a different rate. Figure 2.4(b) illustrates the general schedule of DSWP parallelization. Assuming the rate of completion for each thread, r_t, is constant, then the performance of the DSWP parallelized loop is limited by the slowest thread, $t_{\text{slowest}} = \arg\min_t r_t$. Denoting the length of a loop iteration in thread t as $L_t = \frac{1}{r_t}$, the speedup over sequential execution of a DSWP

parallelization is:

$$\text{Speedup}_{\text{DSWP}} = \frac{NL}{(N-1)L_{t_{\text{slowest}}} + L} \approx \frac{L}{L_{t_{\text{slowest}}}} \quad \textit{(for large } N\textit{)} \tag{2.8}$$

DSWP can partition the body of the loop being parallelized however it likes provided the resulting threads form a pipeline. Consequently, there is no single DSWP parallelization for a particular loop. Instead, depending on the number of threads being targeted and heuristic partitioning algorithms in a DSWP implementation many partitions of the code are possible. However, dependence patterns in the loop being parallelized limit the performance offered by DSWP. Since DSWP must produce a pipeline of threads, no dependence cycle can span multiple threads. Consequently, instructions comprising a strongly connected component (SCC), a collection of overlapping cycles, in the loop's dependence graph must be allocated to the same thread. Assuming sufficient ILP resources to execute SCCs optimally, the SCC with the largest dependence height bounds the rate of the slowest thread. Mathematically, if θ_s is the dependence height of SCC s, then

$$L_{t_{\text{slowest}}} \geq \theta_s \tag{2.9}$$

In contrast to DOACROSS, even for control-intensive general purpose codes, this bound is tight. If DSWP is provided sufficiently many threads, it can ensure that the SCC with the largest dependence height is placed in its own thread, and that no other thread's latency exceeds the dependence height of the slowest SCC (in the limit, a DSWP implementation could allocate each SCC to its own thread). Further, since each thread in a DSWP parallelized loop executes every original loop iteration, for any program point in one thread, a control equivalent point can inexpensively be created in all later threads [56]. Consequently, produce-consume synchronization can be inserted anywhere in the loop, including within deeply nested control structures and inner loops. Compared to the original DSWP code generation algorithm, synchronization overhead can be further reduced using com-

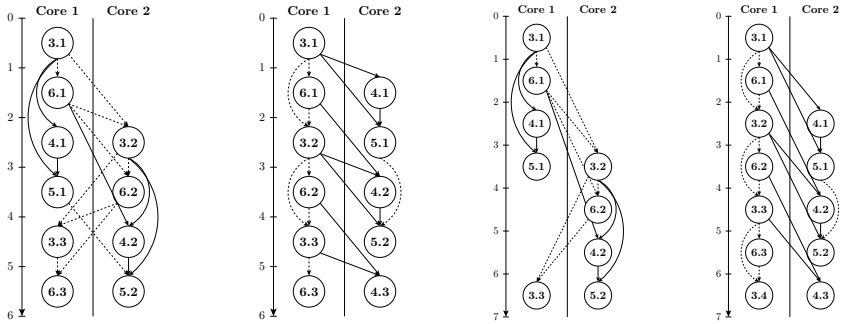
munication optimizations that minimize the number of dynamic synchronizations and dramatically reduce the amount of control structure replication [55].

DOACROSS and DSWP both seemingly offer similar parallelization potential. As the number of threads grows, DOACROSS ideally approaches a speedup of $\frac{L}{\max_c \tau_c}$, and DSWP ideally approaches a speedup of $\frac{L}{\max_s \theta_s}$. However, since each dependence cycle is contained in some strongly connected component, $\max_s \theta_s \geq \max_c \tau_c$. Consequently, under *ideal* conditions, DOACROSS should always outperform DSWP. However, for several reasons, DSWP often outperforms DOACROSS in practice. First, recall that the performance bound for DOACROSS is not tight for control-intensive general purpose applications, while the bound is tight DSWP. Consequently, DSWP parallelizations in practice more closely track the ideal speedups compared to DOACROSS parallelizations. Second, parallel-stage DSWP, an extension to DSWP, effectively reduces the latency of any SCC that does not contain a dependence carried by the loop being parallelized [58]. This often narrows the gap between the $\max_s \theta_s$ and $\max_c \tau_c$. Further, as the next few sections will demonstrate, DSWP is tolerant to communication latency, dynamic load imbalance, and can often make better use of speculation. These properties further differentiate DSWP from DOACROSS in practice.

2.2 Tolerance to Communication Latency

This section explores the sensitivity of the three parallelization paradigms to inter-thread (inter-core) communication latency. Obviously, the performance of an IMT parallelization is independent of communication latency since the threads do not communicate. However, the situation is not so obvious for CMT and PMT.

Figure 2.5 shows DOACROSS and DSWP schedules for the loop from Figure 2.2(a). Figures 2.5(a) and 2.5(b) show the schedules assuming no inter-thread communication latency (i.e., inter-thread communication is no more expensive than intra-thread communi-



(a) DOACROSS (b) DSWP (c) DOACROSS (d) DSWP

Comm. latency = 0 cycles Comm. latency = 1 cycles

Figure 2.5: This figure illustrates the sensitivity of DOACROSS and DSWP parallelizations to long inter-thread communication latency. Figures (a) and (b) show the execution schedules in the presence of no inter-thread communication latency, while Figures (c) and (d) show the execution schedules with a 1-cycle latency.

cation). Figures 2.5(c) and 2.5(d) shows the schedule assuming an inter-thread communication latency of 1 cycle. Notice that the organization used by DOACROSS forces it to communicate dependences that participate in cycles (dashed lines in the PDG from Figure 2.2(b)) from thread to thread. This puts communication latency on the critical path. DSWP's organization allows it to keep these dependences thread-local (in fact the algorithm requires it) thus avoiding communication latency on the critical path. Consequently, even after the communication latency increase, DSWP completes one iteration every two cycles. DOACROSS, however, now only completes one iteration every three cycles.

In terms of the analytical models developed in the previous section, the speedup for DOACROSS in the presence of non-zero communication latency, λ, is

$$\text{Speedup}_{\text{DOACROSS,T}} = \frac{NL}{\frac{N}{T}L + (T-1)(\delta+\lambda)} \approx T \quad \textit{(for large } N\textit{)} \tag{2.10}$$

$$\text{Speedup}_{\text{DOACROSS},\infty} = \frac{NL}{(N-1)(\delta+\lambda)+L} \approx \frac{L}{\delta+\lambda} \quad \textit{(for large } N\textit{)} \tag{2.11}$$

Dependence and communication delay begin to dominate when $T > \frac{L}{\delta+\lambda}$. Consequently, fewer threads are useful and the maximum attainable speedup is smaller. These effects are

more pronounced when δ is small[2], that is to say, when the sequential portion of the loop is small.

For DSWP, the speedup in the presence of non-zero communication latency is

$$\text{Speedup}_{\text{DSWP}} = \frac{NL}{(N-1)L_{t_{\text{slowest}}} + L + (T-1)\lambda} \approx \frac{L}{L_{t_{\text{slowest}}}} \quad \textit{(for large } N\textit{)} \qquad (2.12)$$

As the example demonstrated, the speedup for a large number of iterations is *completely* unaffected by communication latency. Only the pipeline fill time, $L+(T-1)\lambda$, is affected. This communication latency independence makes DSWP particularly promising given that the latency of inter-thread synchronization is often high compared to the latency of computation. Further, this latency can only be expected to increase as more cores are integrated onto multi-core processors.

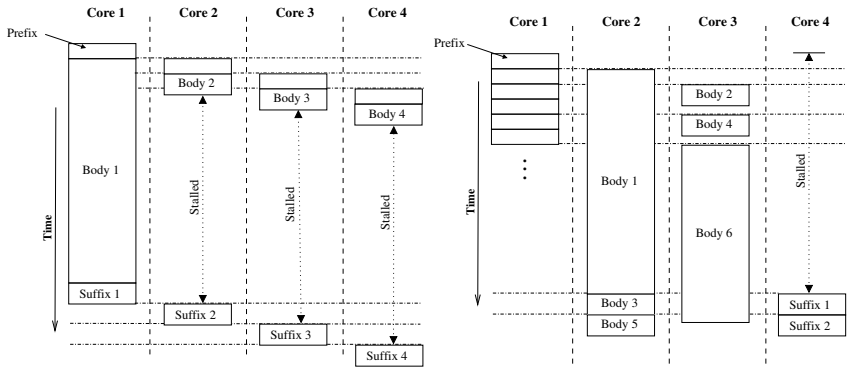
2.3 Tolerance to Load Imbalance

Thus far it has been assumed that loop iterations exhibit no variability in latency, and consequently, the threads produced by parallelization are inherently balanced. This section considers the efficiency of each paradigm in the presence of load imbalance.

Load imbalance can be divided into static and dynamic load imbalance. Static load imbalance is caused by different threads executing different code which may not be balanced. Dynamic load imbalance, on the other hand, is caused by runtime effects. Variability in the execution time of loop iterations due to input data, differing control flow paths, cache effects, branch prediction, etc. are all instances of dynamic load imbalance.

Since all threads in a DOALL or DOACROSS parallelization execute the same code, there is no possibility for static load imbalance. DSWP, conversely, partitions a loop's body allocating different pieces to different threads. Statically, the work allocated to each thread may not be balanced, and the pipeline requirement DSWP imposes may inhibit load

[2]More precisely, when λ is large compared to δ



(a) DOACROSS Schedule (b) DSWP Schedule

Figure 2.6: Comparison of DOACROSS and DSWP on dynamically imbalanced workloads.

balancing. This imbalance leads to under-utilization of processor cores executing lightly loaded threads. However, this dissertation shows, in practice, that DSWP typically only produces two lightly-loaded threads with the remaining threads being roughly balanced. Consequently, the resource under-utilization is not significant. Further still, since these under-utilized threads do not lie on the critical path, it is possible for these threads to share a single processor core via traditional operating system scheduling.

Dynamic load imbalance, on the other hand, affects all three parallelization paradigms. For DOALL, and more generally IMT, dynamic load imbalance can be overcome using work queues. The approach works by placing all tasks to be completed (sets of loop iterations in DOALL parallelization) into a queue. When a thread completes one task, it goes to the queue to acquire another task. Since in IMT, all tasks are independent, the work queue can be populated with all tasks that need completion before any thread begins processing. Using this approach, all processor cores are kept occupied until the work queue is empty, leading to high utilization.

The same approach, unfortunately, cannot be used by DOACROSS, or more generally CMT, parallelizations since each task (a loop iteration in DOACROSS) is dependent on all previous tasks being at least partially completed. Figure 2.6(a) shows how a DOACROSS parallelization responds to dynamic load imbalance. In the example, each iteration can start

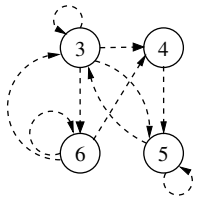
processing after a short code region at the beginning of the previous iteration completes. Before completing an iteration, however, each thread must synchronize once more with the previous iteration to guarantee that output is generated sequentially. The figure illustrates how large variability in the "meat" of the loop (the code between the two synchronizations) leads to under-utilization of many processor cores. While its tempting to use more threads than cores and rely on OS scheduling to fill the slack time, this may be counter productive since all threads lie on the critical path. This implies time spent context switching a thread that is ready to execute adds to the length of the critical path potentially slowing the parallel execution.

While not applicable to CMT, the work queue approach to smoothing out load imbalance in IMT can be simply adapted to work for PMT. With IMT, since all tasks are independent a single shared work queue can be populated with all the tasks before any thread begins processing. In PMT, not all tasks are independent. Consequently, tasks can only be enqueued when all data necessary to execute the task is available. DSWP adopts this model by implementing the cross-thread producer-consumer synchronization with queues. Essentially, this amounts to using a private per-thread work queue although, in practice, it is implemented with a collection of scalar queues. Multiple values can be buffered between the stages allowing each stage to execute unhindered by stalls or varying load in other threads provided the queues are neither full nor empty.

Figure 2.6(b) illustrates how this works for DSWP. In the example, each loop iteration is broken up into three pieces, each comprising a stage in the DSWP pipeline. These pieces correspond to the code before the first DOACROSS synchronization, the code between the first and second DOACROSS synchronization, and the code following the last DOACROSS synchronization. Since the second stage does not depend on data produced in the stage in previous iterations, the stage can be *replicated* and executed by multiple threads using an extension to DSWP known as *parallel-stage DSWP* (this will be described more thoroughly in Section 3.2) [58]. Notice that cores 1 and 4 are not fully utilized due to static load

```
1  cost=0;
2  node=list->head;
3  while(cost<T && node) {
4    ncost=doit(node);
5    cost += ncost;
6    node=node->next;
7  }
```

(a) Loop



(b) PDG

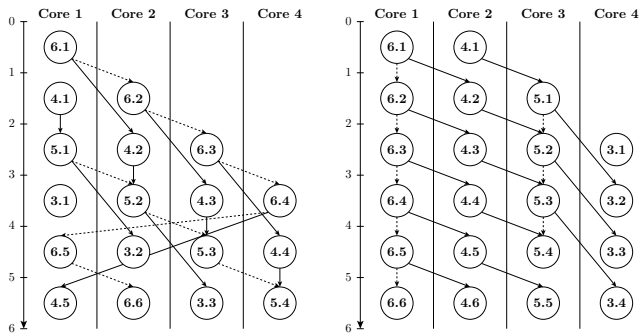
Figure 2.7: The loop shown in (a) is similar to the loop in Figure 2.2(a), but cannot be parallelized with either DOACROSS or DSWP. It can, however, be parallelized with TLS and Speculative DSWP. The PDG for the loop is shown in 2.2(b).

imbalance. However, core 1 is not stalled by the variability in execution time in the second pipeline stage; it simply enqueues more work for the later stages to complete. Cores 2 and 3 are fully utilized despite dynamic load imbalance benefitting from the work enqueued by core 1. Core 4 is delayed by the dynamic load imbalance, however, as it is not on the critical path, this delay is of little consequence. Further, as the number of cores used for the replicated stage increases, the fraction of under-utilized cores diminishes. This tolerance to *dynamic* load imbalance is yet another distinguishing factor making DSWP attractive in practice.

2.4 Applicability of Speculation

While DOALL, DOACROSS, and DSWP (or more generally, non-speculative IMT, CMT, and PMT transformations) can parallelize a certain class of loops, for many loops the dependence patterns preclude effective non-speculative parallelization. In the case of DOALL, any loop carried dependence makes the transformation inapplicable. For DOACROSS, if $\delta + \lambda$ is a significant fraction of the length of a loop iteration L, then only small speedups are attained. Similarly, for DSWP, if the dependence height of the largest SCC, $L_{t_{\text{slowest}}}$, is too large, then little to no speedup is attainable.

For example, neither DOALL, DOACROSS, nor DSWP can effectively parallelize the loop shown in Figure 2.7(a). This loop is almost identical to the loop in Figure 2.2(a)



(a) TLS (b) SpecDSWP

Figure 2.8: TLS and SpecDSWP schedules for the loop shown in Figure 2.7(a).

(which could be parallelized by DOACROSS and DSWP) except this loop can exit early if the computed cost exceeds a threshold. Since all the loop statements participate in a single dependence cycle (they form a single strongly-connected component in the dependence graph), DSWP is unable to parallelize the loop. Similarly, the dependence height of the longest cycle in the dependence graph is equal to the dependence height of the entire loop iteration (i.e., $\delta = L$) rendering DOACROSS ineffective as well.

In response to this, there have been many proposals for thread-level speculation (TLS) techniques which speculatively break various loop dependences [7, 13, 34, 38–40, 49, 66, 68, 73, 80]. Once these dependences are broken, DOACROSS and sometimes even DOALL parallelization is possible. In the example, if TLS speculatively breaks the loop exit control dependences (the dependences originating from statement 3), then the execution schedule shown in Figure 2.8(a) is possible. This parallelization offers a speedup of 4 over single threaded execution.

In this example, to achieve DOACROSS parallelization, only one dependence needed to be speculated. Assuming that the loop iterates many times, this speculation will yield little misspeculation and effectively parallelizes the loop (provided the communication latency λ is not large compared to a loop iteration's execution time). In general, to improve performance, a TLS implementation must reduce δ, meaning that it must only speculate the loop-carried dependences on the longest dependence cycles. Each speculation, eliminates the longest cycle from the application, reducing the bound on δ, and making the next

longest dependence cycle the bottleneck. In the limit, if all cycles are broken by speculation, TLS has transformed the loop into speculative DOALL, rather than speculative DOACROSS.

However, there are two key problems with this approach. First, TLS requires that the *loop-carried dependence* along a dependence cycle be speculated. In practice, this dependence may not be easily speculatable while another dependence along the cycle may be. A TLS implementation must choose between not speculating the dependence or speculating it and incurring large misspeculation penalties at runtime. In either case, the speedup obtained is greatly reduced. Second, reducing the dependence height of the longest cycle does not always improve performance. Recall that the dependence height of the longest cycle, $\max_c \tau_c$, is only a lower bound on δ (see Equation (2.4)). In practice, the set of instructions participating in the longest cycle often cannot be optimally scheduled due to control flow within each loop iteration. To circumvent restrictions due to control flow, TLS implementations are forced to speculate that certain control flow paths will not occur, allowing more flexibility in scheduling [78]. However, the branches that must be speculated may not be heavily biased making them difficult to speculate. A TLS implementation must once again choose between a sub-optimal parallelization or costly runtime misspeculation. When combined with the other shortcomings of DOACROSS (communication latency on the critical computation path and sensitivity to dynamic load imbalance), TLS typically offers only modest performance improvements on inner program loops.

This dissertation proposes Speculative DSWP. Speculative DSWP can be considered the DSWP analogue of TLS. Just as adding speculation to DOALL and DOACROSS expanded their applicability, adding speculation to DSWP allows it to parallelize more loops. Returning to the example in Figure 2.7, Figure 2.8(b) shows the execution schedule achieved by applying SpecDSWP. Just as in TLS, by speculating the loop exit control dependence, the largest SCC is broken allowing SpecDSWP to deliver a speedup of 4 over single-threaded execution.

Despite the similarity in this example, the flexibility of DSWP (and the PMT paradigm in general) allows Speculative DSWP to avoid many of the pitfalls of TLS. Recall that the performance of a loop parallelized with DSWP is determined by the performance of the slowest thread. This performance is in turn bound by the maximum dependence height of all the SCCs in the dependence graph, $\max_s \theta_s$. Therefore, theoretically, the performance of DSWP can be improved by speculatively breaking the SCCs with the largest dependence heights. However, unlike TLS, theory meets practice. Rather than limiting the compiler to speculate only loop-carried dependences, Speculative DSWP is free to speculate *any* dependence that breaks an SCC. This gives the compiler the freedom to select the most predictable dependence. Further, recall that the bound on the performance of the slowest thread (see Equation (2.9)) is tight. This means any reduction in the dependence height of the largest SCCs directly translates to performance in practice. Coupled with the other practical benefits of DSWP (insensitivity to communication latency and decoupled execution to smooth dynamic load imbalance), this freedom to speculate *intelligently* makes Speculative DSWP a promising automatic parallelization framework. The remainder of this dissertation describes Speculative DSWP in more detail, introduces the runtime system necessary to support Speculative DSWP, and evaluates its performance on range of control-intensive general purpose applications. The next chapter begins the detailed description of Speculative DSWP by reviewing the details of the non-speculative DSWP transformation.

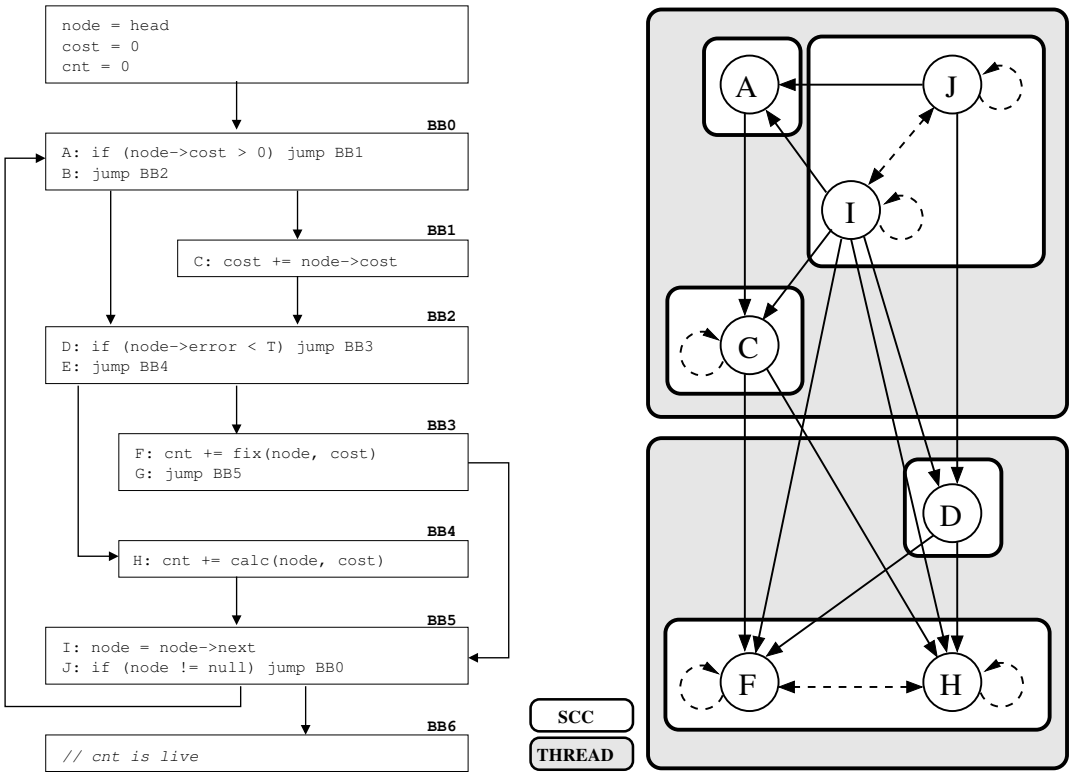
Chapter 3

Decoupled Software Pipelining

This chapter describes the DSWP transformation, from which the Speculative DSWP algorithm is built. DSWP is a *non-speculative* pipelined multi-threading transformation that parallelizes a loop by partitioning the loop body into stages of a pipeline. Like conventional software pipelining (SWP), each stage of the decoupled software pipeline operates on a different iteration of the loop, with earlier stages operating on later iterations. DSWP differs from conventional SWP in three principal ways. First, DSWP relies on thread-level parallelism (TLP), rather than instruction-level parallelism (ILP), to run the stages of the pipeline in parallel. Each pipeline stage executes within a thread and communicates to neighboring stages via communication queues. Second, since each pipeline stage is run in a separate thread and has an independent flow of control, DSWP can parallelize loops with complex control flow. Third, since inter-thread communication is buffered by a queue, the pipeline stages are *decoupled* and insulated from stalls in other stages. Variability in the execution time of one stage does not affect surrounding stages provided sufficient data has been buffered in queues for later stages and sufficient space is available in queues fed by earlier stages [61].

DSWP has four main steps (see [56] for a detailed discussion):

1. The program dependence graph (PDG) is constructed for the loop being parallelized.



(a) CFG (b) PDG

Figure 3.1: Loop illustrating how the DSWP transformation operations. Figure (a) shows the CFG for the loop. Figure 3.1(b) shows the PDG for the loop, highlights the SCCs in the dependence graph, and shows one possible partition of the operations.

The PDG contains all register, memory, and control dependences present in the loop.[1]

2. All dependence cycles are found in the PDG by identifying its strongly-connected components (SCCs). To ensure that there are no cyclic cross-thread dependences after partitioning, the SCCs will be the minimum scheduling units.

3. Each SCC is allocated to a thread ensuring that no cyclic dependences are formed between the threads. SCCs are partitioned among the desired number of threads according to a heuristic that tries to minimize the static imbalance between the threads [56].

4. Based on the SCC allocation, DSWP generates the parallel code.

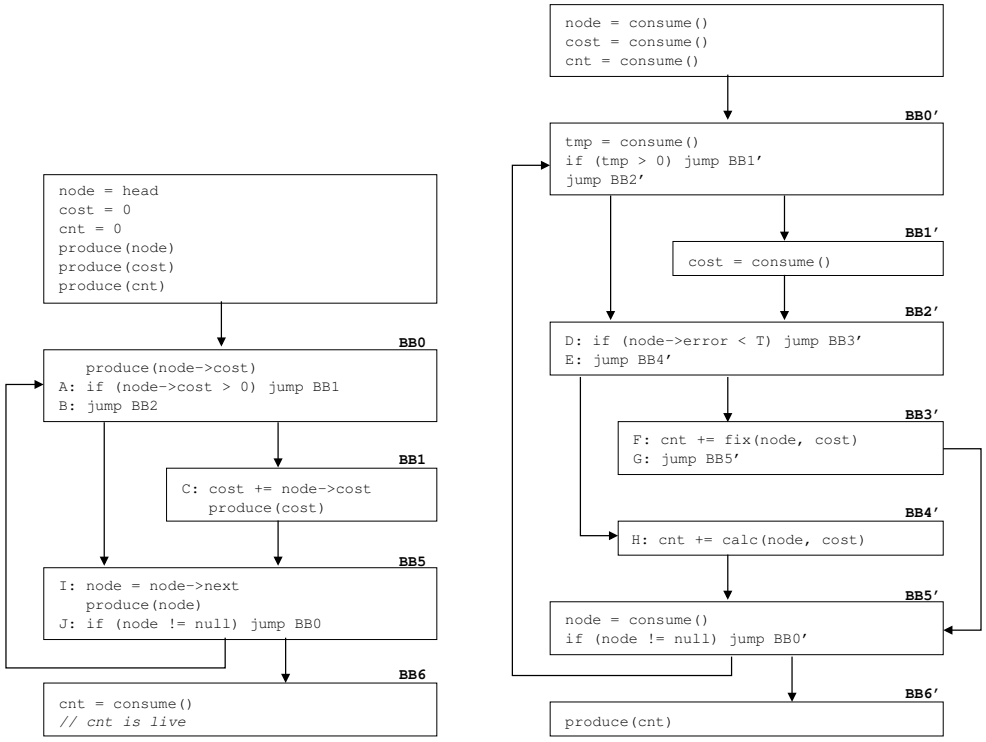
Figure 3.1(b) illustrates steps 1–3 of this process on the loop in Figure 3.1(a).

3.1 Code Generation

Figure 3.2 illustrates the code would be generated for the partition shown in Figure 3.1(b). The following paragraphs explain how this code is generated. To start, DSWP encapsulates each thread it produces in a function allowing the thread to be spawned using traditional operating system or library calls such as `clone` or `pthread_create`.

Handling Live-Ins After creating a function for each thread, the DSWP transformation inserts code to initialize each thread by synchronizing any loop live-in values needed by a particular thread. The set of registers that need synchronization is easily computed using the upwards-exposed uses data flow analysis and comparing the uses exposed at the loop header with the operations allocated to each thread. In the example, the variables `node`, `cost`, and `cnt` are live into the loop and are used by thread 2. Consequently, in the loop preheader, these variables are synchronized between the threads.

[1] Register anti- and output-dependences are ignored since each thread will have an independent set of registers.



(a) Thread 1 (b) Thread 2

Figure 3.2: Parallel code produced by DSWP for the loop in Figure 3.1(a).

Block Creation After inserting code to synchronize live-ins, basic blocks are created to hold the operations for each thread. A basic block from the original single-threaded loop is needed in a thread t if some instruction from that block has been allocated to thread t, or if thread t will have a consume operation in that block. In the example, thread 1 needs basic blocks 0, 1, and 5 because operations A, C, I, and J were allocated to it (note, unconditional jumps do not get allocated to any thread). Thread 2 needs basic blocks 2, 3, and 4 because operations D, F, and H were allocated to it. Additionally, it needs basic blocks 0, 1, and 5 to hold synchronization operations. The set of blocks where thread t will have consumes is computed iteratively. The set is initialized with all the basic blocks containing instructions that are the source of cross-thread dependences feeding thread t. For thread 2 in the example, this seed set is blocks 1 and 5. Then, any block containing a branch that controls a block in the set is added to the set. In the example, since operation A controls basic block 1, block 0 is added to thread 2. This process iterates until convergence.

Control Flow After the blocks are created, the basic control flow in each thread is created by linking basic blocks with unconditional jumps. For each original block that ended with an unconditional jump, an unconditional jump is placed at the end of the corresponding blocks in each thread. In the example, unconditional jumps are placed at the end of basic blocks 0, 2, and 3. Since the target of an unconditional jump at the end of a basic block in the single-threaded code may not exist in all threads, the target of the inserted unconditional jump is calculated by traversing the post-dominator tree. For example, the unconditional jump at the end of basic block 0 originally pointed to basic block 2. However, in thread 1, since basic block 2 does not exist, the inserted unconditional jump targets block 5, the first block that exists in thread 1 and that post dominates basic block 2 (in the original code).

Instruction Allocation After the initial control flow has been established, each thread is populated with the instructions allocated to it. Each instruction is placed in the basic block corresponding to the basic block in the single-threaded code from which the instruction came. When a branch instruction is placed, its targets must be updated to point to blocks in its thread. Once again, since not all basic blocks exist in all threads, branch targets are updated using the post-dominator tree.

Synchronization Next, synchronization is inserted to preserve the original loop's semantics. For each dependence in the PDG that crosses between two threads, a produce-consume pair must be inserted. Each produce explicitly identifies the recipient thread[2], and the compiler must ensure that each dynamic produce is matched with a corresponding dynamic consume in the target thread. For register and memory dependences, this is guaranteed by inserting a produce immediately after a defining instruction, and a consume in the corresponding location in the consuming thread. For a register dependence, the value contained in the register is passed between the threads. For example, since the variable `cost` is defined in thread 1 and used in thread 2, a cross-thread dependence exists. Assuming that

[2]In the example, the target thread numbers are omitted since there are only two threads.

`cost` is stored in a register, a produce operation is inserted immediately after the variable is defined in block 1, and in thread 2, a consume operation is inserted in the corresponding location in its copy of the block.

For memory dependences, a produce consume pair is used only to synchronize the two threads, not to communicate a value. To ensure loads performed by the consuming thread observe stores performed by the producing thread, the produce and consume operations used for memory synchronization must have the appropriate *barrier* semantics for the architecture's memory consistency model.

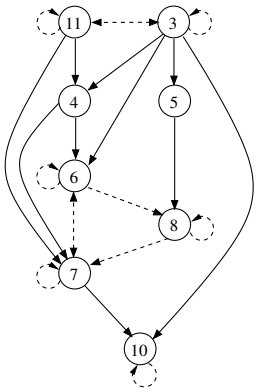
For control dependences, produce-consume pairs are inserted *before* the controlling branch. One pair is used to communicate each value used by the branch instruction. Just as for register dependences, the consumes are placed in the corresponding location in the consumer thread. In the example, thread 1 communicates the value of `node->cost` to thread 2 in basic block 0 to synchronize the control dependence between operation A and basic block 1. Additionally, however, the consuming thread also inserts a copy of the branch instruction updating the branch targets appropriately. This effectively communicates control flow from one thread to another. In the example, the branch instruction immediately following the consume operation in BB0' in thread 2 is a copy of operation A.

Note, this algorithm for inserting synchronization does not optimize the location of the synchronization. For example, if `cost` were synchronized in BB2 rather than in BB1, it would have been unnecessary to communicate the value of `node->cost` from thread 1 to thread 2. Ottoni et al. have proposed a more sophisticated algorithm that would perform this and other optimizations [55].

Handling Live-Outs Finally, code generation concludes by inserting synchronization of loop live out values. These values may be defined in any thread and must be communicated back to the primary thread. In the example, the variable `cnt` is live out of the loop, but its value is stored locally in thread 2. To return this value to thread 1, a produce-consume pair

```
1  cost=0;
2  node=list->head;
3  while(node) {
4    state = node->start_state;
5    i = 0;
6    while (!state->end) {
7      state = xfer(state, node->in[i]);
8      i++;
9    }
10   cost += state->cost;
11   node=node->next;
12 }
```

(a) Loop



(b) PDG

Figure 3.3: A loop amenable to PS-DSWP. In Figure (b) dashed edges participate in a SCC.

is inserted in basic block 6.

3.2 Parallel-Stage DSWP

As Chapter 2 mentioned, the performance of DSWP is limited by the SCC with the largest dependence height. Speculation can be used to reduce this dependence height, but in certain cases, the non-speculative technique, parallel-stage DSWP (PS-DSWP), can effectively reduce this dependence height without incurring the overhead of misspeculation. This section will describe how the technique works conceptually through an example. The details of the code generation algorithm can be found in the literature [58].

Consider the loop and dependence graph shown in Figure 3.3. The loop contains three non-trivial SCCs: the statements in the inner loop, the cost update, and the node update. Assuming the iteration count for the inner loop is large, then the dynamic dependence height of that SCC will be the large and will limit the speedup offered by DSWP.

However, while the statements in the inner loop do form an SCC in the dependence graph, none of the dependences in the SCC are carried by the outer loop. Consequently, the dependence height of this SCC can be effectively divided by an arbitrary factor by unrolling the outer loop. Figure 3.4 illustrates the dependence graph for the outer loop unrolled once.

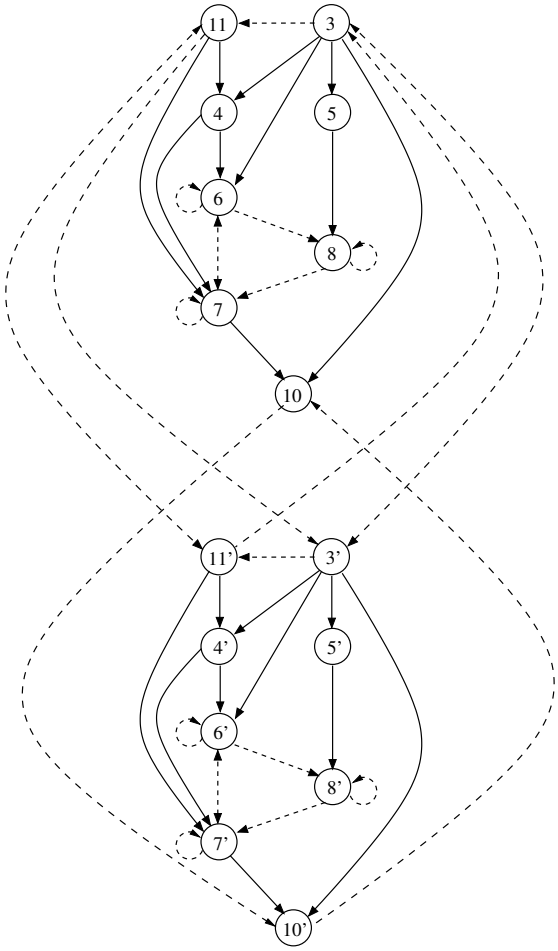


Figure 3.4: The PDG from figure 3.3(b) unrolled once. In the figure dashed edges participate in a SCC.

For the SCC that computed cost and the SCC that updated node, the original instructions and their duplicates form a single SCC in the unrolled code. However, each copy of the inner loop forms its own *independent* SCC. Disregarding the effects of single-threaded scheduling, if the dependence height of a single iteration of the original outer loop is L, then the dependence height of the unrolled loop is $2L$. If the dependence height of the inner loop SCC is L_s, then the bound on speedup obtainable by DSWP doubles from $\frac{L}{L_s}$ to $\frac{2L}{L_s}$. Unrolling can continue until the available cores are exhausted or this SCC no longer has the largest dependence height.

The code growth due to excessive unrolling may be undesirable, however, Raman et al. propose an extension to the code generation algorithm described in the previous section that does not require unrolling [58].

Chapter 4

Speculative DSWP

Despite the success of DSWP and PS-DSWP, these non-speculative transformations do not leverage the fact that many dependences are easily predictable or manifest themselves infrequently. If these dependences were *speculatively* ignored, large dependence recurrences (SCCs) may be split into smaller ones with more balanced performance characteristics. These smaller recurrences provide DSWP with more scheduling freedom, leading to greater applicability, scalability, and performance.

4.1 Execution Model

Before describing the compiler transformation used by speculative DSWP, this section outlines the basic execution paradigm and hardware support necessary. SpecDSWP transforms a loop into a pipeline of threads with each thread's loop body consisting of a portion of the original loop body. SpecDSWP speculates certain dependences to ensure no dependence flows between a later thread and an earlier thread. In the absence of misspeculation, SpecDSWP achieves decoupled, pipelined multi-threaded execution like DSWP.

To manage misspeculation recovery, threads created by SpecDSWP conceptually checkpoint architectural state each time they initiate a loop iteration. When misspeculation is detected, each thread is resteered to its recovery code, and jointly the threads are respon-

sible for restoring state to the values checkpointed at the beginning of the misspeculated iteration and re-executing the iteration non-speculatively. The SpecDSWP implementation described in this dissertation uses software to detect misspeculation and recover register state. It, however, relies on *multi-threaded transactions* implemented in hardware (see Chapter 7) to rollback the effects of speculative stores.

4.2 Compiler Transformation Overview

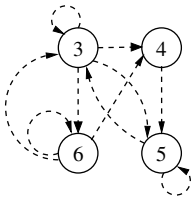
To generate parallel code, a Speculative DSWP compiler executes the steps below.

1. Build the PDG for the loop to be parallelized.
2. Select a candidate set of dependences to speculate. (See Section 5.1).
3. Build the speculative PDG – a PDG with the candidate set of speculated dependences removed.
4. Find the dependence cycles in the speculative PDG by identifying its SCCs.
5. Allocate each SCC to a thread ensuring that no cyclic dependences are formed between the threads (ignoring the candidate set of speculated dependences).
6. Using the partition, compute the final set of speculation necessary to guarantee pipelined execution. (See Section 5.2.)
7. Transform the single-threaded code to realize the speculation, and insert code to detect misspeculation. Update the partition to reflect the code changes. (See Sections 6.1 and 6.2.)
8. Apply the non-speculative DSWP code generation algorithm. (See Chapter 3.)
9. Insert code to recover from misspeculation (See Section 6.3).

Before describing each step in more detail, this chapter will illustrate these steps using the loop from Figure 4.1(a). First, a SpecDSWP compiler builds the program dependence graph for the loop being parallelized (Step 1). Figure 4.1(b) shows the PDG for the loop in Figure 4.1(a). Using this PDG, the compiler then analyzes each dependence deciding

```
1  cost=0;
2  node=list->head;
3  while(cost<T && node) {
4    ncost=doit(node);
5    cost += ncost;
6    node=node->next;
7  }
```

(a) Loop



(b) PDG

Figure 4.1: A loop demonstrating Speculative DSWP. This example reproduces the loop from Figure 2.7. The loop is not amenable to DSWP or DOACROSS since all the instructions form a single dependence cycle.

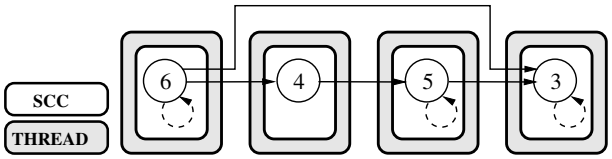


Figure 4.2: The speculative PDG for the loop in Figure 4.1(a) formed by speculating the loop exit control dependences originating from statement 3.

whether it is sufficiently predictable to be sepculated (Step 2). In the example, the data dependences always manifest, so assuming they cannot be value predicted, they are *not* candidates for speculation. Assuming the average number of loop iterations per invocation is large, the loop exit control dependences (i.e., the dependences originating from statement 3) are easily predicted, so the compiler selects these as speculation candidates. The compiler then removes these dependences from the PDG forming the speculative PDG shown in Figure 4.2 (Step 3). The compiler next identifies the SCCs in the speculative PDG (Step 4) so that it can schedule the SCCs to threads (Step 5). Figure 4.2 shows the SCCs in the speculative PDG and shows one possible allocation of SCCs to threads. In the last step before code generation, the compiler identifies which of the speculation candidates are backwards dependences in the chosen partition. These speculations are retained while all others are discarded. Since the speculated control dependences are all backwards in the selected partition, the compiler would retain all the speculations (Step 6).

To realize the speculation, the compiler inserts the necessary value speculation and misspeculation detection code in to the single-threaded loop. Figure 4.3 shows the results of

```
1  cost=0;
2  node=list->head;
3  loop:
4    if(cost<T && node) misspec;
5    ncost=doit(node);
6    cost += ncost;
7    node=node->next;
8    goto loop;
```

Figure 4.3: The code from Figure 4.1(a) after Step 7 is applied.

```
1  node=list->head;
2  produce(T2, node);
3  produce(T4, node);
4  loop:
5    node=node->next;
6    produce(T2, node);
7    produce(T4, node);
8    goto loop;
```

(a) Thread 1

```
1  node=consume(T1);
2  loop:
3     ncost=doit(node);
4     produce(T3, ncost);
5     node=consume(T1);
6     goto loop;
```

(b) Thread 2

```
1  cost=0;
2  produce(T4, cost);
3  loop:
4     ncost=consume(T2);
5     cost += ncost;
6     produce(T4, cost);
7     goto loop;
```

(c) Thread 3

```
1  node=consume(T1);
2  cost=consume(T3);
3  loop:
4    if(cost<T && node) misspec;
5    cost=consume(T3);
6    node=consume(T1);
7    goto loop;
```

(d) Thread 4

Figure 4.4: The code from Figure 4.3 after Step 8 is applied.

applying Step 7 to the example loop. Notice, the loop is now infinite reflecting the speculation that the loop will not exit. However, the loop exit condition is still computed, and if it is true, misspeculation is flagged. After realizing speculation, the compiler applies the non-speculative DSWP code generation algorithm (Step 8) generating the code shown in Figure 4.4. Just as in the single-threaded code, the loop contains no exits in all threads. However, the misspeculation detection code was allocated to thread 4, and it will flag misspeculation whenever the original loop would have exited.

The next two chapters will provide more details on the SpecDSWP transformation. Details on selecting what to speculate (Steps 2 and 6) are described in Chapter 5. Details regarding code generation (Steps 7 and 9) are described in Chapter 6.

Chapter 5

Selecting Speculation

Since the scheduling freedom enjoyed by SpecDSWP, and consequently its potential for speedup, is determined by the number and performance (dependence height) of the SCCs in a loop's dependence graph, SpecDSWP ideally would speculate *only* to remove dependences which break the "slowest" SCCs. Unfortunately, finding this set of speculations is difficult. To break an SCC into multiple SCCs with lower dependence heights, SpecDSWP must break the longest cycle in any given SCC. Unfortunately, finding the longest cycle is NP-complete [29] and large SCCs make exhaustive search intractable in practice. Consequently, SpecDSWP uses a heuristic solution. First, SpecDSWP provisionally speculates all dependences which are highly predictable. Next, the partitioning heuristic allocates instructions to threads using a PDG with the provisionally speculated edges removed. Once the partitioning is complete, SpecDSWP identifies the set of speculations necessary to permit the partition. Speculations not in this set (i.e., speculations that eliminate dependences between a thread and one of its successors) are *unspeculated*. Since these speculations are *not* passed to the code generator, they cannot cause misspeculation at runtime.

This chapter describes this process in detail. Section 5.1 describes the profile-guided algorithms to produce the set of provisionally speculated dependences. The section concludes by describing how the set of speculations are used to form the PDG used by the

partitioning heuristic. Section 5.2 then describes how the set of provisionally speculated dependences are pruned to the final set of speculations using the partition generated by the heuristic. It describes the challenges created by inter-dependent speculations (sets of speculations that jointly eliminate a dependence) and introduces new data flow analyses to overcome these challenges to efficiently identify the final set of speculations.

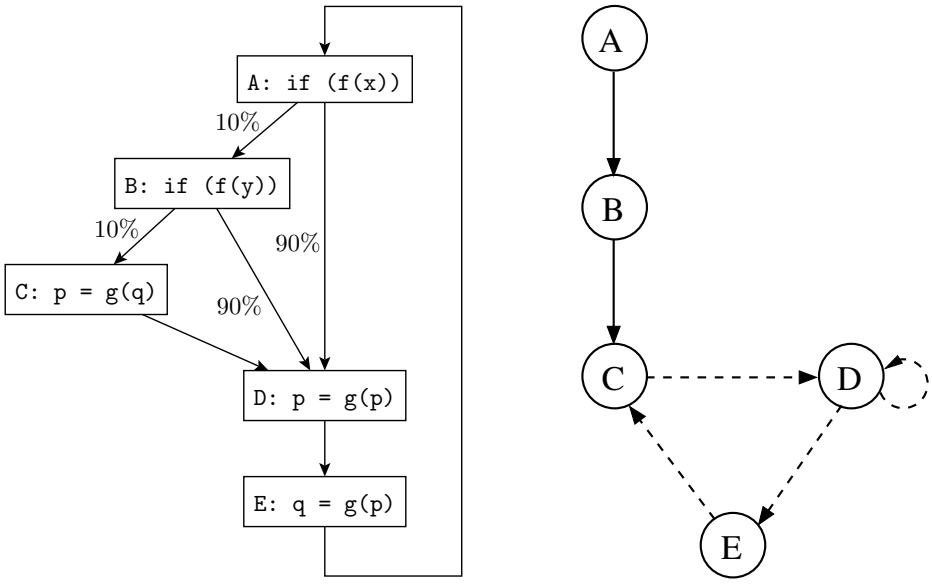
5.1 Selecting Edges to Speculate

A SpecDSWP implementation can speculate any dependence that has an appropriate misspeculation detection mechanism and, for value speculation, that also has an appropriate value predictor. Each misspeculation detection mechanism and value predictor can be implemented either in software, hardware, or a hybrid of the two. The implementation used for this dissertation, relies solely on software for value prediction and misspeculation detection. The remainder of this section will detail the speculation carried out by the SpecDSWP compiler used for this dissertation.

5.1.1 Biased Branch Speculation

As the example presented earlier (Figures 2.7, 4.2, 4.3, and 4.4) showed, speculating biased branches can break dependence recurrences. Recall from the example that the loop terminating branch (statement 3) was biased provided that the loop ran for many iterations. Consequently, the compiler can speculatively break the control dependences between the branch and other instructions. Figures 2.7(b) and 4.2 show the dependence graph before and after speculation, respectively.

Biased branch speculation can additionally break data dependences by altering the control flow of the program. For example, consider the code shown in Figure 5.1. If the control flow edge between B and C (or equivalently, between A and B) is speculated not to occur, then the control dependence between B and C (A and B) is broken. Additionally,



(a) CFG (b) PDG

Figure 5.1: This figure illustrates how control speculation (both biased branch and infrequent block speculation) can eliminate data dependences. When the edge from B to C is speculated, the dependence cycle between statements C, D, and E is broken.

any data dependences originating from or terminating at statement C can be ignored because, if at runtime execution would have reached operation C, then misspeculation will be flagged. Consequently, the control speculation breaks the data dependence cycle between statements C, D, and E by eliminating the dependence between instruction C and D, and the dependence between E and C.

Our SpecDSWP compiler assumes that all branches that exit the loop being parallelized are biased and speculates the loop will not terminate. For other branches, profile information is used to compute an edge probability for each control edge originating at the branch. The edge probability is equal to the edge's weight divided by the profile weight of the branch instruction. Given a speculation threshold T and the probability for each edge p_i, n edges will be speculated by sorting the edges in ascending order by probability, and speculating the first n edges such that $\sum_{i=0}^{n} p_i < T$. For a two way branch, this will speculate the low probability branch direction provided the branch is sufficiently biased.

Additional care must be taken when speculating branches within inner loops. While

a branch itself may be biased, the low probability path may occur frequently with respect to the loop being parallelized. The quintessential example of this phenomenon is an inner loop exit branch. If an inner loop has a high trip count, then the inner loop exit will be highly biased. However, for each invocation of the inner loop, the loop exit will be taken exactly once. Consequently, if the inner loop is invoked once per outer loop iteration, then with respect to the outer loop, the exit branch has probability 1. Since misspeculation rolls back execution to the beginning of the outer loop iteration, speculating the inner loop exit would yield a 100% misspeculation rate.

To avoid this, the compiler places additional restrictions on branch bias speculation. The compiler represents biased branch speculations using a set of control flow edges that are expected not to execute frequently. First, this set is filtered to remove any edge that has a high probability of execution with respect to the loop being parallelized. This probability is computed by dividing the number of outer loop iterations where the edge was traversed by the total number of outer loop iterations. Second, since an inner loop can have multiple exits, it is possible for each exit itself to have low probability even with respect to the outer loop. If all such exits were speculated, it would once again be tantamount to speculating that the inner loop is not invoked. To avoid this scenario, the compiler also ensures that some set of exits from inner loops are preserved.

5.1.2 Infrequent Basic Block Speculation

While speculating biased branches is successful in many cases, it can miss important speculation opportunities. Reconsider the example shown in Figure 5.1. If the branch bias threshold is set to 5%, then neither the control edge from A to B nor the control edge from B to C will be speculated. Further, notice that, despite not being the target of a sufficiently biased branch, operation C only has a 1% chance of execution. Since its execution is very unlikely, SpecDSWP will assert that the basic block containing operation C is infrequent and consequently speculatively remove all the control flow edges entering the block. Just

as was described earlier, this speculation breaks the dependence cycle between statements C, D, and E.

In general, SpecDSWP will break all incoming control flow edges in blocks whose execution probability fall below a certain threshold. Just as with biased branch speculation, it is important that block execution probabilities be computed with respect to the loop being parallelized. Additionally, if infrequent block speculation results in all of the control flow edges leaving an instruction to be speculatively removed, then the result is tantamount to speculating that all the control flow edges entering that instruction also do not execute. If all of those edges have not been selected for speculation, then the SpecDSWP compiler will conservatively discard the infrequent block speculations that lead to this condition.

5.1.3 Silent Store Speculation

In addition to control speculation, our SpecDSWP compiler speculates memory flow dependences originating at frequently silent stores [41]. To enable this speculation, the compiler first profiles the application to identify silent stores. Then, the compiler transforms a silent store into a hammock that first loads the value at the address given by the store, compares this value to the value about to be stored, and only if the two differ perform the store. After this transformation, the biased branch speculation mechanism described above is applied. The compiler will predict that the store will never occur, and if it does occur, misspeculation will be flagged.

5.1.4 Alias Speculation

Silent store speculation removes memory dependences when stores frequently write to a location, but do not change the location's value. However, a compiler's static, conservative approximation of memory aliasing contains many dependences that do not manifest in practice. Static analyses necessarily over estimate an application's memory dependences for two reasons. First, an application at runtime may use an unbounded amount of memory,

however, the compiler's analysis must bound its memory usage and complete in finite time. This restriction introduces conservativeness in memory dependence analysis. Second, the results of the compiler's analysis must indicate what memory dependences *can* occur, even if such dependences manifest rarely in practice. While some analyses have been developed to estimate the frequency of an alias [18, 65], understanding the dynamic memory dependence pattern is difficult without representative input.

Consequently, our SpecDSWP compiler speculates memory flow dependences based on the results of memory alias profiling. In its simplest form, a memory alias profile reports the number of times a load aliases with a store. This information is valuable, however it does not capture important *path* information. In the context of SpecDSWP, knowing what loop back edge carries the dependence greatly affects the scheduling freedom. For example, consider a loop nest with an outer and inner loop. If a load and store in the inner loop alias with one another, and the compiler is attempting to parallelize the inner loop, then the memory dependence is only relevant if the dependence is either frequently intra-iteration or frequently carried around the inner loop's back edge. If the dependence is primarily carried around the outer loop back edge, then for one invocation of the inner loop, the dependence will rarely manifest and it can be safely ignored speculatively. Conversely, if the outer loop is being parallelized and the dependence is primarily intra-iteration (for the inner loop) or carried around the inner-loop back edge, then the outer loop is a prime candidate for speculative *parallel-stage* DSWP; if the memory dependence creates an SCC, the SCC will not have any outer loop-carried dependences allowing the compiler to replicate the pipeline stage containing the SCC.

To capture this information, our SpecDSWP compiler relies on a *loop-aware memory profiler* (LAMP). LAMP profiles an application in much the same way as a traditional memory profiler, however each time a loop back edge or exit is traversed, alias information is summarized onto various loops in the loop hierarchy. The details of the implementation of LAMP are beyond the scope of this thesis.

```
1  int stack[STACK_SIZE];
2  int stack_index = 0;
3
4  ...
5  while(...) {
6    for (i = 0; i < 10; i++)
7      stack[stack_index++] = ...;
8    ...
9    for (i = 0; i < 10; i++)
10     ... = stack[stack_index--];
11 }
```

Figure 5.2: Loop illustrating a committed value speculation opportunity. While analysis reveals a dependence from the last store to `stack_index` in one iteration of the outer loop to the first load of `stack_index` in the subsequent iteration of the outer loop, `stack_index` is easily predictable since its value is always the same at the beginning of each iteration.

5.1.5 Committed Value Speculation

While alias speculation eliminates memory flow dependences that do not manifest frequently in practice, there are many memory flow dependences that manifest frequently, but where the loaded value is predictable. The literature is abound with value speculation mechanisms [21, 43–45, 50, 54, 70]. Our SpecDSWP implementation relies on *committed value speculation* to allow more pipeline stages to be replicated with parallel-stage DSWP.

Bridges et al. observed that in many applications, certain loads are frequently fed by stores within the same loop iteration, or they load a constant value [14]. While the locations loaded from may change value frequently, the application tends to reset the value to constant before the iteration completes. For example, consider the code shown in Figure 5.2. In each iteration of the outer loop, `stack_index` is modified many times. Consequently, data flow analysis would reveal a dependence between the last store to `stack_index` in one iteration of the outer loop and the first load of `stack_index` in the subsequent iteration of the outer loop. However, since the increments to `stack_index` are perfectly balanced with the decrements, the variable's value is always the same at the beginning of each outer loop iteration, thus making the dependence easily predictable. This constant value may not be known at compile time (e.g., the value may be non-zero depending on

what code executes before the outer loop), however, the value is a runtime constant for each invocation of the outer loop.

In such instances, the memory subsystem itself proves to be an effective value predictor to break the loop-carried dependence. Since SpecDSWP relies on runtime support to rollback speculative execution, the runtime allows loads to specify a version (in this case a loop iteration) from which to retrieve a value. For loads which follow the observed pattern, if a value is found in the version corresponding to the current iteration (i.e., if the load is being fed by stores within the same iteration), it is the correct value for the load. However, if a value is unavailable in the current version, then the value stored in the non-speculative, committed state is likely to be the correct value. Using this value breaks the loop-carried dependence and consequently enables speculative parallel-stage DSWP.

To identify loads which follow this pattern, SpecDSWP once again relies on profiling. Here, the profiler simply snapshots the state of memory at the header of the loop being targeted for parallelization. The profiler retains the last several snapshots, and for each load records two facts. First, it records whether the load was fed by a store within the current loop iteration or a previous iteration. Second, if the load was fed by a prior iteration, it records whether the loaded value is equal to the value from the retained snapshots. During compilation, depending on the number of threads being targeted, the compiler can estimate how many loop iterations will be uncommitted when a load executes. Given this distance, it consults the profile to see if snapshots a greater distance in the past accurately predict the value that will be loaded. If so, all loop-carried memory dependences feeding the load can be speculatively removed.

5.1.6 False Memory Dependence Removal

Finally, our compiler disregards all memory anti- and output-dependences. Since our recovery mechanism relies on a versioned memory (see Chapter 7) to recover from misspeculation, loads and stores can explicitly target a particular memory version. The compiler

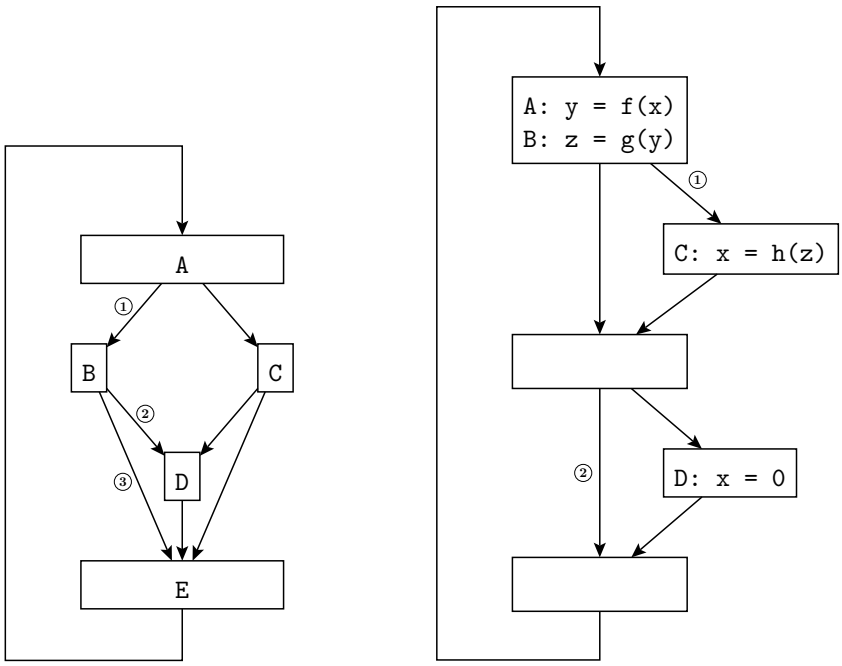
can ensure that two memory operations with a false dependence between them execute in different memory versions. Consequently, the false memory dependences no longer need to be respected. As will be seen in Chapter 6, the ability to safely ignore false memory dependences is essential for software-only alias misspeculation detection. Since the compiler is removing edges from the PDG, removing false memory dependences is similar to speculation, but since these dependences truly do not need to be respected, there is no need to detect misspeculation or initiate recovery.

5.1.7 Building the Speculative PDG

Once the candidate set of speculations is computed, the compiler must create a PDG with the speculated dependences removed. The speculations described above fall into two categories. The first type defines a *control edge* that should be speculatively removed from the CFG. The second defines a set of memory dependences that should be ignored. Updating the PDG to reflect the effects of speculation from the second category is trivial; the respective dependence edges simply need be removed from the PDG. However, updating the PDG to reflect the speculations in the first category is not as straightforward.

First, removing a control edge from the program's CFG directly eliminates control dependences. These control dependences must be removed from the PDG. However, as Figure 5.3(a) illustrates, by eliminating paths in the program, removing a control flow edge can also transform a *transitive* control dependence (which are not represented in the PDG) into a *direct* control dependence. In the example, initially control flow edge 2 controls block D, and therefore block B controls block D. Since block A controls block B, transitively block A controls block D although either path from block A can reach block D. However, if edge 3 is speculatively removed from the CFG, then edge 1 (and therefore block A) directly controls block D. This new direct control dependence must be *added* to the PDG.

Second, removing a control flow edge also eliminates data dependences. An obvious example is a data dependence originating or terminating in code predicted not to execute.



(a) Example 1 (b) Example 2

Figure 5.3: Examples illustrating how speculation affects program dependences. Figure (a) illustrates a transitive control dependence turning into a direct control dependence. Figure (b) illustrates how removing a control flow edge *non-locally* affects data flow dependences.

However, other data dependences can also be affected. Figure 5.3(b) illustrates how a control flow speculation *non-locally* affects a register-flow dependence. Prior to speculation, statement C and D both potentially feed statement A. However, if control flow edge 2 is speculated away, then statement D *always* overwrites the assignment of `x` in statement C. Consequently, after speculation, no dependence exists between statements C and A even though neither statement was speculated not to execute. These data flow effects also need to be reflected in the PDG.

Rather than try to compute directly which dependences are affected by control speculation, SpecDSWP simply reruns the analyses to build the PDG using a modified CFG with the speculated control flow edges removed. This newly built PDG will naturally not have the dependences eliminated by control speculation. Then, the PDG is updated to reflect the results of memory speculation by removing all the dependence edges indicated by those speculations. The resulting PDG is fed to the partitioning heuristic to decide the allocation of operations to threads.

5.2 Unspeculation

After the partitioning heuristic allocates instructions to threads, SpecDSWP determines what speculations must be retained to ensure pipelined execution and what speculations can be undone to lower runtime misspeculation. SpecDSWP must ensure that all speculations that removed backward dependences (i.e., dependences that flow from later pipeline stages to earlier stages, as determined by the partitioning heuristic) are retained. All other speculations can be undone. Unfortunately, identifying the smallest set of speculations to retain is non-trivial.

The example in Figure 5.3(b) illustrates this difficulty. Consider the scenario where the SpecDSWP compiler tentatively speculates that the edges marked 1 and 2 will not execute, and the partitioning heuristic allocates instructions A and D to stage 1, instruction

B to stage 2, and instruction C to stage 3. After partitioning, there is one speculatively removed backward dependence, the data flow dependence between instruction C (in stage 3) and instruction A (in stage 1). This dependence is removed *either* by speculating that the control edge marked 1 will not execute (because the instruction C becomes unreachable) *or* by speculating that the control edge marked 2 will not execute (making instruction D's definition of x unconditional). It is unnecessary for the compiler to retain both speculations. Consequently, the compiler has choice in its unspeculation.

While not shown by the example, it is also possible that a collection of speculations jointly eliminate a backwards dependence. Given that choice exists and that collections of speculations may jointly eliminate dependences, the compiler must consider all possible combinations of retained speculations to determine which are legal, and among the legal ones, estimate which would lead to the least misspeculation at runtime. Since, as the previous section showed, it is difficult to determine what dependences exist for a given set of *control* speculations, it would appear that the compiler needs to build a speculative PDG, and therefore run a suite of data flow analyses, for all possible combinations of retained speculation. Given the vast number of options and the expense of running data flow analysis, this solution would be untenable.

However, recall that speculation does not affect the results of data flow analysis provided that the speculation does not affect the control flow graph.[1] Consequently, only one set of data flow analysis needs to be run per potential CFG (i.e., per combination of retained control speculation). While this too would be prohibitively expensive, this section introduces *conditional analysis* which conceptually analyzes all the possible CFGs simultaneously. Rather than analyzing each CFG independently, a conditional analysis analyzes a single *conditional CFG* that conceptually represents all possible CFGs created by control speculation. The results of conditional analysis can be used to build a *conditional PDG* that the compiler can use to determine what speculation to retain.

[1] In particular, memory speculation only removes dependences from the speculative PDG; it does not affect the results of data flow analysis.

The next section will describe the conditional analysis in detail, and the subsequent section will describe how the results of this analysis, combined with the set of non-control speculations can be used to build the conditional PDG. Finally, this section concludes with the algorithm to unspeculate dependences based on the conditional PDG.

5.2.1 Conditional Analysis

Classical data flow analyses (e.g., reaching definitions, liveness, dominators, etc.) can be formulated in the gen-kill data flow framework [6]. In this framework, a data flow analysis is defined by a set of equations formed from the CFG that relate information at various program points. In these equations, there are two variables per node in the CFG being analyzed: one corresponding to the data flow facts at the entry to the node in_n, and one corresponding to the data flow facts at the exit to the node out_n. Data flow facts come from a predefined set of values V for the analysis, and the variables in_n and out_n take on values that are subsets of V. The result of the data flow analysis is obtained by simultaneously solving the set of equations.

The equations defining the data flow analysis having the following form:

$$\mathrm{out}_n = \mathrm{gen}_n \cup (\mathrm{in}_n \setminus \mathrm{kill}_n) \tag{5.1}$$

$$\mathrm{in}_n = \bigoplus_{m \in \mathrm{pred}(n)} \mathrm{out}_\mathrm{m} \tag{5.2}$$

In Equation 5.1, gen_n and kill_n are given by the analysis. In Equation 5.2, the analysis decides whether the *confluence* operator ($\oplus$) is either set union or intersection. Finally, $\mathrm{pred}(n)$ returns the set of predecessor nodes for n in the CFG. These equations can be solved by iteratively evaluating them until in_n and out_n converge using initial values for in_n provided by the analysis.

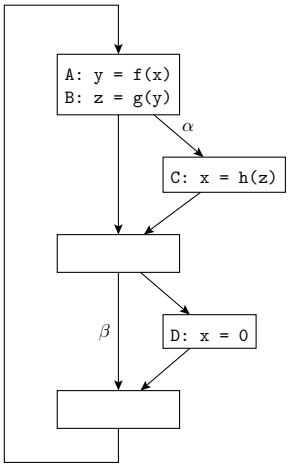
For example, reaching definitions fits into the framework as follows. The set of data flow facts V is the set of instruction, destination register pairs. The gen set for an instruction

is the subset of V where the instruction in the pair is the given instruction. Similarly, the kill set for an instruction is the subset of V where the destination register is contained in the set of registers defined by the given instruction. Finally, the confluence operator is set union and the initial values for in_n are $\emptyset$. Conceptually, this parameterization of the framework has definitions flow forward along all control flow paths until a redefinition halts the flow.

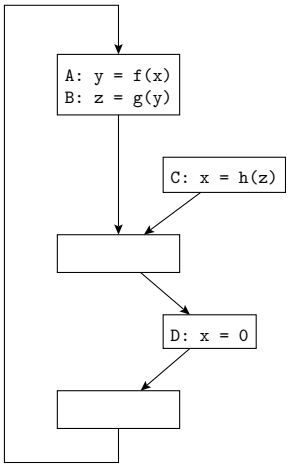
To extend gen-kill analyses to conceptually operate on many control flow graphs simultaneously, the set of possible control flow graphs are represented in a single conditional CFG. Figure 5.4(a) shows a conditional CFG, and Figures 5.4(b)–(e) show the corresponding traditional CFGs represented by it. A conditional CFG is identical to a traditional CFG except some control flow edges are annotated with boolean variables (in the figure, two edges are labeled with the variables α and β respectively). If the boolean variable associated with an edge is true, then the edge is considered to exist, otherwise it is considered absent. Consequently, each variable assignment corresponds to a single traditional CFG.

Using conditional CFGs, traditional gen-kill data flow analyses can be adapted to generate a set of data flow facts conditioned with a boolean formula. Conceptually, the result of the conditional data flow analysis represents the results of the traditional data flow analysis on *all* the traditional CFGs represented by the conditional CFG. For any particular variable assignment, substituting the variables' values into the boolean formula associated with a data flow fact determines whether the fact exists in the traditional analysis operating on the corresponding traditional CFG.

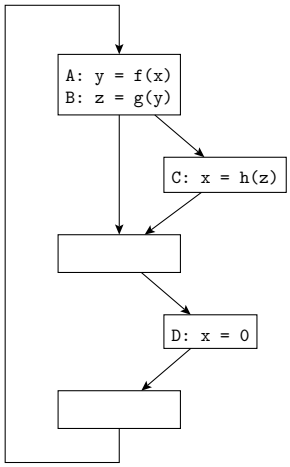
While conditional analysis may resemble predicate-aware data flow analysis [11], the two analyses are quite different. Predicate-aware data flow analysis reconstructs the control flow graph for predicated code in a single hyperblock (basic block with predicated code). By performing analysis on the reconstructed control flow graph, it enables "path" sensitive analysis within a hyperblock. On the other hand, by annotating control flow edges with boolean predicates, conditional analysis effectively provides path sensitivity across many blocks (rather than within a single block) in the control flow graph.



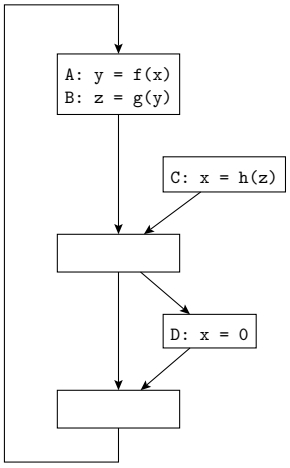
(a) Conditional CFG



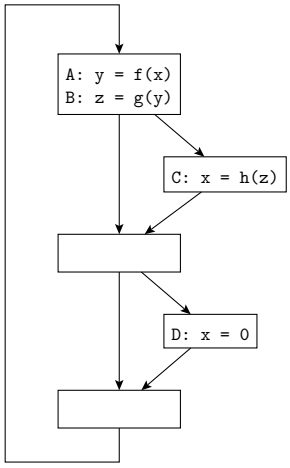
(b) $\bar{\alpha}\bar{\beta}$



(c) $\alpha\bar{\beta}$



(d) $\bar{\alpha}\beta$



(e) $\alpha\beta$

Figure 5.4: Figures(b)–(e) show the traditional CFGs represented by the conditional CFG in Figure (a). Each traditional CFG is labeled with the variable assignment that realizes it.

To see how to make a gen-kill analysis conditional, first consider a simplified analysis where data flow facts are just boolean values rather than sets.[2] One can think of this simplification as the "slice" of the data flow analysis with respect to one element $v \in V$. Making this analysis conditional, only requires modifying the confluence operator. Assuming the traditional analysis's confluence operator is $\vee$ ($\cup$), then the boolean values entering a node from all of its predecessors get logically ORed. This can be conditioned using variables annotated on the conditional CFG edges. Using $c_{1,n}$, $c_{2,n}$, ..., $c_{m,n}$ to denote the variables annotated onto the incoming control edges for node n, the new confluence will be:

$$\mathrm{in}_n = \bigvee_{m \in \mathrm{pred}(n)} (c_{m,n} \wedge \mathrm{out_m}) \tag{5.3}$$

Notice, using the variable assignment that reduces a conditional CFG to a particular traditional CFG, the conditional confluence operator reduces to the the traditional confluence operator. If an edge is absent, its corresponding variable will be 0, and by logically ANDing it to the incoming data flow fact, its effect is nullified. Similarly, for an edge that is present, its corresponding variable will be 1, and the logical AND operation is just the identity function. Leveraging the duality between the two confluence operators, the conditional confluence operation corresponding to $\wedge$ ($\cap$) is:

$$\mathrm{in}_n = \bigwedge_{m \in \mathrm{pred}(n)} (\neg c_{m,n} \vee \mathrm{out_m}) \tag{5.4}$$

Unfortunately, in this formulation, the conditional confluence operator is not constant for the whole analysis, but rather varies per node in the CFG. This is unsatisfying because it deviates from the underlying theory behind data flow analysis; one cannot define a lattice over the data flow facts. However, the formulation can be salvaged by using the traditional confluence operators and introducing a transfer function (i.e., a gen and kill) along each

[2] All the data flow equations are still valid since both ($\{\mathrm{true}, \mathrm{false}\}$, $\wedge$, $\vee$, $\neg$) and (V, $\cup$, $\cap$, $\neg$) are boolean algebras.

edge in the conditional CFG that is annotated with a variable. The transfer function to be used is:

$$\mathrm{out}_e = \mathrm{false} \vee (\mathrm{in}_e \wedge \neg (\neg c_e)) \qquad \textit{(if confluence is } \vee\textit{)} \tag{5.5}$$

$$\mathrm{out}_e = \neg c_e \vee (\mathrm{in}_e \wedge \neg \mathrm{false}) \qquad \textit{(if confluence is } \wedge\textit{)} \tag{5.6}$$

For the confluence operator $\vee$, the gen value is false and the kill value is $\neg c_e$. Following duality, for the confluence operator $\wedge$, the gen value is $\neg c_e$ and the kill value is false.

Recognizing that for an edge between nodes m and n, $\mathrm{in_{m,n}} = \mathrm{out_m}$, for the confluence operator $\vee$ substituting Equation 5.5 into Equation 5.2 yields Equation 5.3.

$$\mathrm{in}_n = \bigvee_{m \in \mathrm{pred}(n)} \mathrm{out}_{m,n} \tag{5.7}$$

$$\mathrm{in}_n = \bigvee_{m \in \mathrm{pred}(n)} (\mathrm{false} \vee (\mathrm{in}_{m,n} \wedge \neg (\neg c_{m,n}))) \tag{5.8}$$

$$\mathrm{in}_n = \bigvee_{m \in \mathrm{pred}(n)} (\mathrm{in}_{m,n} \wedge c_{m,n}) \tag{5.9}$$

$$\mathrm{in}_n = \bigvee_{m \in \mathrm{pred}(n)} (\mathrm{out}_m \wedge c_{m,n}) \tag{5.10}$$

The substitution works similarly for the operator $\wedge$.

$$\mathrm{in}_n = \bigwedge_{m \in \mathrm{pred}(n)} \mathrm{out}_{m,n} \tag{5.11}$$

$$\mathrm{in}_n = \bigwedge_{m \in \mathrm{pred}(n)} (\neg c_{m,n} \vee (\mathrm{in}_{m,n} \wedge \neg \mathrm{false})) \tag{5.12}$$

$$\mathrm{in}_n = \bigwedge_{m \in \mathrm{pred}(n)} (\neg c_{m,n} \vee \mathrm{in}_{m,n}) \tag{5.13}$$

$$\mathrm{in}_n = \bigwedge_{m \in \mathrm{pred}(n)} (\neg c_{m,n} \vee \mathrm{out}_m) \tag{5.14}$$

Thus far, the conditional analysis formulation has been with respect to one value $v \in V$.

A complete conditional analysis is built by repeating the process for each value in V. The Section 5.2.2 will describe a practical implementation that simultaneously computes the results of a conditional analysis for all members of the set V.

Before describing the practical implementation of conditional analysis, let us re-consider the example in Figure 5.4(a) to illustrate how conditional analysis works in the context of reaching definitions analysis. Since only two edges have been annotated with boolean variables, for all the other edges the transfer function is simply the identity transfer function. For the two annotated edges, since the confluence operator for reaching definitions is set union, the transfer functions are:

$$\text{out}_\alpha = \text{in}_\alpha \wedge \alpha \tag{5.15}$$

$$\text{out}_\beta = \text{in}_\beta \wedge \beta \tag{5.16}$$

In the first iteration of the analysis, along the edge marked α,, the incoming data flow facts are $\{(A, y) : \top, (B, z) : \top\}$. Applying the edge transfer function, the incoming data flow facts at node C are $\{(A, y) : \alpha, (B, z) : \alpha\}$. Propagating the analysis through node C yields the data flow facts $\{(A, y) : \alpha, (B, z) : \alpha, (C, x) : \top\}$. Applying the confluence operator at the merge point leaves the data flow facts unchanged. Propagating them through the edge marked β yields $\{(A, y) : \alpha \wedge \beta, (B, z) : \alpha \wedge \beta, (C, x) : \beta\}$. Applying the confluence operator at the second merge point yields $\{(A, y) : \alpha \wedge \beta, (B, z) : \alpha \wedge \beta, (C, x) : \beta, (D, x) : \top\}$. The analysis would continue until it converged. The analysis reveals that there are two reaching definitions for the variable x at the header of the loop $\{(C, x) : \beta, (D, x) : \top\}$. Looking at the conditions, one observes that if β is false, the definition from instruction C does not reach the loop header, making the definition from instruction D an unconditional define. One may have expected the definition from instruction C to have also been conditioned on α. However, since reaching definition analysis propagates forward through the control flow graph, α does not condition the definition coming from statement C. Sec-

tion 5.2.3 will demonstrate how to merge the results of conditional reaching definition analysis with conditional reachability analysis to ensure that unreachable instructions do not cause dependence edges to appear in the conditional PDG.

5.2.2 Practical Implementation of Conditional Analysis

To practically implement a conditional analysis there are two principle obstacles to overcome. First, since the analysis iterates until it reaches a fixed point, to determine whether a fixed point has been reached it is necessary to check if two boolean formulae are equal. For this check to be efficient, it requires a canonical form for boolean formulae. Second, the conditional analysis must be able to concurrently operate on many data flow facts (members of the set V) simultaneously. Implementations that require one round of analysis per value are untenable because $|V|$ often scales with the number of nodes in the CFG and CFGs can exceed thousands of nodes.

The first obstacle is overcome using reduced ordered binary decision diagrams (BDD) [16]. A BDD represents a boolean formula using a directed acyclic graph. Each node in the graph represents a boolean variable, the value true, or the value false. Each boolean variable node has two out edges. The target of one edge is connected to the subgraph corresponding to the function if the variable is true, and the other edge is connected to the subgraph corresponding to the function if the variable is false. The true and false nodes have no out edges. One can evaluate the boolean function represented by a BDD by traversing from the root of the BDD to one of the terminal nodes, following the edges corresponding to the variable assignment. If the traversal ends at the true (false) node, then the function is true (false). If BDDs are reduced and the variables ordered, they canonically represent a boolean function. Additionally, formulae can be efficiently combined (i.e., logically ANDed or ORed together). Not only do reduced ordered BDDs offer efficient equality checks between formulae, they also conserve memory by using only a single in-memory data structure for common sub-expressions between two formulae.

The second obstacle is overcome by recognizing that a set of $(\text{value}, \text{logical formula})$ pairs can be represented using a boolean formula. Further, any operation that need be performed between all corresponding formula in two sets can, instead, directly be performed on the formula representing the two sets. Consequently, the conditional data flow analysis can be performed on all values simultaneously. The encoding that achieves this is as follows. First, $B = \lceil \log_2 |V| \rceil$ variables are used to represent values in the set V. Each assignment of these B variables uniquely identifies one element from the set V. Consequently each value $v \in V$ can be represented with a minterm m_v. A pair (v, F), where $v \in V$ and F is a boolean formula, can then represented with the boolean formula $m_v \wedge F$, and a set of such pairs can be represented by taking the disjunction of the formula for each pair, $(m_{v_1} \wedge F_{v_1}) \vee (m_{v_2} \wedge F_{v_2}) \vee \ldots$. In this representation, the formula for a particular value v can be extracted by restricting the set's formula with m_v. Further, observe that the logical OR of formulae representing two sets is equal to the formula representing the logical OR of corresponding formulae from each set.

$$\Big((m_{v_1} \wedge F_{v_1}) \vee (m_{v_2} \wedge F_{v_2}) \vee \ldots\Big) \vee \Big((m_{v_1} \wedge F'_{v_1}) \vee (m_{v_2} \wedge F'_{v_2}) \vee \ldots\Big) = \tag{5.17}$$
$$\big(m_{v_1} \wedge (F_{v_1} \vee F'_{v_1})\big) \vee \big(m_{v_2} \wedge (F_{v_2} \vee F'_{v_2})\big) \vee \ldots$$

Similarly, the logical AND of formulae representing two sets is equal to the formula representing the logical AND of corresponding formulae from each set.

$$\Big((m_{v_1} \wedge F_{v_1}) \vee (m_{v_2} \wedge F_{v_2}) \vee \ldots\Big) \wedge \Big((m_{v_1} \wedge F'_{v_1}) \vee (m_{v_2} \wedge F'_{v_2}) \vee \ldots\Big) \tag{5.18}$$
$$= \big(m_{v_1} \wedge (F_{v_1} \wedge F'_{v_1})\big) \vee \big((m_{v_1} \wedge m_{v_2}) \wedge (F_{v_1} \wedge F'_{v_2})\big) \vee$$
$$\big(m_{v_2} \wedge (F_{v_2} \wedge F'_{v_2})\big) \vee \big((m_{v_1} \wedge m_{v_2}) \wedge (F'_{v_1} \wedge F_{v_2})\big) \vee \ldots$$
$$= \big(m_{v_1} \wedge (F_{v_1} \wedge F'_{v_1})\big) \vee \big(m_{v_2} \wedge (F_{v_2} \wedge F'_{v_2})\big) \vee \ldots \tag{5.19}$$

The above equality holds since $m_i \wedge m_j$ is always false if $i \neq j$. Finally, the logical NOT of

a set's formula is equal to the formula for the set where each contained formula is inverted.

$$\neg\Big((m_{v_1} \wedge F_{v_1}) \vee (m_{v_2} \wedge F_{v_2}) \vee \dots\Big) = (\neg m_{v_1} \vee \neg F_{v_1}) \wedge (\neg m_{v_2} \vee \neg F_{v_2}) \wedge \dots \quad (5.20)$$

$$= \left(\left(\bigwedge_{i \neq 1} \neg m_{v_i}\right) \wedge \neg F_{v_1}\right) \vee \left(\left(\bigwedge_{i \neq 2} \neg m_{v_i}\right) \wedge \neg F_{v_2}\right) \vee \dots \quad (5.21)$$

$$= (m_{v_1} \wedge \neg F_{v_1}) \vee (m_{v_2} \wedge \neg F_{v_2}) \vee \dots \quad (5.22)$$

The above equality holds since $\bigwedge_{i \neq j} \neg m_{v_i} = m_{v_j}$ and the terms not shown in the equations are subsumed by the terms shown.

5.2.3 Building the Conditional PDG

These conditional analyses can be used to build a *conditional PDG* by annotating dependence edges in the PDG with the condition guarding their existence. In particular, register flow dependences can be computed with conditional reaching definition analysis, control dependences can be computed using conditional post dominator analysis, and memory dependences can be computed with a conditional memory analysis. Additionally, these analyses can be further refined using conditional reachability analysis.

Returning to the example in Figure 5.4(a), conditional reaching definition analysis would conclude that the reaching definitions for the variable x at the instruction A is $\{(C, x) : \beta, (D, x) : \top\}$. Consequently, the conditional PDG would have a data dependence edge between instruction C and A, annotated with the condition β, and another data dependence edge between D and A, annotated with the condition true.

However, no dependence can exist between instructions C and A if instruction C is unreachable. Therefore, conditional reachability analysis can be used to refine the results of the conditional reaching definition analysis. Traditional reachability analysis identifies what CFG nodes are reachable from the entry of the control flow graph. Analogously, conditional reachability defines the condition under which each CFG node is reachable from

the CFG entry. Using conditional reachability, the condition on any dependence between two nodes can be refined by logically ANDing the reachability condition for each of the nodes. For the register dependence edge between instructions C and A in Figure 5.4(a), reachability analysis would conclude that instruction A is always reachable, and instruction C is reachable if α is true. Thus, the condition on the dependence edge would be $\alpha \wedge \beta$. Such refinements work equally well for register, control, and memory dependences. In fact, since most points-to memory analyses do not fall directly into the gen-kill data flow framework, refinement through reachability is the most expedient method of incorporating differences in control flow due to speculation with memory analysis.

Thus far, the edges in the conditional PDG are conditioned only by the effects of control speculation. Control speculation introduces variables in the conditional PDG which propagate through the various conditional analyses onto the dependence edges in the PDG. If any such variable is true, the control flow edge is presumed to exist meaning that the corresponding speculation has been discarded. Conversely, if any such variable is false, the control flow edge is presumed absent meaning the corresponding speculation has been retained. To account for the affects of non-control speculation (i.e., alias speculation and committed value speculation), new boolean variables are introduced for each non-control speculation. Similarly to control speculation, if the variable corresponding to a non-control speculation is false, any dependences removed by the speculation should be absent from the conditional PDG. Recall that the relation between non-control speculation and dependences removed from the PDG is straightforward, unlike for control speculation. Consequently, for each dependence edge eliminated by a non-control speculation, its condition in the conditional PDG should be logically ANDed with the boolean variable associated with the speculation.

5.2.4 Finalizing Speculation

After building the conditional PDG, for a given set of retained speculation, the compiler can quickly determine which dependence edges will exist and which will be absent. This computation is at the heart of deciding the final set of speculations to retain. Given a partition, to ensure that it corresponds to pipeline parallelism, the compiler must ensure that no dependences flow from later pipeline stages to earlier ones. Thus, the compiler collects this set of backwards dependences from the conditional PDG. For each such dependence, the compiler must ensure that the condition on the dependence edge is false. While the compiler could trivially set all variables to false (i.e., the compiler could retain all the provisional speculations), the compiler would also like to eliminate unnecessary speculation to minimize the runtime cost of misspeculation.

Finding a set of speculation that guarantees pipelined execution and that minimizes the number of speculations can be found by formulating the problem as an instance of MinCostSAT [42]. MinCostSAT is a variation of the boolean satisfiability problem where variables are assigned a cost. A solution to the problem is a variable assignment that satisfies a given boolean formula and that minimizes the sum of the costs for the variables set to `true`. The unspeculation problem can be formulated as MinCostSAT as follows. If the condition on a dependence edge is false, then the set of speculations eliminates the dependence. Since the compiler requires all such backwards edges to be speculated, any variable assignment that satisfies the logical AND of the complement of the conditions of all backwards dependence edges in the conditional PDG is sufficient. Next, since each variable in the formula corresponds to a speculation, the compiler can assign a cost to each variable based on a misspeculation estimate for the given speculation. Finally, the resulting formula and costs can be fed to a MinCostSAT solver, such as MinCostChaff [28], to identify the set of speculations to retain. Note, since a speculation is discarded if its corresponding variable is true, the compiler wishes to maximize the number of variables set to true. Since MinCostSAT tries to reduce the number of true variables, before passing the problem to

MinCostSAT, all the variables should be inverted.

Let us once again return to the example from Figure 5.4(a). Assume that the compiler tentatively speculated the control flow edges marked α and β in the figure. Further assume that the partitioning heuristic assigned instructions A and D to thread 1, instruction B to thread 2, and instruction C to thread 3. The only backwards dependence in the conditional PDG would be between instructions C and A. The condition on this dependence would be $\alpha \wedge \beta$. The complement of this formula is $\neg\alpha \vee \neg\beta$. To prepare the formula for MinCostSAT, all the variables must be inverted. Letting $\alpha' = \neg\alpha$ and $\beta' = \neg\beta$, then the formula that would be passed to the MinCostSAT solver is $\alpha' \vee \beta'$. The compiler would then estimate the misspeculation rate for each of the two speculations and assign an appropriate cost. Assuming the cost for α' is greater than the cost for β', the MinCostSAT solver would return the satisfying assignment $\alpha' = \text{false}, \beta' = \text{true}$. Consequently, $\alpha = \text{true}$ and $\beta = \text{false}$ meaning that the edge marked α would *not* be speculated, while the edge marked β would be speculated.

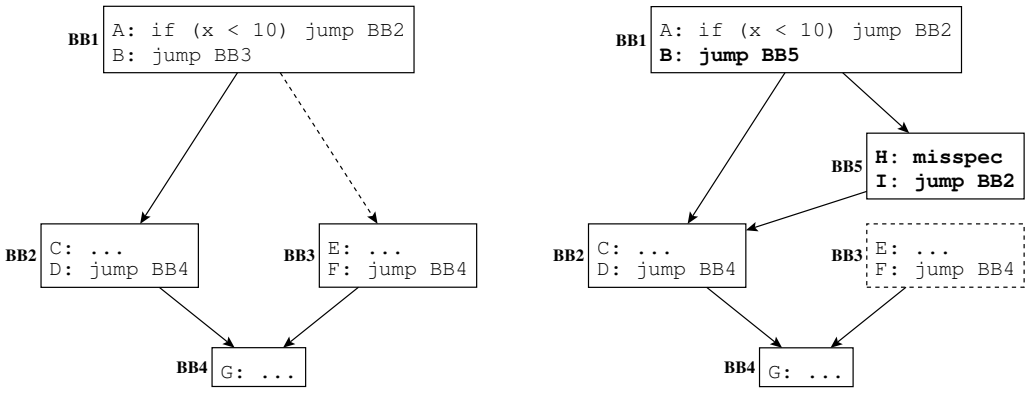
Chapter 6

Code Generation

The previous chapter described how a SpecDSWP compiler decides what to speculate. This chapter describes how the base DSWP code generation algorithm is extended to implement this speculation. Code generation involves three principle parts. First, code must be inserted to realize the speculation; branches must be updated for control speculation, value predictors must be inserted for committed value speculation, branches inserted for silent store speculation, etc. Second, code must be inserted to *detect* misspeculation. Finally, code must be inserted to *recover* from misspeculation once it is detected. This chapter will describe each of these parts, and it will describe the necessary runtime support for misspeculation detection and recovery.

6.1 Realizing Speculation and Detecting Misspeculation

To implement speculation and detect misspeculation SpecDSWP inserts code directly into the single-threaded program rather than inserting it during or after multi-threaded code generation (MTCG). Code must be inserted for each speculation retained during the first phase of compilation. After inserting the code, the compiler updates the partition provided by the partitioning heuristic to include newly added instructions and to elide instructions that were removed. After updating the partition, the non-speculative MTCG algorithm can



(a) Before (b) After

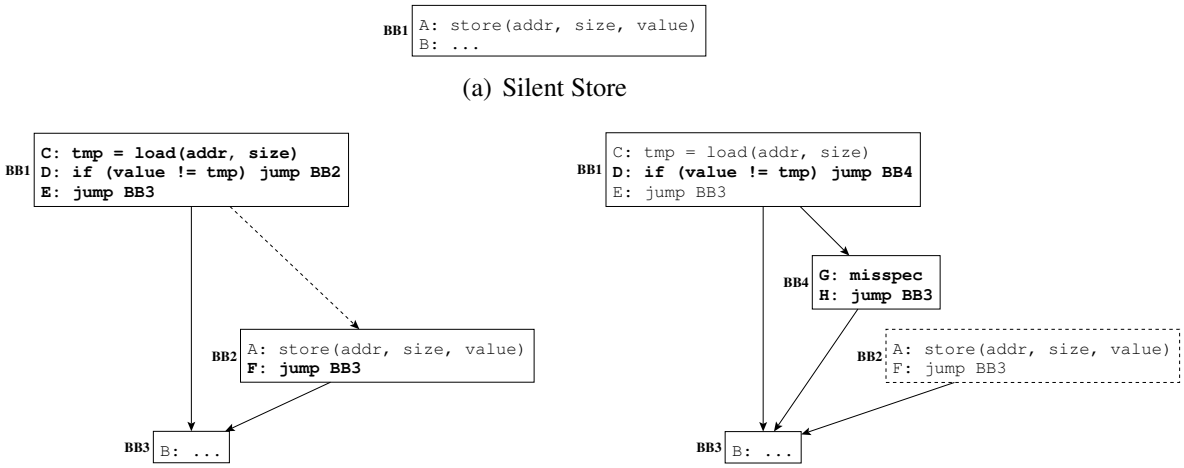
Figure 6.1: This figure illustrates how control misspeculation is detected. Misspeculation code is inserted along the speculated control flow edge (the dashed edge). The misspeculation code then jumps back into the normal program flow to eliminate all control dependences due to the speculated control flow path.

be used to generate the parallel code. The next few sections describe the transformation that occurs for each speculation type.

6.1.1 Control Speculation

Recall from the previous chapter that biased branch speculation and infrequent block speculation both ultimately specify an edge in the control flow graph that is unlikely to be traversed. To realize this speculation, the code must be transformed to remove any control or data dependences arising due to program paths including the speculated edges. Additionally, misspeculation must be detected whenever the speculated edge would have been executed. Figure 6.1 illustrates how code is transformed to achieve this. Assume that the dashed edge in Figure 6.1(a) has been speculated not to execute. At runtime, if `x` were greater than 10, then misspeculation has occurred. Misspeculation is detected by redirecting the speculated edge to a new basic block. In Figure 6.1(b), this corresponds to updating operation B to point to basic block 5. The code in this new block is responsible for initiating misspeculation recovery (represented by operation H in the figure).

At runtime, once misspeculation has been detected, the program state will be reverted



(a) Silent Store

(b) After Silent Store Check

(c) After Control Speculation

Figure 6.2: This figure illustrates how silent store misspeculation is detected. First, the store is translated into a hammock that checks to see if it is silent (Figure (b)). Then, control speculation is applied to the edge going to the store (Figure (c)).

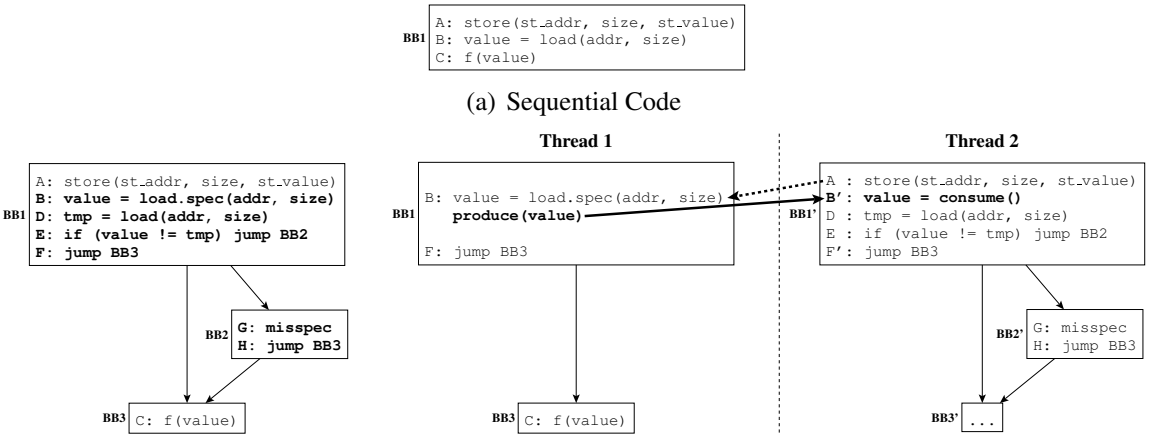
to the state at the beginning of the current iteration, and the iteration will be re-executed non-speculatively. Consequently, the compiler should understand that the `misspec` operation restores program state (i.e., defines all live registers and memory locations) and jumps back to the loop header (or loop exit if the iteration that misspeculated is the final loop iteration). If this approach were used, subsequent data flow analysis would correctly identify that any *data* dependences arising due to program paths including the speculated edge are now absent. However, *control* dependences due to the speculated edge would still exist since the controlling branch's outcome determines whether or not recovery code executes. To avoid this, our SpecDSWP implementation treats the `misspec` instruction like a no-op (it acts only as a place holder) and inserts an unconditional jump to one of the other targets of the source basic block. In the figure, operation I jumps to basic block 2. This transformation eliminates both the control and data dependences arising from the speculated edge by folding the paths originating at the speculated edge into existing paths in the program. The corresponding branch is retained only for misspeculation detection.

6.1.2 Silent Store Speculation

Silent store speculation is implemented by inserting code that conditionally executes the speculated store only when the value to be stored is *different* than the value already in memory. Since the store is frequently silent, the inserted branch controlling the store is biased, and is speculated using the control speculation transformation described in the previous section.

Figure 6.2 illustrates the transformation. Figure 6.2(a) shows the silent store and Figure 6.2(b) shows the code after the store is made conditional. Operations C, D, E in basic block 1 compare the value currently stored in memory with the value to be stored. If the values are different, the code jumps to block 2 and executes the store. Otherwise, the store is skipped by jumping to block 3. Figure 6.2(c) illustrates the code after the control speculation transformation is applied. The code in block 2 is replaced with a `misspec` operation and a jump to block 3. After the transformations are applied, the store instruction is no longer reachable, thus future data flow analyses will no longer find any data dependences originating from the speculated store.

In addition to the transformation just described, the compiler must allocate operations C, D, and G (the three new operations, excluding unconditional jumps, inserted by the transformation) to some pipeline stage. Operation C depends on the values `addr`, `size`, and the value stored in the corresponding memory location. Similarly operation D depends on `value`. If the silent store was allocated to thread t, then `addr`, `size`, and `value` can be communicated to thread t without introducing any backward dependences. However, the value in memory at address `addr` may not be available in thread t because it may be written in some thread $t' > t$. However, since operation C only feeds operation D, operation D only controls operation G, and operation G is not the source of any dependence, the compiler is free to move this group of instructions to any later thread. In particular, if t' is the latest thread that writes to the address `addr`, then the compiler must choose some thread $t'' \geq t'$. Our SpecDSWP implementation will allocate these operations to thread t'', however an



(a) Sequential Code

(b) Value Speculation and Alias Check

(c) Partitioned Code

Figure 6.3: This figure illustrates how alias and committed value speculation is implemented. The original load is converted into a speculative load to break the dependence (operation B in Figure (b)), and value misspeculation code is inserted to detect misspeculation (operations D–H in (b)). When the code is partitioned (Figure (c)), the value speculation is allocated to an early thread, and the misspeculation detection is allocated to a late thread. In the figure the solid thread to thread arrows represent synchronized dependences, whereas the dashed arrows represent unsynchronized speculated dependences.

implementation may use more sophisticated heuristics to ensure these new operations do not significantly upset the thread load balance.

6.1.3 Alias Speculation and Committed Value Speculation

Consider the example in Figure 6.3. Assume that in the single-threaded program alias analysis determines that the load, operation B in Figure 6.3(a), potentially aliases with the store, operation A. Further, profiling reveals this dependence is in infrequent so it can be speculated. The partitioner then chooses to allocate the load to thread 1 and the store to thread 2. Since this creates a backwards memory dependence from thread 2 to thread 1, the code generator must insert code to break this dependence. To implement this speculation, the code generator converts the alias speculation into a value speculation. This requires inserting a software value predictor and a software misspeculation detector. Figure 6.3(b) illustrates this new code in bold. First, the original load instruction is converted to a specu-

lative load instruction. A speculative load differs from its non-speculative counterpart only in that the multi-threaded code generator knows that it need not synchronize the load with potentially aliasing stores. To avoid speculating *all* store to load memory dependences at a particular load, the load is annotated with the points-to set that need not be synchronized. Next, a non-speculative duplicate load is inserted (operation D in the example). The result of the duplicate load is then compared to the speculative load, and if the results differ, misspeculation is flagged (operations E–H). Otherwise execution continues unfettered.

While the duplicate misspeculation-detecting load seems redundant with the speculative load (implying misspeculation will *never* be detected), its purpose becomes evident after considering the code produced by MTCG illustrated in Figure 6.3(c). In the partitioned code, the speculative load (operation B) and its dependent operation (operation C) are allocated to thread 1. The misspeculation detection code, including the redundant load, are allocated to thread 2, the thread containing the store operation (operation A). The MTCG will communicate the value loaded by the speculative load to thread 2 so it can compare the speculative and non-speculative values (illustrated by the solid arrow connecting the produce and consume operations in Figure 6.3(c)). However, the backwards memory dependence between operation A and B is *not* synchronized (illustrated by the dashed arrow between operations A and B in Figure 6.3(c)). Consequently, the speculative load can execute *before* the store operation. Since the load and store alias only infrequently, the speculative load typically produces the correct value. However, if the load executes before the store, *and* the load and store reference the same address, then misspeculation has occurred. The speculative load (operation B) will have read the wrong value (provided the store is not silent) since it will have executed *before* the store, and the duplicate load (operation D) will read the correct value since it always executes after the store. Since the two values differ, the misspeculation comparison (operation E) will cause thread 2's control flow to be directed to block 2, and misspeculation will be flagged.

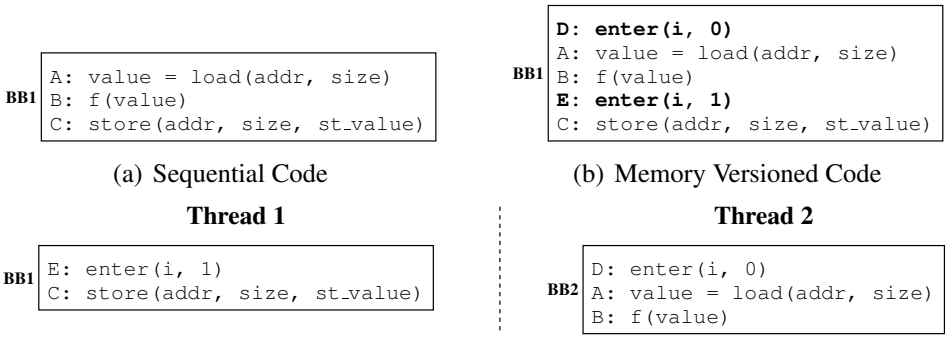
The code generator is responsible for allocating the code it inserts to pipeline stages.

The value speculative load remains in the thread where the original load was allocated. The misspeculation detection code (including the redundant load) is allocated to the thread containing the store operation, or some later thread. This guarantees the backwards flow memory dependence is broken, and no new backwards flow dependences are created. However, new backwards false memory dependences can be created by the redundant load. For example, if another potentially aliasing store were allocated to thread 1, then such a backwards dependence would be created. The next section will discuss how such backwards false memory dependences are removed.

Committed value speculation differs from alias speculation only in the choice of value speculator. The alias speculation mechanism uses the latest value stored at the given location as a guess of the true value to be read. Committed value speculation however uses the value of the location stored in *committed* state or a value produced in the current loop iteration. As described earlier, this predictor works well when a memory location is changed frequently within an iteration, but is often constant across loop iterations. Unlike alias speculation where, at runtime, the speculative load is just a traditional load instruction, committed value speculation requires runtime support to perform the committed value load. This support will be described in more detail in the Chapter 7.

6.1.4 False Memory Dependence Removal

While alias speculation and committed value speculation can remove flow memory dependences from the program, false memory dependences (anti- and output-dependences) are handled differently. As the previous section demonstrated, the ability to remove false memory dependences is particularly important because, in addition to removing false memory dependences that naturally occur in the program, it must also be used to eliminate backwards false memory dependences created by the alias and committed value misspeculation detection code. Additionally, as Section 6.2 will demonstrate, all speculation can *create* new false memory dependences that were not discovered by the memory analysis. Ensur-



(a) Sequential Code

(b) Memory Versioned Code

(c) Partitioned Code

Figure 6.4: This figure illustrates how memory versioning can be used to break false memory dependences. In Figure (a), the load and store alias because the both access address `addr`. In Figure (b), the `enter` instructions cause the load and store to access different memory versions. Figure (c) illustrates that no synchronization is necessary after the versioned code is parallelized.

ing these dependences are not violated is critical to ensuring correct execution.

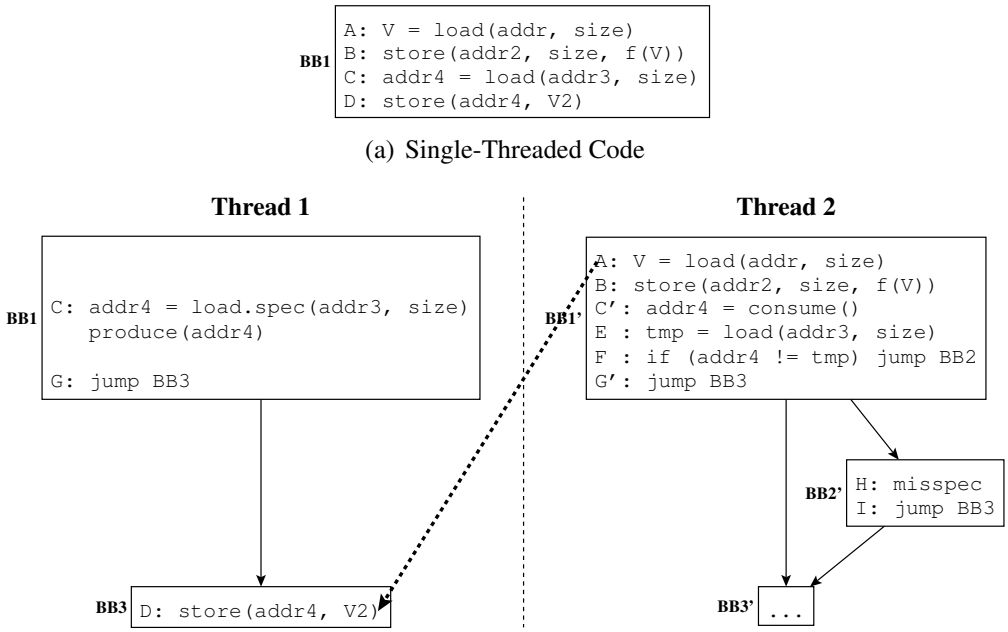
Consider the code shown in Figure 6.4(a). Since the load (operation A) and the store (operation C) access the same location, the two operations are anti-dependent. If the store were to execute before the load, the load would read `st_value` instead of what was previously stored in memory. Similar problems would occur for an output dependence between two stores. Consequently to guarantee correct execution, if the load and store are in different threads the two must be synchronized to ensure the load executes before the store. However, if the store is allocated to an early thread and the load is allocated to a later thread, the synchronization creates a *backwards* memory dependence. Consequently, synchronization cannot be used to enforce backwards false memory dependences.

Rather than synchronizing false memory dependences, our SpecDSWP compiler relies on runtime support to *version* memory. Memory versioning renames memory locations allowing a load or store to specify from which renamed instance (a version) it will read or to which it will write. Figure 6.4(b) illustrates how this would work in the example. Before the load, an `enter` operation is inserted which causes future memory operations to access the memory version $(\texttt{i}, 0)$, where `i` represents the current loop iteration number. Before the store, an `enter` operation is inserted which enters the memory version $(\texttt{i}, 1)$. In the

parallel code, Figure 6.4(c), the load and store can execute unsynchronized because the load explicitly specifies that it should read from $(\texttt{i}, 0)$ and the store side effects the later version $(\texttt{i}, 1)$. Versioning guarantees that loads always read the correct value and that the correct store persists in memory. Consequently, there is no need to insert code to detect misspeculation. Section 6.2 will explain how the compiler chooses where to insert `enter` operations and describes why two-dimensional version numbers are necessary. Chapter 7 will describe the semantics of a versioned memory system in detail and will also describe a concrete implementation.

6.2 Memory Versioning

This section describes how `enter` instructions are inserted into the code. The algorithm described in this section assigns a version to each instruction in the program. This version assignment is used to insert an `enter` operation before each load, store, or external (library) function call. Redundant `enter` operations are then eliminated using a post-pass. The algorithm guarantees that two operations are in different versions if there exists a backwards false memory dependence between them. This section begins by describing additional false memory dependences created by speculation. Then it describes the versioning algorithm for acyclic code regions without function calls. The algorithm is then generalized to handle loops by using two-dimensional version numbers, and finally to handle function calls and recursion.



(a) Single-Threaded Code

(b) Multi-Threaded Code

Figure 6.5: This figure illustrates how speculation can induce new false memory dependences. Figure (a) shows single-threaded code with no false memory dependences. Figure (b) shows the corresponding multi-threaded code assuming the flow memory dependence between operations B and C is speculated. The dashed arrow in the figure shows the false memory dependence created by the alias speculation.

6.2.1 Speculation Induced False Memory Dependences

This section will illustrate how speculation creates false memory dependences between operations that were previously had no false memory dependences between them. Ensuring these dependences are respected is critical to guaranteeing that the various software misspeculation detection mechanisms operate as expected. Consider the code shown in Figure 6.5(a). In the example, assume

1. `addr` is never equal to `addr2`, `addr3`, or `addr4`,
2. `addr2` is never equal to `addr4`, but is infrequently equal to `addr3`,
3. and `addr3` is never equal to `addr4`.

Since operation B only infrequently aliases with operation C, assume the SpecDSWP compiler speculates this dependence. Given this speculation, assume the compiler allocates operations C and D to thread 1, and operations A and B to thread 2. With this alloca-

tion, the compiler would generate the multi-threaded code shown in Figure 6.5(b). Notice, since there are no false memory dependences between any of the operations, no `enter` operations are necessary and all the operations execute in a single memory version.

Given this multi-threaded code, consider what happens if the operations C and D execute before operations A and B, and if operations B and C alias (i.e., misspeculation should be flagged). Under these conditions, operation C will read the wrong value because it executes before operation B and the two operations alias. Consequently, when operation D executes, it will store to the wrong address. It is possible that the address written to is `addr`, an address that the compiler statically proved could not be written to (in the absence of speculation). Consequently, when operation A executes, it reads the wrong value. The compiler did not anticipate that operation A would read the wrong value and consequently did not insert code to ensure correct execution. Reading this incorrect value is benign provided that thread 2 reaches the misspeculation detection code (operations C', E, F, and H). However, for example, it is possible that the incorrect value causes `f(V)` to loop infinitely thus preventing the code from ever reaching the misspeculation detection code.

If the compiler had anticipated the false memory dependence from operation A to operation D, then it would have ensured that the two operations were in different memory versions. This would have protected operation A from the side effects of operation D and thus would have guaranteed that the misspeculation code was executed.

To ensure that the effects of these speculation induced false memory dependences are considered, during versioning the compiler uses a set of synthetic false memory dependences in addition to the false memory dependences identified by memory analysis. The set of synthetic dependences is computed by first calculating the forward slice for each load that is the target of some speculated memory dependence (e.g., operation C in the example). Any store in the slice is considered *poisoned* because it can write to an unanticipated address. Consequently, a synthetic false memory dependence is created between all prior memory operations and the poisoned stores. The versioning algorithm described in the

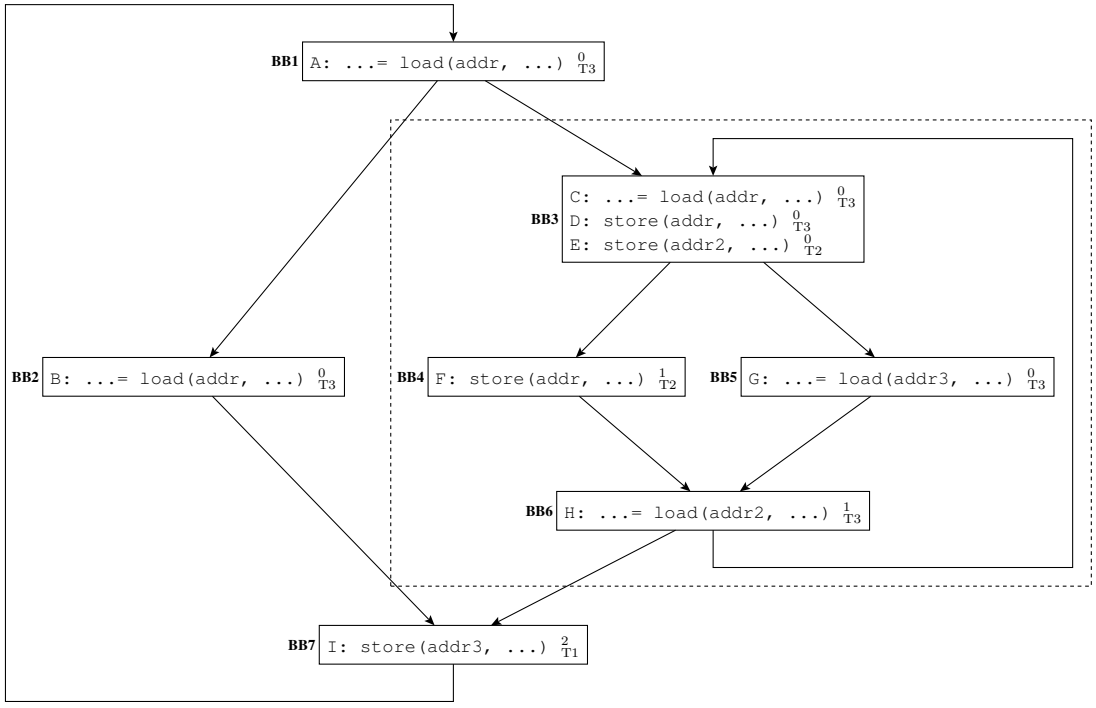


Figure 6.6: Example program demonstrating memory versioning. Only `load` and `store` operations are shown. The superscript adjacent to each operation represents the static memory version assigned to each operation. The adjacent subscript indicates to which thread the operation was allocated. The code within the dashed box is an inner loop and is versioned independently of the enclosing outer loop.

next few sections will guarantee that version numbers monotonically increase in the code. Consequently, when computing the forward slice it is unnecessary to traverse memory dependences since any store at the head of a memory dependence in the slice will be poisoned and monotonicity will guarantee that any subsequent operation is versioned appropriately. This greatly reduces the number of synthetic memory dependences that must be inserted.

6.2.2 Versioning Acyclic Regions

Given the memory dependences (including the synthetic false dependences) in the program, the compiler can now compute where to insert memory versioning code. This section will describe where memory versioning code must be inserted for an acyclic region without any function calls. The next two sections will generalize the algorithm to cyclic regions, and

regions that contain function calls.

Consider the acyclic code region within the dashed box in Figure 6.6 (ignoring the inner loop back edge). In the figure, the subscript $\mathrm{T}n$ next to each instruction indicates the thread to which the instruction has been allocated, and the superscript number indicates the version number assigned to each instruction by the algorithm described in this section. Ignoring the loop back edge and considering only the acyclic loop body, the code contains three false memory dependences,

1. the anti-dependence between C and D,
2. the anti-dependence between C and F,
3. and the output-dependence between D and F,

and one (forward) flow memory dependence, the dependence between E and H.

Since operations C and D are in the same thread, they are guaranteed to execute in order and no special memory versioning is required. However, since operations C and F are in different threads, and the backwards anti-dependence between them will not be synchronized, there is no guarantee that operation C will execute temporally before operation F. Consequently, for the backwards anti-dependence between C and F, the version assignment must ensure that even if operation F (a store) executes first, operation C (a load) reads the value that existed in memory before the store. To guarantee this, operation C's version number (0) must be less than operation F's version number (1). Similarly, for the backwards output dependence between operations D and F, the version assignment must be such that the value written by operation F persists in memory even if operation D executes temporally after operation F. Thus, operation D's version number (0) is less than operation F's version number (1). For the forward memory flow dependence, to ensure that the load H observes the result of the store E, the version assignment must ensure that operation H's version number (1) is greater than or equal to operation E's version number (0) even though the dependence will cause threads 2 and 3 to synchronize.

In general, to guarantee correct execution, the versioning algorithm must ensure two properties. First, if there is a backward false memory dependence between two operations then the version assigned to a dynamic instance of the source of the dependence must be strictly less than the version for a later (in the single-threaded execution) dynamic instance of the destination of the dependence. Second, if there is a flow memory dependence between two operations then the version assigned to a dynamic instance of the source of the dependence must be less than or equal to the version for a later dynamic instance of the destination of the dependence. The algorithm described below further guarantees that in any particular thread, version numbers increase monotonically to avoid unnecessarily jumping between memory versions. This latter requirement subsumes the flow dependence requirement for acyclic regions.

For an acyclic code region, these constraints suggest a graph based version allocation strategy. Given the control flow graph for an acyclic region of code, the instructions can be versioned in topological order. For each instruction, two version numbers are computed. The first, denoted P, is the maximum of the version numbers for the instruction's immediate predecessors. The second, denoted F, is the maximum of the version numbers for all instructions feeding this instruction through a false, backwards memory dependence. Because the algorithm processes instructions in topological order, if the source of a dependence is unversioned, the dependence must come from outside the region or be loop-carried. Since this algorithm is only versioning the given *acyclic* region, when computing P or F it is safe to assume the version number for such unversioned instructions is 0. Given P and F for an instruction, the version number given to the instruction is:

$$V = \max(P, F + 1) \tag{6.1}$$

The variable P in the maximum enforces the monotonicity constraint described earlier. The term $F + 1$ enforces the false dependence constraint, including the strict inequality of

version numbers. The version numbers for operations in the inner loop body (operations in the dashed box) in Figure 6.6 illustrate the result of this algorithm.

6.2.3 Handling Loops using Two-Dimensional Version Numbers

The *constraints* on version numbers described in the previous section are just as valid on cyclic regions as they are on acyclic ones. Unfortunately, the above *algorithm* breaks down when considering cyclic regions. Reconsider the code within the dashed box in Figure 6.6 *including* the loop back edge. When considering the cyclic region, it is no longer safe to ignore loop-carried dependences. Thus, the loop-carried flow memory dependence from operation F to C implies that the version number for operation F (1) in iteration i of the inner loop should be less than or equal to the version number for operation C (0) in iteration $j > i$. Similarly, the loop-carried, backwards, false memory dependence from operation H to operation E implies that operation H's version number (1) in iteration i should be strictly less than operation E's version number (0) in iteration $j > i$. Unfortunately, both these requirements are violated by the current version assignment.

These problems illustrate that different dynamic instances of a single static operation must be versioned differently. The obvious solution is to use a base + offset versioning strategy. With this solution, the acyclic versioning algorithm is run on a loop's body. Then, assuming N distinct version numbers were used, the version number for a dynamic instruction is $Ni + v$ where i is the loop iteration number, and v is the version number assigned by the acyclic versioning algorithm. Applying this to the example, where $N = 2$, observe that all the versioning constraints are satisfied. For the flow dependence from operation F to C, the dynamic assignment yields $v_{\mathrm{F}}^{i} = 2i + 1 \leq 2j + 0 = v_{\mathrm{C}}^{j}$ where v_{F}^{i} is the version assigned to operation F in iteration i and $j > i$. Similarly, for the backwards, false memory dependence from operation H to operation E, the dynamic versioning yields $v_{\mathrm{H}}^{i} = 2i + 1 < 2j + 0 = v_{\mathrm{E}}^{j}$.

Handling dependences between an inner loop and the body of an enclosing loop also

merits further consideration. Returning to Figure 6.6, consider the backwards anti-dependence between operations G and I. To guarantee correct execution, the dynamic version number for operation I in iteration i of the outer loop must be strictly greater than the dynamic version number for *all* dynamic instances (across all inner loop iterations) of operation G. While seemingly benign, handling this situation with a naïve versioning strategy will introduce *true* backwards data dependences. In a given iteration of the outer loop, $2T$ versions will be used by the inner loop, where T is the trip count of the inner loop. Consequently, operation I's dynamic version number must be larger than $2T$. However, observe that operation I has been allocated to thread 1, and no operation in the inner loop has been allocated to thread 1. Since thread 1 has no operations in the loop, it cannot count the number of iterations, and therefore the inner loop trip count must be communicated to it. However, only threads 2 and 3 can count the iterations and communicating this value back to thread 1 yields a backwards dependence.

This pattern is an instance of nested parallelism [5]. The outer loop has been parallelized and invokes the inner loop, which itself has also been parallelized. Speculative DSWP relies on two-dimensional version numbers to handle versioning in the presence of nested parallelism. The version numbers are assigned using a structural approach. First the loop nest tree is built[1] with the loop being parallelized at the root of the tree. Version assignment starts at the leaves of the loop nest tree and proceeds upward toward the outermost loop being parallelized. Operations in each loop's body are first assigned a static integer version number using the algorithm described in Section 6.2.2. However, if the loop body contains an inner loop, the inner loop is reduced to a single instruction in the parent loop's body. The inner loop body is assumed to load from and store to all possible addresses so the compiler assumes that all stores in the parent loop are anti-dependent on the inner loop and all loads in the inner loop are anti-dependent on stores from the parent loop. Further, it is assumed that these loads and stores in the inner loop exist in all threads for which some

[1]This approach only works if the code contains no irreducible loops. If the code contains irreducible loops, tail duplication can be used to eliminate them.

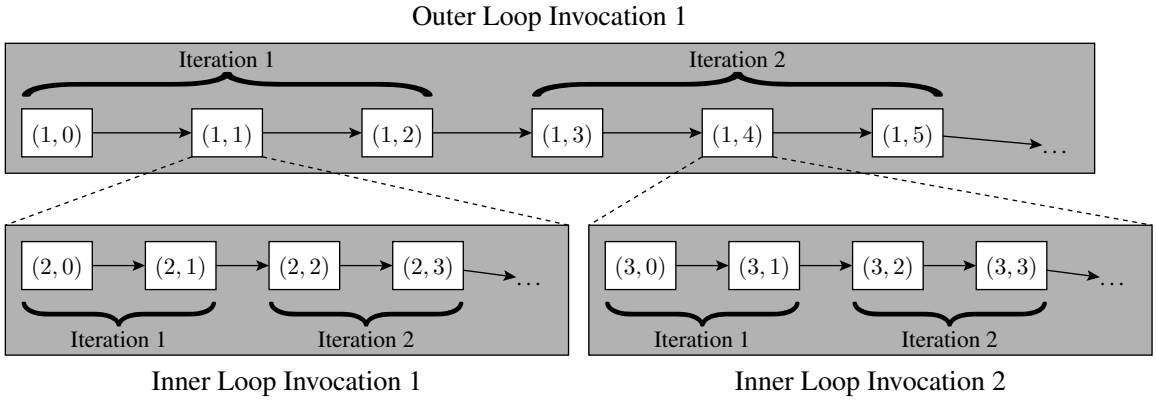


Figure 6.7: This figure illustrates the *version tree* that results from versioning the code in Figure 6.6. Each dotted box represents a memory version subspace (corresponding to some loop invocation). Each solid box represents a memory version. Each version is labeled with its identifier. The first number in the 2-tuple identifies the memory version subspace and the second identifies the specific version within the subspace. The figure further illustrates where inner loop version subspaces are embedded within the outer loop version space.

operation in the inner loop has been allocated. Using this approach, when versioning the outer loop in Figure 6.6, the inner loop shown in the dashed box would be reduced to a single operation. According to the acyclic versioning algorithm described previously, the inner loop operation would be assigned the version number 1 because it is presumed to be anti-dependent on the load operation A. Similarly, the store operation I is assigned version number 2 because it is assumed to be anti-dependent on the inner loop.

Just as was described earlier, each dynamic instruction will execute in version $Ni + v$ where N is the number of versions assigned in a given loop, i is the current iteration number for that loop, and v is the static version number assigned to the instruction. However, for an inner loop, these version numbers are not absolute. Rather, they represent a subspace in their parent loop's version space[2]. Figure 6.7 illustrates the resulting *version tree*. In the figure, each shaded box represents one loop invocation. Each loop invocation is allocated a unique number, and the versions within the invocation are identified with two-dimensional version numbers. The first number in the pair is the number assigned to the loop invocation

[2]Note, as an optimization, if an inner loop does not contain any backwards false memory dependences (either intra-iteration or loop-carried), then the inner loop does not need to be versioned independently of its parent and therefore does not need a separate version space.

and the second number is the dynamic version number ($Ni + v$). The version number assigned to the inner loop in the parent loop's versioning identifies where the inner loop version space is embedded in the parent loop's version space. The figure illustrates the inner loop being invoked twice in the outer loop. Each invocation of the inner loop is in its own version space embedded into versions $(1, 1)$ and $(1, 4)$ of the outer loop's version space.

Figure 6.8 illustrates the code that would be generated to realize this dynamic versioning for thread 3 from the running example. The `allocate` operations (J and P) return unique numbers to identify a version subspace and the operands to `allocate` indicate where the subspace should be embedded in the parent space [3] For the outer loop, the parent space is specified as $(0, 0)$ indicating the committed memory state. The variables `x` and `y` track the identifier for the outer and inner loop version spaces, respectively. The variables `i` and `i2` track the current iteration number for the outer loop (`i2` is just used to store the next iteration number to avoid introducing code along the loop back edge). Similarly, `j` and `j2` track the current iteration number for the inner loop. Before each load and store operation, the code generator has inserted a `enter` instruction to enter the appropriate memory version. Notice that all operations in the inner loop specify a version relative to the memory subspace identified by `y` (indicated by using `y` as the first operand to `enter`) and, in the outer loop, `x` is used instead. Further, the second operand to each `enter` operation is the dynamic version number given by $Ni + v$.

The observant reader will notice that several of the `enter` operations are redundant. For example, since operation N dominates operation operation O, and there are no redefinitions of `x` or `i` between N and O, operation O is unnecessary and can be removed. Similarly, operation V is redundant with operation U. A SpecDSWP compiler would eliminate these redundancies using variants of any classical redundancy elimination optimization (e.g., common subexpression elimination, partial redundancy elimination [52], global

[3] In reality, the `allocate` operations would exist in thread 1 and would be communicated to threads 2 and 3. However, they are shown in thread 3 for illustrative purposes.

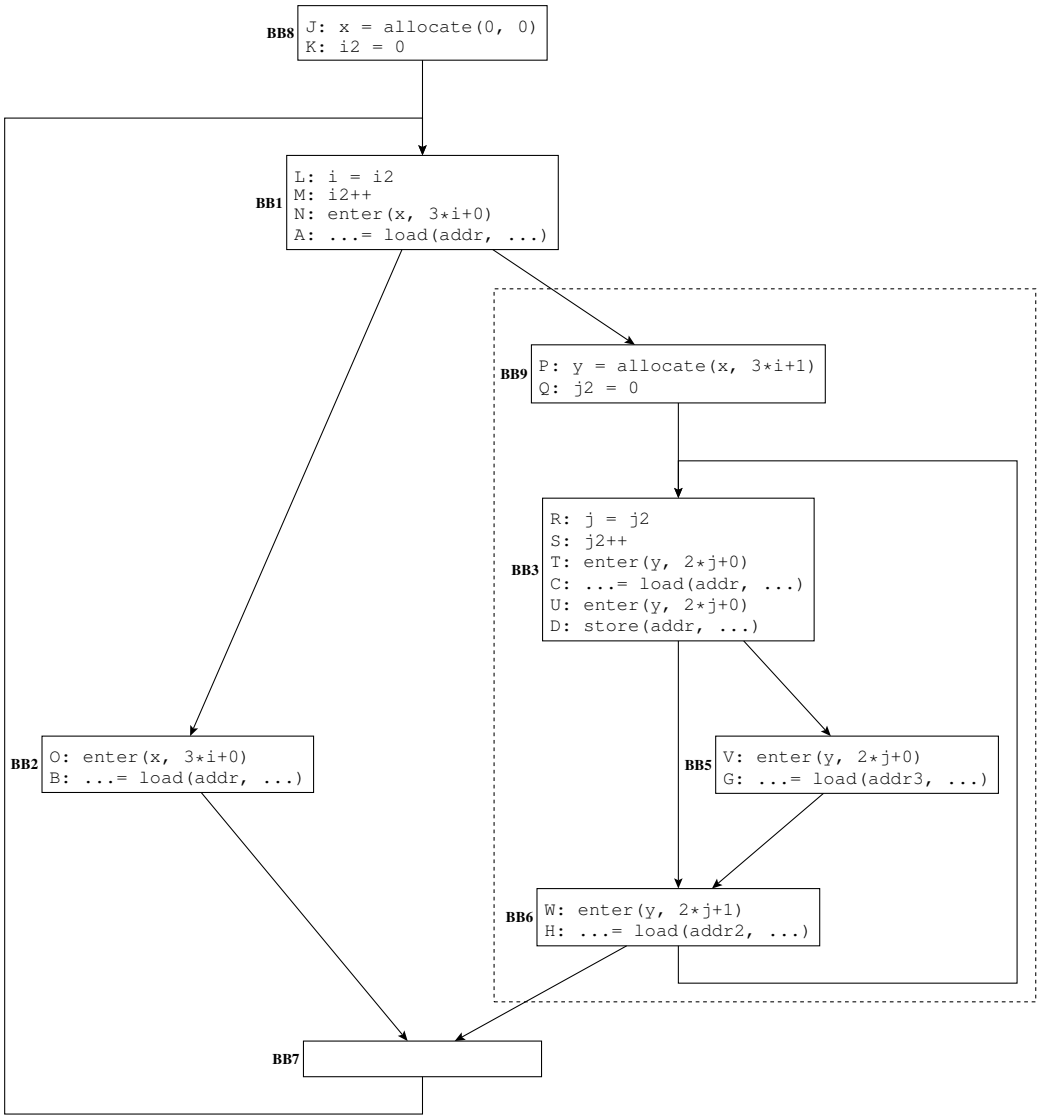


Figure 6.8: This figure illustrates the versioned code that would be generated from Figure 6.6 for thread 3. The code includes `enter` operations before each `load` and `store` to enter the proper memory version. Additionally, it includes `allocate` operations to allocate the necessary version subspaces creating the version tree shown in Figure 6.7.

value numbering [9]). Each `enter` operation would be treated as defining a single free temporary (essentially representing the runtime state that remembers the current version number) as a function of its two operands. Whenever the classic redundancy elimination optimization would eliminate redundant computation and insert a move, the `enter` eliminating variant would just delete the `enter` operation (without inserting a compensating move operation).

6.2.4 Handling Functions and Recursion

Versioning code with function calls can be handled similarly to code with inner loops. First, the call graph is built and function bodies are versioned from the leaves to the root. Two strategies exist to version a function call within an acyclic region. First, a version subspace could be allocated for the callee (akin to the subspace that was allocated to inner loops). This approach is necessary for calls to self-recursive functions or functions that are part of mutually recursive groups. Alternatively, since the call graph is processed bottom-up, when versioning the call site, the callee has already been versioned. Consequently, the version numbers allocated to operations in the callee can be used as offsets from the version number allocated to the call site. Using this strategy, the dynamic version number for the call site would be passed as an argument to the callee, and the callee would add the the version numbers assigned to its operations to this base to generate final dynamic version number1s at runtime. Finally, when versioning operations in the caller dependent on the call operation, the version number of the call site will be assumed to be the *last* version used by the callee.

6.3 Misspeculation Recovery

Section 6.1 described how misspeculation is detected. This section describes what action is taken to recover from misspeculation. Recall from Section 4.1 that SpecDSWP

```
1  while (true) {
2    produce_reg_chkpt(commit_thread);
3    status = loop_iteration();
4    produce(commit_thread, status);
5
6    if (status == EXIT)
7      break;
8    else if (status == MISSPEC)
9      wait_for_resteer();
10   else if (status == OK)
11     continue;
12
13 recovery:
14   produce_resteer_ack(commit_thread);
15   flush_queues();
16   regs =
17     consume_reg_chkpt(commit_thread);
18   restore_regs(regs);
19 }
```

(a) Worker Thread

```
1  do {
2    regs = consume_reg_chkpts(threads);
3    status = poll_worker_status(threads);
4
5    if (status == MISSPEC) {
6      resteer_threads(threads);
7      consume_resteer_acks(threads);
8      rollback_memory();
9
10     regs = full_loop_iteration(regs);
11     produce_reg_chkpts(threads, regs);
12   } else if (status == OK ||
13              status == EXIT) {
14     commit_memory();
15   }
16 } while (status != EXIT);
```

(b) Commit Thread

Figure 6.9: Pseudo-code for (a) a SpecDSWP worker thread and (b) the SpecDSWP commit thread.

handles misspeculation at the iteration level. When thread j in a pipeline of T threads detects misspeculation, several actions must be taken to recover. In this discussion, assume that, when misspeculation is detected, thread j is executing iteration n_j. These are the actions that need to be performed:

1. The first step in recovery is waiting for all threads to complete iteration $n_j - 1$.
2. Second, speculative state must be discarded and non-speculative state must be restored. This includes reverting the effects of speculative stores to memory, speculative writes to registers, as well as speculative produces to cross-thread communication queues. Since values produced to communication queues in a particular iteration are consumed in the same iteration, it is safe to flush the communication queues (i.e., after all threads have reached iteration n_j, there are no non-speculative values in the queues).
3. Third, the misspeculated iteration must be re-executed. Since re-executing the speculative code may again result in misspeculation, a non-speculative version of the iteration must be executed.
4. Finally, speculative execution can recommence from iteration $n_j + 1$.

To orchestrate the recovery process, SpecDSWP relies on an additional *commit thread*, which receives state checkpoints and status messages from each of the worker threads. Pseudo-code for a worker thread and the commit thread are shown in Figure 6.9. The next few sections describe the co-ordination between the worker threads and the commit thread.

6.3.1 Saving Register State

Each worker thread starts by collecting the set of live registers that need to be checkpointed and sending them to the commit thread. The commit thread receives these registers and locally buffers them. Since recovery occurs at iteration boundaries, each thread need only checkpoint those registers which are live into its loop header, since these are the only registers necessary to run the loop and code following the loop. In addition to high-level compiler temporaries (virtual registers), it is necessary to checkpoint machine-specific registers which may be live into the loop header such as the stack pointer, global pointer, or register window position.

Callee-saved (preserved) registers must be handled specially. While the function containing the loop being parallelized may not use or modify (and thus not spill) a particular callee-saved register, misspeculation may cause the register's value to change. For example, assume the function `A` contains the loop being parallelized, and the loop contains a call to function `B`. Since misspeculation recovery is initiated with an asynchronous resteer, the thread containing the call to `B` may be redirected to recovery code from the middle of the body of function `B`. Since `B` could have modified a callee-saved register and was not given a chance to restore it, even after recovery the register's value could be lost. To remedy this, if the compiler cannot prove a callee-saved register will not be changed, it must be checkpointed. However, it is only necessary to checkpoint and restore callee-saved registers once at the entry and exit, respectively, of the function containing the SpecDSWPed loop. Consequently, if a function contains more than one SpecDSWPed loop, the cost of checkpointing these registers can be amortized over all the loops.

The set of registers to be checkpointed can be optimized by recognizing that registers that are live into the loop entry, but that are not modified in the loop (loop invariants in the speculative code), need not be checkpointed each iteration, since their values are constant for the loop execution. Instead, these registers can be checkpointed once per loop invocation. While these loop invariant registers can be checkpointed once per invocation, upon misspeculation, they *must* be recovered since, due to misspeculation, unexpected code may have executed and modified their values.

6.3.2 Saving Memory State

To support rolling back of speculative memory updates, Speculative DSWP relies on runtime system support. Section 6.2 described how SpecDSWP encapsulates the memory operations performed by the loop being parallelized in memory versions. In addition to using these versions to break backwards, false memory dependences, SpecDSWP relies on being able discard a given memory version (rollback) or to merge the version into architectural state (commit). Chapter 7 describes *multi-threaded transactions* which support this behavior.

6.3.3 Initiating Recovery

After checkpointing architectural state, each worker thread executes the portion of the original loop iteration allocated to it. Execution of the loop iteration generates a status: the loop iteration completed normally, misspeculation was detected, or the loop iteration exited the loop. The worker thread sends this status to the commit thread, and then, based on the status, either continues speculative execution, waits to be redirected to recovery code, or exits the loop normally.

The commit thread collects the status messages sent by each thread and takes appropriate action. If all worker threads successfully completed an iteration, then all the memory versions corresponding to the current iteration are committed to architectural state, and the

register checkpoint is discarded.[4] If any thread detected misspeculation, the commit thread initiates recovery.

By collecting a status message from all worker threads each iteration, it is guaranteed that no worker thread is in an earlier iteration than the commit thread. Consequently, when recovery is initiated, step 1 is already complete. Recovery thus begins by asynchronously resteering all worker threads to thread-local recovery code. Once each thread acknowledges that it has been resteered, step 2 begins. The resteer acknowledgment prevents worker threads from speculatively modifying memory or producing values once memory state has been recovered or queues flushed. To recover state, the commit thread discards all writes to memory versions corresponding to the current or later loop iterations. Additionally, the worker threads flush all queues used by the thread.

6.3.4 Iteration Re-execution

Lastly, the misspeculated iteration is re-executed. The commit thread uses the register checkpoint to execute the original single-threaded loop body. The register values *after* the iteration has been re-executed will then be distributed back to the worker threads so that speculative execution can recommence. Note, while the commit thread is re-executing the misspeculated iteration, the worker threads can concurrently execute some of their recovery code. In particular, our experiments have shown that the time to flush queues and restore certain microarchitectural registers (whose values are unaffected by iteration re-execution) can be significant. Consequently, overlapping this recovery with re-execution can considerably reduce the misspeculation penalty.

Note that the commit thread incurs overhead that scales with the number of worker threads. While this code is very light weight, with small loops or many worker threads it is possible that one iteration in the commit thread takes longer than any worker thread.

[4] Since the register checkpoint is saved in virtual registers in the commit thread, no explicit action is required to discard the register checkpoint.

Various solutions to this problem exist. First the commit thread code is parallelizable. Additional threads can be used to reduce the latency of committing a loop iteration. Second, the problem can be mitigated by unrolling the original loop. This effectively increases the amount of time the commit thread has to complete its bookkeeping. This can potentially increase the misspeculation penalty, but provided misspeculation is infrequent, the tradeoff should favor additional worker threads.

Chapter 7

Multi-Threaded Transactions

Chapter 6 described how Speculative DSWP relies on memory versioning to enable memory speculation *and* to support memory rollback in case of misspeculation. In addition to SpecDSWP, other speculative automatic parallelization approaches, such as TLS, rely on memory versioning to enable parallelization. Even on the manual parallelization front, memory versioning has begun to gain popularity in the form of transactional programming models [33], a promising approach to mitigating the correctness issues associated with fine-grained locks and the performance issues associated with coarse-grained locks. All three approaches rely on *atomicity* to ease the burden of parallelization.

Atomicity eases the burden of manual parallelization by allowing developers to simply mark regions of code as atomic, rather than designing complicated locking protocols to guarantee mutual exclusion and proper synchronization. Regions marked atomic execute as if no other threads were concurrently executing. Run-time systems can guarantee atomicity by logically checkpointing state at the beginning of an atomic region and optimistically executing it concurrently with other threads while checking for access conflicts. In the event of a conflict, the checkpointed state is restored, and the atomic region is re-executed.

Atomicity eases the burden of automatic parallelization by allowing the compiler to *speculatively* execute potentially dependent regions of single-threaded code concurrently.

The compiler marks such regions atomic, and, if a speculated dependence manifests, the runtime system can rollback any state updates and re-execute the regions sequentially. This frees the compiler from relying on heroic analysis to prove independence between code regions or from being restricted because a dependence could occur even though dynamically such a dependence is rare.

Most transactional programming systems rely on transactional memories [37] and speculative parallelization techniques rely on TLS memory systems [30] to guarantee atomicity. Unfortunately, all proposed hardware *and* software implementations of transactional and TLS memories only provide atomicity of single-threaded regions. We refer to this as *single-threaded atomicity*. Supporting only single-threaded atomicity presents two key problems that we collectively refer to as the *single-threaded atomicity problem*.

First, providing only single-threaded atomicity precludes combining *nested parallelism* [5] with transactional programming. With nested parallelism, threads in a parallel section of code spawn more threads to exploit parallelism to complete a subtask. However, if the thread is executing within a transaction, single-threaded atomicity precludes the spawned threads from executing in the spawning thread's transaction. This limits the amount of parallelism that can be expressed in transactional code. Intel has recognized the importance of nested parallelism for non-transactional programs, and they fully support it in version 2.1 of their Threading Building Blocks library [3]. For transactional programs, applications not leveraging nested parallelism may be able to saturate today's multi-core processors, but future generation processors will offer many more cores. Saturating these processors will almost certainly require explicit support for nested parallelism in transactional code to allow both automatic and manual parallelization of individual transactions in today's transactional programs.

For automatic parallelization, the presence of only single-threaded atomicity forces the compiler to use atomic regions that do not span threads. Just as for manual parallelization, this forces the compiler to parallelize at one program level since the lack of nested

parallelism precludes simultaneously parallelizing an outer and inner loop.

Second, and perhaps more important, for both automatic and manual parallelization the lack of nested parallelism breaks modularity; the *implementation* of a function is exposed to the user (or compiler). For example, a single-threaded implementation of a library routine can be invoked from within a transaction, but a parallel one cannot. For both a developer and a compiler, this proves problematic since the implementation of many functions are unavailable during development or compilation. More distressing, for languages that support polymorphism (e.g., object-oriented languages) or function pointers, a single call site may invoke many different implementations, some single-threaded others potentially multi-threaded. Consequently, while parallelized code may work during testing, at runtime unanticipated code may be invoked leading to many subtle, hard to debug errors. In practice, the absence of nested parallelism will significantly inhibit the adoption of transactions, both in transactional programming and automatic parallelization.

To address the single-threaded atomicity problem, this chapter introduces the concept of *multi-threaded transactions* (MTXs). Like distributed transactions from the database world, an MTX represents an atomic set of memory accesses where these accesses may originate from many threads. Like its single-threaded counterpart, all the stores in an MTX will be merged with architectural state, or they will all be rolled back. Depending on implementation, either the runtime system or the software client is responsible for detecting conflicts between concurrently executing MTXs. However, unlike its single-threaded counterpart, MTXs support *multi-threaded atomicity*. Many threads can concurrently access a single MTX, yet these accesses will *all* commit atomically or *all* rollback.

In summary, this chapter will:

1. Illustrate how *single-threaded atomicity* is a crucial impediment to modularity in transactional programming *and* efficient speculation in automatic parallelization (Section 7.1).
2. Introduce multi-threaded transactions, a generalization of single-threaded transac-

```
1  atomic {
2    int *results =
3      get_results(&n);
4    sort(results, n);
5    for (i = 0; i < 10; i++)
6      sum += results[i];
7  }
```

(a) Application code.

```
1  void sort(int *list, int n) {
2    if (n == 1) return;
3    atomic {
4      sort(list, n/2);
5      sort(list + n/2, n - n/2);
6      merge(list, n/2, n - n/2);
7    }
8  }
```

(b) Sequential library implementation.

```
1  void sort(int *list, int n) {
2    if (n == 1) return;
3    atomic {
4      tid = spawn(sort, list, n/2);
5      sort(list + n/2, n - n/2);
6      wait(tid);
7      merge(list, n/2, n - n/2);
8    }
9  }
```

(c) Parallel library implementation.

Figure 7.1: Transactional nested parallelism example.

tions that supports *multi-threaded atomicity* (Section 7.2).

3. Propose an implementation of multi-threaded transactions based on an invalidation-based cache coherence protocol (Sections 7.3 and 7.4).

7.1 The Single-Threaded Atomicity Problem

This section explores the single-threaded atomicity problem, first with a transactional programming example, then with an automatic parallelization example. Both examples illustrate how the lack of nested parallelism and modularity preclude parallelization opportunities. The section concludes by describing two necessary properties to enable multi-threaded atomicity.

7.1.1 Transactional Programming

Consider the code shown in Figure 7.1(a). This code gathers a set of results, sorts the results, and then accumulates the first ten values in the sorted set. The code is executing within an atomic block, so the underlying runtime system will initiate a transaction at the

beginning of the block and attempt to commit it at the end of the block. Figures 7.1(b)-(c) show two possible implementations of the `sort` routine. Both sorts partition the list into two pieces and recursively sort each piece. The two sorted pieces are then merged to produce a fully sorted list. The first sort implementation is sequential and is compatible with the code executing in the atomic block. The atomic block contained inside the `sort` function creates a nested transaction, but *not* nested parallelism. The second sort implementation is parallel and delegates one of the two recursive sorts to another thread. Since nested parallelism is unsupported by proposed transactional memory (TM) systems, the parallel sort will not run correctly.

Problems first arise at the call to `spawn`. Since current TM proposals only provide single-threaded atomicity, the spawned thread necessarily does not run in the same transaction as the spawning thread. Consequently, the newly spawned thread cannot read the list it is supposed to sort since the data is still being buffered in the uncommitted spawning thread. Even if the data were available, the problem resurfaces in the call to `merge`. The `merge` function must be able to read the results of stores executed in the spawned thread. Unfortunately, those stores are not executed in the transaction containing the call to `merge`. Transaction isolation ensures that these stores are *not* visible.

Avoiding these problems means that the spawning thread has to commit its transaction before spawning the recursive call to `sort` and before the call to `merge`. Similarly, the spawned thread must commit its transaction before returning. However, if a transaction in another part of the system conflicts with the code executing in the atomic block in Figure 7.1(a), the modifications made within the block must be able to roll back. Unfortunately, if any of the commits just described occur, such a roll back is impossible. Consequently, current TM systems cannot support this nested parallelism.

As was discussed earlier, forbidding nested parallelism both limits parallelism and breaks modularity by exposing an interface's implementation. If the code in Figure 7.1(a) is object-oriented, then `get_results` can return one of many implementations of a `List`

interface. If `sort` is a virtual function in the `List` interface, the developer writing the atomic region does not know if the `sort` function is implemented with sequential or parallel code. While such information could be added to the function signature, to maintain composability, all parallel functions would also require a sequential counterpart. As modularity and code reuse are the cornerstone of software engineering, the single-threaded atomicity problem will significantly inhibit the use of transactional programming in large systems.

7.1.2 Automatic Parallelization

This section illustrates how the single-threaded atomicity problem can inhibit automatic parallelization, specifically for SpecDSWP.

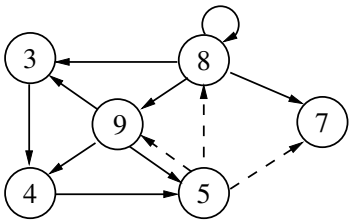
Figure 7.2(a) shows pseudo-code for a loop amenable to the SpecDSWP transformation. The loop traverses a linked list, extracts data from each node, computes a cost for each node based on the data, and then updates each node. If the cost for a particular node exceeds a threshold, or the end of the list is reached, the loop terminates.

Figure 7.2(b) shows the dependence graph among the various statements in each loop iteration (statements 1, and 6 are omitted since they are not part of the loop). Easily speculated dependences are shown as dashed edges in the figure. If no dependences are speculated, statements 3, 4, 5, 8, and 9 participate in a single recurrence. Consequently, non-speculative DSWP would be forced to put all five statements into a single thread. While the call to `update` (statement 7) could be placed into a separate thread, assuming the bulk of the runtime is spent in `extract`, `calc`, and in cache misses traversing the linked list (statement 8), such a partition would yield very little parallelism.

The loop can be effectively parallelized with SpecDSWP. Assuming `cost` will rarely exceed the threshold, the control dependence between the early exit (statement 4) and the subsequent statements can be speculated. After speculation, each statement will be in its own trivial SCC allowing SpecDSWP to schedule each statement into its own thread.

```
1  if (!node) goto exit;
2  loop:
3    data = extract(node)
4    cost = calc(data);
5    if (cost > THRESH)
6      goto exit;
7    update(node);
8    node = node->next;
9    if (node) goto loop;
10 exit:
```

(a) Single-Threaded Code



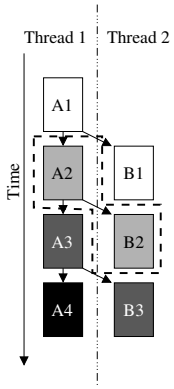
(b) PDG

```
1  if (!node) goto exit;
2  loop:
3    data = extract(node);
4    produce(T2, data);
5    update(node);
6    node = node->next;
7    produce(T2, node);
8    if (node) {
9      produce(CT, OK);
10     goto loop;
11   }
12 exit:
13 produce(CT, EXIT);
```

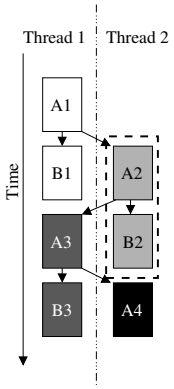
(c) Parallelized Code Thread 1

```
1  loop:
2    data = consume(T1);
3    cost = calc(data);
4    if (cost > THRESH)
5      produce(CT, MISSPEC);
6    node = consume(T1);
7    if (node) {
8      produce(CT, OK);
9      goto loop;
10   }
11 exit:
12 produce(CT, EXIT);
```

(d) Parallelized Code Thread 2



(e) SpecDSWP Schedule



(f) TLS Schedule

Figure 7.2: This figure illustrates the single-threaded atomicity problem for SpecDSWP. Figures (a)–(b) show a loop amenable to SpecDSWP and its corresponding PDG. Dashed edges in the PDG are speculated by SpecDSWP. Figures (c)–(d) illustrate the multi-threaded code generated by SpecDSWP. Finally, Figures (e)–(f) illustrate the necessary commit atomicity for this code if it were parallelized using SpecDSWP and TLS respectively. Stores executed in boxes with the same color must be committed atomically.

Figures 7.2(c) and (d) show the parallel code that results from applying SpecDSWP targeting two threads (misspeculation recovery code has been omitted for clarity). In the figure, statements 3, 7, 8, and 9 (using statement numbers from Figure 7.2(a)) are in the first thread, and statements 4 and 5 are in the second thread. Statements in bold have been added for communication, synchronization, and misspeculation detection. Figure 7.2(e) shows how the parallelized code would run assuming no misspeculation.

As Chapter 6 described, SpecDSWP relies on runtime support to rollback the effects of speculative stores in the event of misspeculation. Unfortunately existing transactional and TLS memory systems are inadequate for SpecDSWP. To understand why, first consider how TLS buffers speculative state. Figure 7.2(f) shows a potential execution schedule for a two-thread TLS parallelization of the program from Figure 7.2(a) assuming the code has been scheduled such that the regions labeled A and B correspond to the code assigned to the first and second threads, respectively, from the SpecDSWP parallelization. TLS executes each speculative loop iteration in a TLS epoch. In the figure, blocks participating in the same epoch are shaded in the same color. One particular epoch is outlined for illustration. TLS epochs resemble transactions except TLS epochs have a predetermined commit order, while transactions compete for commit order. Epochs are assigned according to the single-threaded program order and define the logical order for memory instructions. Stores executed during each iteration are buffered in the epoch, and when a particular epoch becomes the oldest in the system, it is allowed to commit. If an epoch conflicts with a later epoch, all later epochs roll back.

Figure 7.2(e) shows the corresponding execution schedule that would be used by SpecDSWP. The blocks contained in the outlined TLS epoch are similarly outlined in the figure. Since SpecDSWP also uses loop iterations as the unit of speculative work, it also must be able to rollback the effects of the outlined region in the event of a misspeculation. Notice, however, that the outlined region *spans* threads and potentially multiple cores. Consequently TLS epochs and conventional transactions are inapplicable because they only provide single-

threaded atomicity.

An alternate possibility is to wrap each iteration from each thread in a distinct epoch. In Figure 7.2(e), each block would be in a distinct epoch. Epochs 1 and 2 would be assigned to the first iteration in threads 1 and 2 respectively. Since epoch 1 is the oldest epoch, the stores from the first iteration in thread 1 would be treated as non-speculative. Unfortunately, the stores in thread 1's first iteration are indeed speculative. If thread 2 detects misspeculation (line 4 in Figure 7.2(d)) in the first iteration, thread 1 needs to be rolled back to undo the stores in the call to `update`. Attempts to salvage the system by reassigning epoch numbers also do not work since epochs will not be ordered according to the single-threaded program order.

The problem could be solved if four epochs, rather than two, were used per iteration. The first thread would execute statements 2 and 3 in epoch 1 and statements 4–9 in epoch 3. The second thread would execute statements 1–4 in epoch 2 and statement 5 in epoch 4. This assignment of epoch numbers matches single-threaded program order thus guaranteeing correctness (it also matches the assignment of version numbers described in Section 6.2). However, the number of epochs needed per loop iteration scales with the degree of speculation. In general, even with modest speculation, many epochs will be necessary. While such a system works in theory, in practice it is untenable since it requires that the state of all live registers and the contents of the inter-thread communication queues in addition to the state of memory be checkpointed at the beginning of each epoch. With many epochs per iteration, the overhead of such aggressive checkpointing can easily overtake the benefits parallelization. Even if the register and queue checkpointing overhead were not significant, traditional epochs would still not be a complete solution since epoch numbers are single-dimensional (not multi-dimensional as was discussed in Section 6.2.3) and consequently do not support nested parallelism. Consequently, just as in transactional programming, the single-threaded atomicity problem significantly impairs automatic parallelization.

7.1.3 Supporting Multi-Threaded Atomicity

Given the problems just described, it is clear that systems which provide only single-threaded atomicity are insufficient. Here, we identify two key features which extend conventional transactional memories to support multi-threaded atomicity.

Group Transaction Commit The first fundamental problem faced by both examples was that transactions[1] were isolated to a single thread. This limitation could not be avoided by using one transaction for each thread participating in a logically atomic region since these transactions did not all commit atomically. Section 7.2 introduces the concept of an MTX that encapsulates many sub-transactions (subTX). Each subTX resembles a TLS epoch, but all the subTXs within an MTX can commit together providing *group transaction commit*.

Uncommitted Value Forwarding Group transaction commit alone is still insufficient to provide multi-threaded atomicity. It is also necessary for speculative stores executed in an early subTX to be visible in a later subTX (at least within the same MTX). While many TLS implementations support *uncommitted value forwarding* between epochs as an optimization [75], uncommitted value forwarding between subTXs is *necessary* to guarantee multi-threaded atomicity. Ensuring that uncommitted values are forwarded between threads allows the threads executing in a single MTX, but different subTXs, to execute cooperatively. Specifically, it allows threads participating in an MTX to synchronize, thus preventing misspeculation, even for *ambiguous* memory dependences.

In the nested parallelism example (Figure 7.1), the recursive call to `sort` must be able to see the results of uncommitted stores executed in the primary thread, and the call to `merge` must be able to see the results of stores executed in the recursive call to `sort`. Uncommitted value forwarding facilitates this store visibility. Similarly, in the SpecDSWP example (Figure 7.2), if each loop iteration executes within a single MTX, and each itera-

[1] For the remainder of the paper, we will refer to both transactions and epochs as transactions. A distinction will be made only where necessary.

Instruction	Arguments	Description
`allocate`	VID	Returns an unused MTX ID setting its parent to the specified VID.
`deallocate`	MTX ID	Indicate that the specified MTX ID is no longer being used.
`enter`	VID	Enter the specified MTX and subTX.
`commit.p1`	MTX ID	Commit phase 1. Mark the specified MTX as non-speculative and acquire the commit token from the parent version. Future conflicts must rollback the other conflicting transaction.
`commit.p2`	VID	Commit phase 2. All the stores from the current MTX in the specified subTX will be committed to architectural state *or* the parent version. SubTXs within a particular MTX *must* be committed in order. Threads must issue `enter` to enter a legitimate MTX or committed state.
`commit.p3`	MTX ID	Commit phase 3. Return the commit token to the parent version.
`rollback`	MTX ID	All the stores from the specified MTX will be discarded, and the MTX deallocated. Threads must issue `enter` to enter a legitimate MTX or committed state.

Table 7.1: Instructions for managing MTXs.

tion from each thread executes within a subTX of that MTX, uncommitted value forwarding is necessary to allow the stores from `extract` to be visible to loads in `calc`.

7.2 The Semantics of Multi-Threaded Transactions

MTXs provide the illusion of a private memory for the threads participating in the transaction. Conceptually, the private memory is initialized with the contents of architectural memory at the time the MTX is created. Loads are serviced from the private memory, and stores update the private memory without affecting other threads not participating in the MTX. If an MTX commits, then all threads can observe the effects of the stores executed in the transaction. The following sections will more formally define an MTX and introduce new instructions for managing them. For reference, the set of instructions added to the ISA are listed in Table 7.1.

7.2.1 Basics

An MTX provides atomicity and isolation for a collection of memory operations originating from one or more threads. Two MTXs are said to produce an *inter-transaction conflict* if one transaction writes a location that the other reads without first writing. Such conflicts

lead to non-serializable accesses, and one of the conflicting MTXs should be rolled back.

To allow programs to define a memory order *a priori*, MTXs are decomposed into subTXs. While separate MTXs compete for commit order, the commit order of subTXs within an MTX is predetermined just like TLS epochs (Section 7.2.2). A particular subTX within an MTX is identified by a pair of identifiers, $(\text{MTX ID}, \text{subTX ID})$, called the *version ID* (*VID*). Note that an MTX comprised of only one subTX implements the semantics of a conventional single-threaded transaction.

An MTX is created by the `allocate` instruction which returns a unique MTX ID. A discussion of `allocate`'s argument is deferred to Section 7.2.3. A thread enters an MTX by executing the `enter` instruction indicating the desired MTX ID and subTX ID. If the specified subTX does not exist, the system will automatically create it. However, the software is responsible for managing and assigning unique subTX IDs. A thread may leave a particular subTX and enter another (in the same or different MTX) by issuing a subsequent `enter` instruction. The VID $(0, 0)$ is reserved to represent committed architectural state. Consequently, a thread may leave all MTXs and resume issuing non-speculative memory operations by issuing `enter(0,0)`.

7.2.2 Intra-Transaction Memory Ordering

Within a subTX, memory operations are ordered in accordance with single-threaded program order. SubTXs within an MTX are well ordered (like TLS epochs), and this order defines the semantic order of memory operations across threads. Consequently, values stored in one subTX are visible in all subsequent subTXs. Stores to the same address from different subTXs are legal, and subsequent loads will observe the value stored by the nearest earlier subTX. This provides uncommitted value forwarding as was discussed in Section 7.1.3.

However, to allow memory alias speculation within an MTX (e.g., TLS applied to a single task in a transactional program, or non-loop carried memory speculation in SpecDSWP),

if an address is written in a particular subTX after the corresponding address has been read in a later subTX, then an *intra-transaction conflict* has occurred and the entire MTX should be rolled back.

To avoid intra-transaction conflicts, the threads participating in an MTX may synchronize to guarantee that a store in an earlier subTX has executed before a dependent load is issued in a later subTX. Conventional memory-based synchronization or out-of-band (non-memory) synchronization [60, 61] can be used. Since memory-based synchronization may itself trigger an intra-transaction conflict, the ISA must also be extended with synchronization primitives (e.g. compare-and-swap) that do not trigger such conflicts.

7.2.3 Nested Transactions

Many threads participating in a single MTX may be executing concurrently. However, a thread executing within an MTX may not be able to spawn additional threads and appropriately insert them into the memory ordering because no sufficiently sized gap in the subTX ID space has been allocated. As was discussed in Section 6.2, allocating a sufficiently large gap is not always possible due to nested parallelism.

To remedy this and to allow arbitrarily deep nesting, rather than decomposing subTXs into sub-subTXs, an MTX may have a parent subTX (in a different MTX). When such an MTX commits, rather than merging its speculative state with architectural state, its state is merged with its parent subTX. Consequently, rather than directly using a subTX, a thread may choose to allocate a new MTX specifying its subTX as the parent. The thread may then spawn more threads providing them with subTXs within the newly created MTX. An MTX's parent subTX is specified as the argument to the `allocate` instruction. The VID $(0, 0)$ can be used as the argument to specify that an MTX has no parent.

To provide uncommitted value forwarding, values are forwarded from parent subTXs to all descendants and vice-versa. To resolve memory ordering ambiguities between direct accesses to a subTX and accesses resulting from a nested MTX committing to its parent

subTX, direct accesses to a subTX that is the parent of another MTX are forbidden. Additionally, no inter-transaction conflict should be detected in response to a read from a child (parent) MTX receiving a value forwarded from any ancestor (descendant), even though the definition of inter-transaction conflict provided earlier would deem such accesses as conflicting. However, inter-transaction conflicts can still occur between peers (i.e., MTXs that share a common ancestor subTX). In all other respects, the child MTX behaves like any other independent MTX.

7.2.4 Commit and Rollback

An MTX commits to architectural state *or*, if it has a parent, to its parent subTX. The state modifications in an MTX are committed using a three-phase commit. Commit is initiated by executing the `commit.p1` instruction. This instruction marks the specified MTX as non-speculative and acquires the commit token from the parent subTX (or architectural state). After an MTX is marked non-speculative, if another MTX conflicts with this one, the *other* MTX must be rolled back. Additionally, only one commit token exists per subTX, so multiple MTXs cannot commit to architectural state or the same parent subTX simultaneously. If the commit token is unavailable when this instruction issues, the issuing thread blocks until the token becomes available. Next, to avoid forcing hardware to track the set of subTXs that exist in each MTX, software is responsible for committing each subTX within an MTX, but they *must* be committed in order (the results of out-of-order commit are undefined). This is accomplished with the `commit.p2` instruction. This instruction atomically commits all the stores for the subTX specified by the VID. Since the commit token for the parent subTX is acquired prior to executing `commit.p2` instructions, the commit of an MTX is guaranteed to be atomic. Finally, the commit token is returned to the parent subTX (or architectural state) by executing the `commit.p3` thus unblocking stalled committers. Finally, the MTX ID for the committing MTX is returned to the system by executing the `deallocate` instruction. Note, a partial MTX commit is possible if not all subTXs are

committed before executing `commit.p3`, or if new subTXs are entered after executing `commit.p3`. In both cases, the `deallocate` instruction must appear after the given MTX has fully committed.

Rollback is simpler than commit and involves only a single instruction. The `rollback` instruction discards all stores from *all* subTXs from the specified MTX and all of its descendants.

After either a commit or rollback, all threads that were still participating in a subTX that was committed or rolled back must execute an `enter` instruction prior to issuing memory operations. This ensures that the threads enter the committed state or another valid subTX prior to accessing memory.

7.2.5 Putting it Together

This section revisits the examples from Sections 7.1.1 and 7.1.2 demonstrating how MTXs can be used to support nested parallelism and modularity in transactional programming and speculative state buffering in SpecDSWP. The section begins with the SpecDSWP example since it requires fewer code modifications and then proceeds to the slightly more involved transactional programming example.

Speculative DSWP

Recall the Speculative DSWP example from Figure 7.2. Figure 7.3 reproduces the code from Figures 7.2(c) and 7.2(d) with MTX management instructions added in bold. The parallelized loop is enclosed in a single MTX, and each iteration uses two subTXs. Thread 1 starts in subTX 0 and then moves to subTX 2 to break a false memory dependence between `calc` and `update`. Thread 2 operates completely in subTX 1. Since MTXs support uncommitted value forwarding, the data stored by thread 1 in the `extract` function will be visible in thread 2 in the `calc` function. Further, this communication through memory will *never* cause a transaction conflict because the accesses are synchronized using

```
1  if (!node) goto exit;
2  mtx_id = allocate(0, 0);
3  produce(T2, mtx_id);
4  produce(CT, mtx_id);
5  iter = 0;
6  loop:
7    enter(mtx_id, 3*iter+0);
8    data = extract(node);
9    produce(T2, data);
10   enter(mtx_id, 3*iter+2);
11   update(node);
12   node = node->next;
13   produce(T2, node);
14   if (node) {
15     iter++;
16     produce(CT, OK);
17     goto loop;
18   }
19 exit:
20 produce(CT, EXIT);
21 enter(0,0);
```

(a) Parallelized Code Thread 1

```
1  mtx_id = consume(T1);
2  iter = 0;
3  loop:
4    enter(mtx_id, 3*iter+1);
5    data = consume(T1);
6    cost = calc(data);
7    if (cost > THRESH)
8      produce(CT, MISSPEC);
9    node = consume(T1);
10   if (node) {
11     iter++;
12     produce(CT, OK);
13     goto loop;
14   }
15 exit:
16 produce(CT, EXIT);
17 enter(0,0);
```

(b) Parallelized Code Thread 2

```
1  mtx_id = consume(T1);
2  iter = 0;
3  do {
4    ...
5    if (status == MISSPEC) {
6      ...
7      rollback(mtx_id);
8      ...
9      mtx_id = allocate(0, 0);
10     produce(T1, mtx_id);
11     produce(T2, mtx_id);
12     iter = 0;
13     ...
14   } else if (status == OK || status == EXIT) {
15     commit.p1(mtx_id);
16     commit.p2(mtx_id, 3*iter+0);
17     commit.p2(mtx_id, 3*iter+1);
18     commit.p2(mtx_id, 3*iter+2);
19     commit.p3(mtx_id);
20   }
21   iter++;
22 } while (status != EXIT);
23 deallocate(mtx_id);
```

(c) Commit Thread

Figure 7.3: Speculative DSWP example with MTXs.

produce/consume synchronization.

In the event of misspeculation, the commit thread rolls back the MTX (line 7 of Figure 7.3(c)) and allocates a new MTX. With memory state recovered, the recovery code can then re-execute the iteration non-speculatively. If no misspeculation is detected, the commit thread uses group commit semantics, and partial MTX commit, to atomically commit both subTXs comprising the the iteration (lines 15–19 of Figure 7.3(c)). Finally, after finishing the loop, threads 1 and 2 resume issuing non-speculative loads and stores by executing the `enter(0,0)` instruction, while the commit thread deallocates the MTX.

Transactional Programming

Figure 7.4 reproduces the transactional programming example from Figures 7.1(a) and 7.1(c) with code to manage the MTXs. As before, new instructions are shown in bold. In addition to the new code shown in bold, Figure 7.4(c) shows the implementation of a support library used for transactional programming. Finally, to help follow the discussion below, Figure 7.5 shows the MTXs that would be created by executing the above code assuming `build_results` returns a list of size 4.

To manage the nested parallelism in the recursive sort procedure, this example uses the ability to commit an MTX into a subTX of a different MTX. The atomic support library provides three functions: a function to enter a new atomic region, a function to leave an atomic region, and a function to advance to the next subTX within the current atomic region. Additionally, a thread-local global variable, `vid`, is used to track the current MTX and subTX for a given thread.

The application code (Figure 7.4(a)) begins by starting a new atomic region. In the support library, this causes the thread to enter a new MTX to ensure the code marked atomic in Figure 7.1(a) is executed atomically. To create the new MTX, the `begin_atomic` function first stores the current VID into a local variable. Then it executes an `allocate` instruction to obtain a fresh MTX ID, and sets the current subTX ID to 0. Finally, it

```
version_t parent =
  begin_atomic();
int *results = get_results(&n);
sort(results, n);
for (i = 0; i < 10; i++)
  sum += results[i];
end_atomic(parent);
```

(a) Application code.

```
void sort(int *list, int n) {
  if (n == 1) return;
  version_t parent =
    begin_atomic();
  thread = spawn(sort, list, n/2);
  sort(list + n/2, n - n/2);
  wait(thread);
  next_stx();
  merge(list, n/2, n - n/2);
  end_atomic(parent);
}
```

(b) Parallel library implementation.

```
typedef struct {
  int mtx_id;
  int s_id;
} version_t;

__thread version_t vid = {0, 0};

version_t begin_atomic() {
  version_t parent = vid;
  vid.mtx_id = allocate(parent.mtx_id, parent.s_id);
  vid.s_id = 0;
  enter(vid.mtx_id, vid.s_id++);
  return parent;
}

void end_atomic(version_t parent) {
  for(int i = 0; i < vid.s_id; i++)
    commit(vid.mtx_id, i);
  vid = parent;
  enter(vid.mtx_id, vid.s_id++);
}

void next_stx() {
  enter(vid.mtx_id, vid.s_id++);
}
```

(c) Atomic library implementation.

Figure 7.4: Transactional nested parallelism example with MTXs.

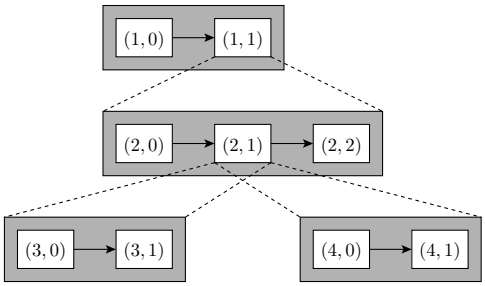


Figure 7.5: MTXs created executing the code from Figure 7.4.

enters the newly allocated MTX and advances the subTX pointer indicating that subTX 0 is being used. After returning from `begin_atomic`, the application code proceeds normally eventually spawning a new thread for the `sort` function.

When it starts, the spawned thread will be in the same subTX as the thread that spawned it. The `spawn` primitive is responsible for copying the thread-local global variable `vid` to the new thread allowing future calls to `begin_atomic` and `end_atomic` to function properly. Recall, with single-threaded transactions the spawned thread would not have been able to see the results of the list construction (line 3 in Figure 7.4(a)) because the spawning transaction was isolated from other transactions. With MTXs, however, since the spawned thread is in the same subTX as the spawning thread, the values are visible. The spawned `sort` function, after checking for the recursive base case, immediately allocates a new MTX whose parent is the version identified by the thread-local variable `vid`. Even after creating this new MTX, uncommitted value forwarding ensures that the list construction is visible to the spawned thread. Further, by creating a new MTX, the `sort` function ensures that even through many recursive calls, this particular call to `sort` will execute atomically *and* will not use many subTXs in the caller's MTX potentially violating the intended memory order. Additionally, the parent-child relationship between the caller's current subTX and the newly created MTX ensure that `sort`'s side-effects will be visible to the caller.

After the main thread recursively invokes `sort`, it waits for the spawned thread to complete sorting its portion of the list. The `sort` function proceeds to the next subTX since writes are not allowed in any subTX that is the parent of another MTX. The function then merges the results of the two recursive sort calls. Once again uncommitted value forwarding allows the primary thread to see the sorted results written by the spawned thread. Finally, `sort` completes by calling `end_atomic` which commits the current MTX into its parent subTX.

After the call to `sort` returns, the application code uses the sorted list to update `sum`.

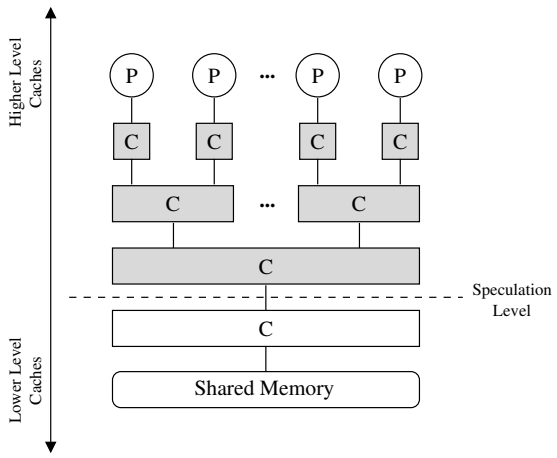


Figure 7.6: Cache architecture for MTXs.

After `sum` is updated, the application code commits the MTX (using `end_atomic`) it allocated. Since all the data generated throughout the execution is buffered in this MTX it will all commit atomically, or, in the event of a conflict, can all be rolled back.

7.3 Implementing Multi-Threaded Transactions

This section and the next section describe a complete implementation of MTXs. Like other transactional memory and TLS implementations [20, 33, 35, 37, 69, 75], the proposed implementation of MTXs buffers speculative state in the processor caches and relies on a modified cache coherence protocol to forward both speculative and non-speculative data, and to detect misspeculation. Figure 7.6 shows the general architecture of the system. The circles marked P are processors. Boxes marked C are caches. Shaded caches store speculative state (speculative caches), while unshaded caches store only committed state (non-speculative caches). Notice that both private and shared caches can store speculative state. The boundary between speculative and non-speculative caches is referred to as the *speculation level* [69]. To facilitate easier understanding, the full system will be incrementally described. This section describes an MTX system that does *not* perform conflict detection, and Section 7.4 describes how to support conflict detection.

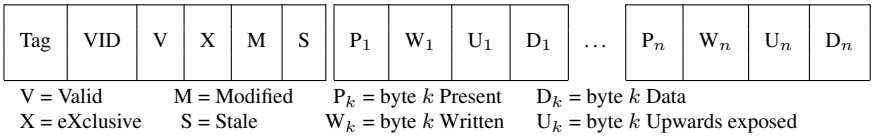


Figure 7.7: MTX cache block.

7.3.1 Cache Block Structure

To support MTXs, blocks in speculative caches are augmented with additional metadata. Figure 7.7 shows the data stored in each cache block. Like traditional coherent caches, each block stores a tag, status bits (V, X, and M) to indicate the coherence state, and actual data. The MTX cache block additionally stores the VID of the subTX to which the block belongs and a stale bit indicating whether later MTXs or subTXs have modified this block. Finally each block stores three bits per byte (P_k, W_k, and U_k) indicating whether the particular data byte is present in the cache (a sub-block valid bit), whether the particular byte has been written in this subTX, and whether the particular byte is upwards exposed. This status is maintained at the byte-level to avoid conflicts due to false sharing. The U_k bits will be discussed in Section 7.4.

In a traditional coherent cache, the status bits indicate what accesses are legal on a given block. The presence of the valid bit indicates that reads are legal, and the presence of both the valid and exclusive bits indicates that writes are legal. Similarly, in an MTX cache, the metadata dictates what accesses are legal. In addition to an address, all cache accesses carry a VID. A read access is a hit if:

- the block is valid ($V = 1$),
- the cache tags match,
- the particular bytes being read are present ($P_k = 1$), and
- either
  - the request VID is equal to the block VID, or
  - the request VID is greater than the block VID, and the block is not stale ($S = 0$).

The greater than check follows the partial order among VIDs. Two VIDs within the same MTX are comparable, and the order is defined by the subTX ID. VIDs from different MTXs are compared by traversing the parent pointers until parent subTXs in a common

Tag	VID	P_1	D_1	P_2	D_2	P_3	D_3	P_4	D_4
0x8001	(3, 7)	1	0xAB	0	-	1	0xBE	0	-
0x8001	(3, 8)	1	0xDE	0	-	0	-	0	-
0x8001	(3, 6)	0	-	1	0xAD	0	-	1	0xEF
0x8001	(3, 9)	1	0xDE	1	0xAD	1	0xBE	1	0xEF

Request Tag: 0x8001
Request VID: (3,9)
Merged Result: 0xDEADBEEF

Figure 7.8: Matching cache blocks merged to satisfy a request.

MTX are found. If the parent subTXs are identical, then the two original VIDs are *incomparable*. Otherwise, the order is once again defined by the subTX IDs. The greater than comparison can be efficiently performed for two VIDs in the same MTX. Section 7.3.4 describes additional support to facilitate comparing VIDs from different MTXs.

To allow multiple versions of the same variable to exist in different subTXs (within a single thread or across threads), caches can store multiple blocks with the same tag but different VIDs. Using the classification developed by Garzarán et al., this makes the system described here MultiT&MV (multiple speculative tasks, multiple versions of the same variable) [30]. This ability implies that an access can hit on multiple cache blocks (multiple ways in the same set). If that occurs, then data should be read from the block with the greatest VID. To satisfy the read request, it may be necessary to rely on *version combining logic* (VCL) [75] to merge data from multiple ways. Figure 7.8 illustrates how three matching blocks would be combined to satisfy a read request. Note that this definition of cache hit implicitly forwards speculative data between subTXs.

On the other hand, a write access (or a read exclusive access) is a hit if the block is valid *and* exclusive, the cache tags match, and the VIDs are *equal*. Note that even if the P bits are unset, the write is still a hit, and, in fact, the write will assert the corresponding P bits. Additionally, the write will set the W bits for written bytes and the M bit for the block.

7.3.2 Handling Cache Misses

In the event of a cache miss, a cache contacts its lower level cache to satisfy the request. A read miss will issue a read request to the lower level cache, while a write miss will

	$VID_{request} < VID_{block}$	$VID_{request} = VID_{block}$	$VID_{request} > VID_{block}$
Read	*No response*	If block is modified, write back and clear M. Clear X.	
Read Excl.	Set $P_k = W_k$ for all bytes in the block.	If block is modified, write back and clear M. Clear V.	If block is modified, write back and clear M. Clear X and set S.

Figure 7.9: Cache response to a snooped request.

issue a read-exclusive request (or an upgrade request if the block is already present in the requesting cache).

Peer caches to the requesting cache must snoop the request (or a centralized directory must forward the request to sharers of the block) and take appropriate action. Figure 7.9 describes the action taken in response to a snooped request. Note that if the request VID and block VID are incomparable (i.e., from different unordered MTXs), no action is necessary.

The column where $VID_{request} = VID_{block}$ describes the typical actions used by an invalidation protocol. Both read and read exclusive requests force other caches to write back data. Read requests also force other caches to relinquish exclusive access, whereas read exclusive requests force block invalidation. The other two columns in the table describe actions unique to an MTX cache.

First, consider the case where $VID_{request} < VID_{block}$. Here, the snooping cache does not need to take action in response to a read request since the request thread is operating in an earlier subTX. As per the semantics of MTXs, this means data stored in the block should not be observable to the requester. For a read exclusive request, however, action must be taken. The read exclusive request indicates that an earlier subTX may write to the block. Since such writes should be visible to threads operating in the block's subTX, the snooping cache must invalidate its block to ensure subsequent reads get the latest written values. Instead of invalidating the entire block, the protocol invalidates only those bytes that have not been written in the block's subTX. This is achieved simply by copying each W_k bit into its corresponding P_k bit. After such a partial invalidation, reads that access data written in the same subTX still hit in the cache.

Next, consider the case where $VID_{request} > VID_{block}$. Here the snooping cache may

have data needed by the requester since MTX semantics require speculative data to be forwarded from early subTXs to later subTXs. Consequently, the snooping cache takes two actions. First, it writes back any modified data from the cache since it may be the latest data (in subTX order) that has been written to the address. Next, it relinquishes exclusive access to ensure that prior to any subsequent write to the block, other caches have the opportunity to invalidate their corresponding blocks. Similar action is taken in response to a read exclusive request. Data is written back and exclusive access is relinquished. Additionally, the snooping cache marks its block stale (by setting the S bit), ensuring that accesses made from later subTXs are not serviced by this block (recall that if $\text{VID}_{\text{request}} > \text{VID}_{\text{block}}$, a read is only hit if the block is not marked stale).

For correct operation, even the requesting cache must "snoop" in response to its own requests. This is necessary since the cache may contain blocks relevant to the request, but that did not cause the access to hit because the blocks were stale, or the request was a write and the VIDs did not match exactly.

The requesting cache can assemble the complete response to its request by using the VCL on all blocks written back and the response from the lower level cache. The assembled cache block can be inserted into the requesting cache using the largest VID of all blocks fed into the VCL. Since all bytes will be returned in response to the request, all its P bits should be asserted. Finally, the stale bit can be copied from the returned block with the largest VID. Similarly, the M and W bits can be set based on the corresponding bits from a returned block where $\text{VID}_{\text{request}} = \text{VID}_{\text{block}}$. If no such block is returned, the M and W bits can be cleared.

7.3.3 Crossing the Speculation Level

The implementation discussed thus far has implicitly assumed that each cache's lower level cache is also speculative. Obviously, this is untrue for the speculative cache right above the speculation level. We will refer to these caches as *speculative boundary caches*. These

caches must react slightly differently than other speculative caches since:

1. They cannot write back speculative data to their lower level cache, and
2. They are responsible for allocating cache blocks for new VIDs.

To handle the inability to write back speculative data to a lower level cache, two modifications are necessary. First, any eviction of a modified speculative block must cause the corresponding MTX to be rolled back. This is identical to how many other transactional proposals handle overflow of speculative state [33, 37]. Note, stalling the subTX that has overflowed is not permissible since that may cause the corresponding MTX to deadlock. Virtualizing MTXs is possible but is deferred to future work.

Second, write backs of speculative state necessitated by the coherence protocol must be handled specially. The cache must still respond to the request, forwarding the speculative data it possesses. This resembles a write back, but differs in that if $\text{VID}_{\text{request}} > \text{VID}_{\text{block}}$, the cache should not clear its modified (M) bit for the block. This means a cache block can potentially be in the modified state, but not exclusive. As Section 7.3.5 will describe, such a state indicates that when the subTX commits, the cache must acquire exclusive access for the block before it can merge the data into committed state or another subTX. This approach does not introduce any new data merging problems since two peer caches will never have the same line (in the same version) in the modified state concurrently. Consequently, the data is always consistent, and at commit, at any particular level of the cache hierarchy the cache with the line in the modified state will cause the other caches to invalidate their copy.

Speculative boundary caches are also responsible for allocating blocks with new VIDs. This occurs because all read exclusive requests for a newly allocated subTX will necessarily miss in all higher level caches. The request will *also* miss in the speculative boundary cache. The cache should proceed normally, propagating the request to the next memory level. This provides peer caches with an opportunity to snoop the request and supply any relevant data. The lower level, non-speculative cache will always respond with committed state. Using these responses, the speculative boundary cache can assemble a block with

the latest available data. Rather than returning this block as a normal speculative cache would, it should replace the VID in the response with the request VID, thereby allocating the address in the requested subTX. To ensure correctness, the newly allocated line must be marked stale if the line has previously been allocated with a larger VID.

7.3.4 Nested Transactions

As mentioned earlier, for each cache access, it is necessary for a cache to compare the request VID to the VID stored in a potentially matching cache block. Comparing two VIDs within the same MTX is simply a straight-forward comparison of the subTX IDs. However, comparing two VIDs from different MTXs is more complicated due to nested MTXs.

In order to compare two VIDs from different MTXs, it is necessary to track the transaction hierarchy. A special region of memory is allocated to store the VID of the parent of each MTX. This memory area is organized as an array indexed by MTX IDs that stores the corresponding parent VID. Values are written into this area by the `allocate` instruction.

Since the hierarchy must be accessed on each cache request, parts of the hierarchy should be cached. Each data cache maintains a dedicated MTX hierarchy cache that is indexed by MTX ID and that stores the first n ancestors of the given MTX (where n determines the hierarchy cache's line size). Each line also contains an additional bit to indicate whether the line contains an MTX's entire ancestry or just the first n ancestors.

For each cache access, the MTX hierarchy cache must be accessed once for the request version, and once for each VID from tag-matching cache blocks. Two VIDs from different MTXs can be compared using the results from the hierarchy cache. The first common ancestor between the versions is found, and their subTX IDs are then compared. The results of the comparisons are fed to the VCL to allow it to filter out cache lines that should not be read.

Accesses to the MTX hierarchy cache can happen concurrently to the tag and data

accesses in the data caches. The initial access to the hierarchy cache uses the request VID and can occur concurrently with the data cache tag access. Matching VIDs read during from the tag array can then be fed sequentially into the MTX hierarchy cache concurrently with the data cache data array access. Consequently, the hierarchy cache only appears on the critical path if many cache blocks match the requested address necessitating many serial lookups into the hierarchy cache.

Finally, misses to the MTX hierarchy cache require that the in-memory MTX tree be accessed. Just as for virtual to physical address translation, dedicated hardware can walk the memory data structure, or the responsibility can be passed onto a software handler. Additionally, many cold misses can be avoided by inserting entries into the cache after an `allocate` instruction executes using information about the parent which may already be stored in the cache.

7.3.5 Commit and Rollback

Commit and rollback are easily handled within the design. The `commit.p1` and `commit.p3` instructions simply need to acquire and release a lock based on the VID of the parent subTX. This can be implemented using conventional memory locks. To handle the `commit.p2` instruction, for each modified block contained in the cache whose version equals the committing version, the cache must acquire exclusive access to the corresponding line in the parent version. Then, the committing block can be merged into the parent block, and finally the committing block can be discarded. Commit should proceed from higher level caches to lower level caches. This prevents the commit of a line in a lower level cache from causing the corresponding line in a higher level cache from being displaced. Note, commit performance can be improved using a structure similar to the ownership-required buffer (ORB) [69] to prevent scanning the entire cache on commit.

Rollback is much simpler, with each cache in the system discarding any cache block whose VID is greater than or equal to the VID of the rollback request. To ensure child

MTXs also get rolled back, the transaction hierarchy must be consulted to generate appropriate rollback messages for all child MTXs. Since rollback does not need to block the execution of any core in the system, its performance is not important.

7.4 Detecting Conflicts

The system described in the previous section implements the semantics of MTXs without hardware conflict detection. The system described thus far is what is used by the current implementation of SpecDSWP. For completeness, this section describes how to augment the system to detect conflicts.

Recall that an intra-transaction conflict occurs if a subTX writes a location after a later subTX has read the location without first writing it. An inter-transaction conflict occurs if one subTX writes a location that another, incomparable subTX reads without first writing.

Given these definitions, the system must track which locations have been read without first having been written to detect conflicts. We refer to such reads as *upwards-exposed uses* [68]. The MTX system uses the U_k bit stored per byte to track upwards-exposed uses (see Figure 7.7)[2].

The MTX cache system must set U_k to $U_k \vee \overline{W_k}$ each time it reads data from the cache. This ensures any bytes that are satisfied by a block whose VID is less than this block's VID are marked. The coherence protocol is then extended to detect intra-transaction conflicts when invalidating bytes. If a coherence request forces bytes to be invalidated (i.e., on a read exclusive request with $ID_{request} < ID_{block}$), U_k is set, and P_k is to be cleared, then a conflict has occurred. To detect inter-transaction conflicts, the coherence protocol must also handle the case when $ID_{request}$ and ID_{block} are incomparable. In particular, for an incomparable read request, if W_k is set for any byte in the block then a conflict has occurred. Similarly for an incomparable read exclusive request, if U_k is set for any byte in the block then a con-

[2]If conflict detection is not implemented in hardware, the U_k bits are unnecessary.

flict has occurred. In all conflicting cases, either the requesting transaction, the snooping transaction, or both may be rolled back.

The system requires one more change to support conflict detection. Recall that a read request hits if a cache has a block whose VID is less than the request ID and the block is not stale. Read requests, however, now also modify cache blocks by potentially setting U bits. Since the access should not modify blocks from previous subTXs, such "hits" should cause the cache line to be duplicated for the request version (i.e., the block ID should be set to the request ID). Since such duplication can occur at arbitrary caches in the hierarchy, strict inclusion will no longer be satisfied. A higher level cache can contain a block with a specific VID that its lower level cache does not contain. Consequently all coherence actions must affect all caches in a particular sub-tree of the hierarchy.

7.5 Other Implementation Possibilities

This chapter has focused on describing an extension to an invalidation-based cache coherence protocol to implement multi-threaded transactions. The literature, however, is abound with implementation alternatives for traditional, single-threaded transactional memories. This section briefly surveys some of these alternate methodologies and identifies challenges in adapting these strategies to multi-threaded transactions.

Some implementations of transactional memories use *eager*, rather than *lazy*, version management. In lazy versioning systems, speculative writes to a location are stored in a speculative buffer while the non-speculative value safely remains in the virtual address space of the process. In the event of a rollback, one simply discards the speculative buffers. Commit, however, requires copying data from the speculative buffers into the virtual address space. In the MTX implementation described in this chapter and in many lazy versioning, single-threaded hardware transactional memories, the speculative buffer is simply comprised of the processor's cache memories.

Recognizing that commit is often more common than rollback, eager implementations, such as LogTM [51] and UTM [10], store speculative values directly in the virtual address space of the process, and non-speculative values are buffered in a rollback log. In these systems, commit does require any special processing, and rollback requires restoring values in the virtual address space using the rollback log. This approach, unfortunately, is fundamentally at odds with MTXs. Recall that with MTXs many versions can exist per address and each version can be directly addressed. This approach however, relies on only a single speculative version existing per address and immediately detects a conflict if a different version is accessed. It may be possible, however, to use the spirit of this approach as an optimization to an MTX implementation. To improve commit speed, the oldest speculative version could be stored directly in the virtual address space. The current committed state could be stored in a rollback log under the assumption that it will be never be accessed or accessed only infrequently. Younger versions can be buffered in caches as was described in this implementation. This approach optimizes for commit, rather than rollback, and can help keep commit off the critical path.

In addition to different version management approaches, researchers have proposed alternate methodologies for performing conflict detection. Ceze et al. propose *bulk disambiguation*, a novel mechanism where, rather than having processor cores generate invalidation messages each time a speculative cache line changes state, each core maintains two signatures representing the set of addresses read from and written to [17]. During commit, these signatures are broadcast and compared at each processor core to detect whether or not a conflict has occurred. Using careful construction of the signatures and certain versioning restrictions, the set of addresses written in a particular version can be reconstructed from the signature, thus allowing a cache to properly commit or rollback. This approach is appealing because it dramatically reduces the design complexity of conflict detection and reduces the space overhead of tracking the read set and write set of a speculative version. For an MTX system that does hardware conflict detection, a Bulk-like approach could be

used to reduce overhead and design complexity. However, Bulk does not address any of the issues related to finding what cache contains the freshest speculative version of data that is older than a given request version. Consequently, the data locating pieces of the coherence protocol described in this chapter would still be necessary.

In addition to hardware schemes, many have proposed implementing transactional semantics purely in software [4, 36, 48, 62]. Both IBM and Intel have recently released STM implementations to the research community [1, 2]. Software transactional memory (STM) systems come in a variety of flavors, however the design choices are similar to the ones for hardware transactional memories. First, an STM may buffer speculative state in dedicated buffers, or store it directly at the target location and use an undo log to recover state in the event of a conflict. Second, STM systems must identify conflict. Due to the expense of checking for conflict on each access, most STM systems log their read and write sets during transaction execution. At commit, the STM verifies that no other thread has committed values to locations in the read set after the values were read. Once this is verified, the thread merges its speculative data into committed state. Depending on the speculative buffering scheme, this may simply involve broadcasting the write set and discarding the undo log, or it may involve copying data from the speculative buffers into architectural state.

The STM approach to transactional memory is exciting because it enables the transactional approach on today's hardware. However, the key obstacle in these systems has been performance. Managing the transactional meta data associated with each transactional object significantly introduces the overhead of each load and store operation. Researchers have shown that with increased thread counts, this overhead can be overcome [62]. However, naïvely extending STMs to support MTXs would certainly increase this overhead by necessitating expensive search operations to find the latest copy of a datum. Developing an STM MTX implementation that scales well remains an interesting avenue of future research.

Chapter 8

Evaluation of Speculative DSWP

This section evaluates our implementation of Speculative DSWP in the VELOCITY backend optimizing compiler [71]. VELOCITY uses the IMPACT [64] compiler as its front-end. IMPACT reads in C code and emits a low-level intermediate representation that virtualizes the Intel Itanium® 2 architecture. VELOCITY relies on IMPACT's pointer analysis [19, 53] to identify both call targets of function pointers and the points-to sets of load, stores, and external functions.

The VELOCITY backend targets the Intel Itanium® 2 architecture and contains an aggressive classical optimizer, which includes global versions of partial redundancy elimination, dead and unreachable code elimination, copy and constant propagation, reverse copy propagation, algebraic simplification, constant folding, strength reduction, and redundant load and store elimination. VELOCITY also performs an aggressive inlining, superblock formation, superblock-based scheduling, and register allocation.

The VELOCITY implementation of Speculative DSWP supports both speculative parallel-stage DSWP *and* interprocedural parallelization. Further, the implementation supports the speculation types described in Chapter 5. This section evaluates SpecDSWP on benchmarks from the SPEC® CPU2000, SPEC® CPU92, and Mediabench suites.

Evaluating SpecDSWP on parallelizations containing up to 32 threads on loops ac-

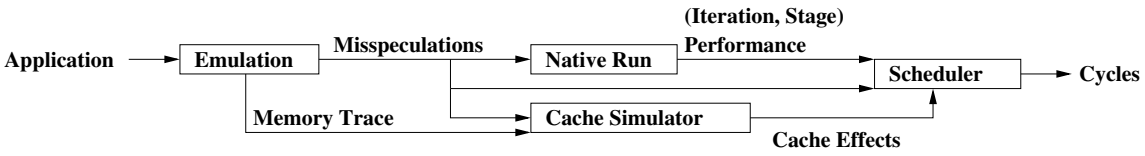


Figure 8.1: The measurement flow for evaluating SpecDSWP parallel performance.

counting for up to 95% of the application's run time precluded using conventional cycle-accurate simulation due to excessive simulation time. Section 8.1 describes the simulation methodology used to evaluate SpecDSWP. Section 8.2 will present summary results for the parallelized applications and will present several case studies deconstructing the results further highlighting strengths of SpecDSWP and MTXs.

8.1 Experimental Methodology

For many of the benchmarks studied, SpecDSWP was applied to the outer most loop in the application. Native executions of these applications ran for tens to hundreds of billion cycles with each loop iteration often taking in excess of a billion cycles. Unfortunately, using traditional cycle-accurate simulators on such large code regions with as many as 32 threads would require excessive simulation time. Further, sampling methodologies like SMARTS [77], TurboSMARTS [76], or SimPoint [57] are inapplicable for evaluating multi-threaded applications. Further, since SpecDSWP relies on the synchronization array (SyncArray) [61] for inter-thread scalar communication and MTXs for speculative state buffering, evaluation on today's multi-core processors was also not possible.

Consequently, to evaluate the performance of a loop parallelized with SpecDSWP, a combination of native execution, to evaluate the single-threaded performance of each thread, coupled with simulation, to measure the effects of parallelism and MTX cache coherence, was used to measure parallel performance. Figure 8.1 illustrates the overall measurement flow. The emulation stage is used to verify application correctness and to identify misspeculation that occurs at runtime. A specially synchronized native binary is

then used to measure the performance of each sequential region across all threads in the application. An MTX cache simulator is then used to calculate the effects of cache coherence on performance. Finally, a scheduler collates the information to estimate the application's performance. Each of these steps is described in more detail below.

8.1.1 Emulation

The SpecDSWP code emitted by VELOCITY cannot be run on existing multi-core Itanium® 2 processors for three reasons.

1. First, the code assumes the existence of a SyncArray. The SyncArray works as a set of scalar queues with efficient enqueue and dequeue operations[1]. The Itanium® 2 ISA was extended with `produce` and `consume` instructions for inter-thread communication. As long as the SyncArray queue is not empty, a `consume` and its dependent instructions can execute in back-to-back cycles.

2. Second, the parallel code relies on hardware support for MTXs. The Itanium® 2 ISA was extended with the operations shown in Table 7.1.

3. Third, SpecDSWP relies on an asynchronous `resteer` instruction that interrupts the execution of the specified core and restarts its execution at the given address.

To check correctness of the parallel code, a library was used to emulate the behavior of the above operations. Instances of `produce` and `consume` were replaced with calls into a library that managed a virtual SyncArray. MTX operations were also replaced to call into an MTX emulation library. Additionally, since MTXs change the semantics of loads and stores after an `enter` operation, all loads and stores were replaced with calls into an the MTX emulation library. This library maintained each memory version as a hash table. Consequently, stores were emulated with a hash table insertion, and loads were

[1]The latency between an enqueue and a dequeue need not be low. However, SpecDSWP does assume that the latency within a thread for executing a enqueue or dequeue operation is low.

emulated using a series of hash table lookups searching for the memory version with the latest value for a given address. Finally, `resteer` operations were replaced with calls into the emulation library. Resteer was emulated using POSIX signals to interrupt the execution of a thread. The signal handler changed the register state saved by the kernel's signal handling mechanism so that returning from the signal handler jumps to the desired recovery code. The standard recovery code, as emitted by the compiler, was used to recover from misspeculation.

In addition to emulating the behavior of the missing architectural features, the code is instrumented to record the set of loop iterations that suffered misspeculation, and to record a memory trace. The former is used to drive the native execution phase of the simulation infrastructure, while the latter is used to drive the cache simulation.

8.1.2 Native Execution

Since the emulation libraries are heavy weight measuring the performance of the emulation runs is not representative. There are three primary sources of performance degradation in the emulation runs.

1. The performance of load and store functions in the MTX emulation library is significantly slower than a native load or store, and significantly slower than would be offered by a hardware MTX implementation.

2. The performance of produce and consume functions in the emulation library is significantly worse than the performance of native produce and consume instructions.

3. Substituting function calls for instructions destroys ILP around the respective operations.

To estimate performance without suffering from these performance penalties, a specially prepared application binary, which will be referred to as the performance binary, is

run on native Itanium® 2 hardware, and the processor's performance counters are used to record performance.

To avoid the first and third pitfalls, the performance binary uses native load and store operations rather than replacing them with function calls into an emulation library. Execution correctness is guaranteed by carefully orchestrating the order in which the threads run. Recall that for SpecDSWP, the set of memory versions used are totally ordered. Provided that rollback is unnecessary, and all operations in one memory version execute prior to any operation in a subsequent version, conventional load and store operations will faithfully implement the semantics of MTXs.

The performance binary uses the misspeculations recorded during the emulation run to avoid corrupting memory state with speculative stores. This obviates the need to support rollback. For each iteration that would misspeculate, each thread in the performance binary jumps directly to the recovery code rather than using the asynchronous resteer mechanism. The threads then execute the remainder of the misspeculated iteration normally.

To ensure that memory operations execute in version order, only one thread is allowed to execute at a time. This is achieved using an execution token that is passed from thread to thread. Only the thread with the token is permitted to execute. The performance binary records the current memory version, and the first thread in the pipeline is allowed to execute until it encounters an `enter` instruction attempting to enter a version later than the current memory version. Just as during emulation, the `enter` instruction is replaced with a library call, however, the implementation for the performance library simply passes the execution token to the next thread in the pipeline. When the commit thread passes the execution token back to the first thread, the current memory version is incremented by one. Using this approach, all threads are permitted to make forward progress, but the semantics of MTXs can be preserved while using conventional load and store operations.

Since only one thread executes at a time, wall time is an inappropriate performance measure. Instead, hardware performance counters are used to measure the number of cy-

cles spent in each sequential region of execution in each thread. These values are summed to compute the total number of cycles spent by each thread in each iteration. Since produce, consume, and MTX operations are still implemented with library calls (the second performance pitfall from above), the performance counters are stopped before each library call and restarted after the call returns. The serialization between disabling and re-enabling the performance counters simulates these operations as single-cycle operations. Since ILP is still disturbed near these function calls, the measurements from this simulation infrastructure are *conservative* approximations of the actual execution times for the sequential regions.

The experiments described below were run on an unloaded HP workstation zx2000 with a 900MHz Intel Itanium 2 processor and 2GB of memory, running CentOS 4.5. Runtime numbers were gathered using version 3.1 of the `libpfm` performance monitoring library [26]. The runtime of an application was determined by monitoring the `CPU_CYCLES`. Since an MTX cache simulation is used to measure cache effects, the infrastructure also monitored the `BE_EXE_BUBBLE_GRALL` and `BE_EXE_BUBBLE_GRGR` events. These events measure the total number of cycles stalled in the integer pipeline and the total number of cycles stalled in the integer pipeline due to non-memory dependences, respectively. The difference between these counters is a conservative estimate[2] of the number of cycles lost due to cache misses. This difference is subtracted from the total cycles measured to estimate the performance of each sequential region modulo cache effects.

8.1.3 Cache Simulation

In the native executions, cycles spent stalled on cache misses were discounted. To account for cache effects, both local and due to coherence, the memory traces generated during the emulation runs were fed into an MTX cache simulator. Table 8.1 shows the parameters

[2]The estimate is conservative because it does not account for stalls due to floating point loads and stores, and it does not account for lost ILP due to cache misses. It only accounts for cycles where no instructions could execute due to cache misses in the integer pipeline.

	L1D	L2	L3	L4
Size	16 KB	256 KB	12 MB	64 MB
Line size	64 bytes	128 bytes	128 bytes	128 bytes
Associativity	4 way	8 way	12 way	256 way
Minimum Latency	1 cycle	5 cycles	10 cycles	50 cycles
Merge Latency (per line)	1 cycle	2 cycles	7 cycles	25 cycles
Commit Latency (per line)	1 cycle	1 cycle	1 cycle	1 cycle
Sharing	Private	Private	Shared	Shared

Table 8.1: MTX cache parameters

for the caches simulated. All caches stored speculative state making the speculation level between the L4 cache and main memory. The latency for a cache access is given by the minimum latency shown in the table plus the merge latency multiplied by the number of lines that were merged to satisfy a given request. For commits, the simulation modeled peer caches committing in parallel. However, different levels of caches committed sequentially starting from the L1 caches and proceeding toward the L4 cache. For each cache, the total time to commit is given by the commit latency multiplied by the number of cache lines present in the given cache in the committing memory version. For rollback, all caches discarded speculative data in parallel. Within each cache, a flash rollback mechanism was modeled. Finally, the simulated caches did *not* support hardware conflict detection as this is handled in the code generated by the VELOCITY SpecDSWP implementation.

This cache configuration was used for all simulations, regardless of the number of threads generated by SpecDSWP. For all the applications explored, speculative state never overflowed from the speculative caches.

The cache simulator outputs a cycle count per thread per loop iteration. Adding the results of the cache simulation to the native execution measurements estimates the total time taken per loop iteration by each thread by *conservatively* assuming no instructions (on the core executing the memory operation) execute in the shadow of a cache miss.

8.1.4 Scheduling

Given the execution time for each loop iteration in each thread, a scheduler is used to compute the total loop execution time accounting for parallelism between the threads. The scheduler models a 32-core Itanium® 2 system. Each thread is assigned to a core, and the scheduler starts a particular loop iteration for a thread only when the respective thread has completed executing the previous loop iteration assigned to it *and* when all data dependences (i.e., produce/consume dependences) are satisfied. The scheduler only starts a particular iteration for a given thread after the *entire* iteration has completed in a dependent thread. This models the worst case where a produce instruction at the end of an iteration feeds a consume at the beginning of the corresponding iteration. Misspeculation is modeled by forcing all threads to synchronize, then using the measured performance for the misspeculated iteration. Speculative execution recommences after the misspeculated iteration completes. The execution time for the parallel loop is taken to be the latest time that any core completes its last loop iteration.

8.1.5 Measuring Baseline Performance

To ensure a fair comparison, the baseline, single-threaded performance was measured using a similar methodology. The single-threaded code was instrumented to measure the execution time of each loop iteration using the hardware performance counters. Just as in the multi-threaded case, the raw cycle count was adjusted to discount time spent stalled on cache misses. A memory trace from the single-threaded execution is fed to the cache simulator to compute the time taken by each loop iteration in the memory system. The sum of the adjusted cycle count with the result of the cache simulator gives the time taken for each loop iteration. The sum of these times determines the total time taken to execute the original, single-threaded loop.

Benchmark	Function	% of Runtime	Average Iterations per Invoc.	Largest SCC Weight		# of Saved Regs	Speculation Types	% of Iters. with Misspec.
				N/S	Spec.			
175.vpr	`try_swap`	99%	5971.0	99.57%	3.31%	13	A, CV, CF, MV	9.1%
181.mcf	`primal_net_simplex`	75%	17428.8	98.37%	44.55%	10	CF, SS, MV	0.2%
197.parser	`batch_process`	98%	345.0	99.31%	0.01%	9	CV, MV	0.0%
256.bzip2	`compressStream`	77%	12.0	99.88%	1.58%	10	A, MV	0.0%
052.alvinn	`main`	98%	30.0	78.56%	38.33%	9	MV	0.0%
mpeg2enc	`dist1`	56%	6.3	100.00%	1.64%	10	CF	15.9%

N/S = Non-Speculative, A = Alias, CV = Committed Value, CF = Control Flow, SS = Silent Store, MV = Memory Versioning

Table 8.2: Benchmark Details

8.2 Experimental Results

Table 8.2 shows statistics about the loops chosen for parallelization. These loops account for between 56% and 99% of the total benchmark execution time. Table 8.2 also presents statistics about the largest SCC in the PDG for each loop both before and after speculating dependences. The weights are estimated by summing the profile weights in each SCC and normalizing by the sum of the profile weights for all the operations in the loop. If an SCC did not have any dependences carried by the loop being parallelized, it was a candidate for inclusion in a parallel stage. Consequently, its weight is never considered the largest. Notice that for all the benchmarks, speculation dramatically decreases the weight of the largest SCC. As was described in Chapter 2, this dramatically increases the maximum speedup attainable by SpecDSWP.

All loops were compiled to generate 3 worker stages. In each case, the compiler was given the freedom to use parallel-stage DSWP to replicate any stage and use at most 31 threads. One thread was reserved to act as the commit thread. Figure 8.2 shows the speedup over single-threaded execution for these parallelizations. For each loop, the loop speedup and the benchmark speedup (assuming only the given loop is parallelized) are shown. In addition, the graph shows the number of threads needed to attain this speedup. Note, in the graph the speedup is absolute (not percent speedup) and both y-axes are logarithmic. All benchmarks exhibit considerable loop speedup ranging from 1.6x to 18.1x. The average speedup over all parallelized loops was 3.23x. The rest of this section examines the results of several of benchmarks in more detail to highlight interesting features of SpecDSWP and

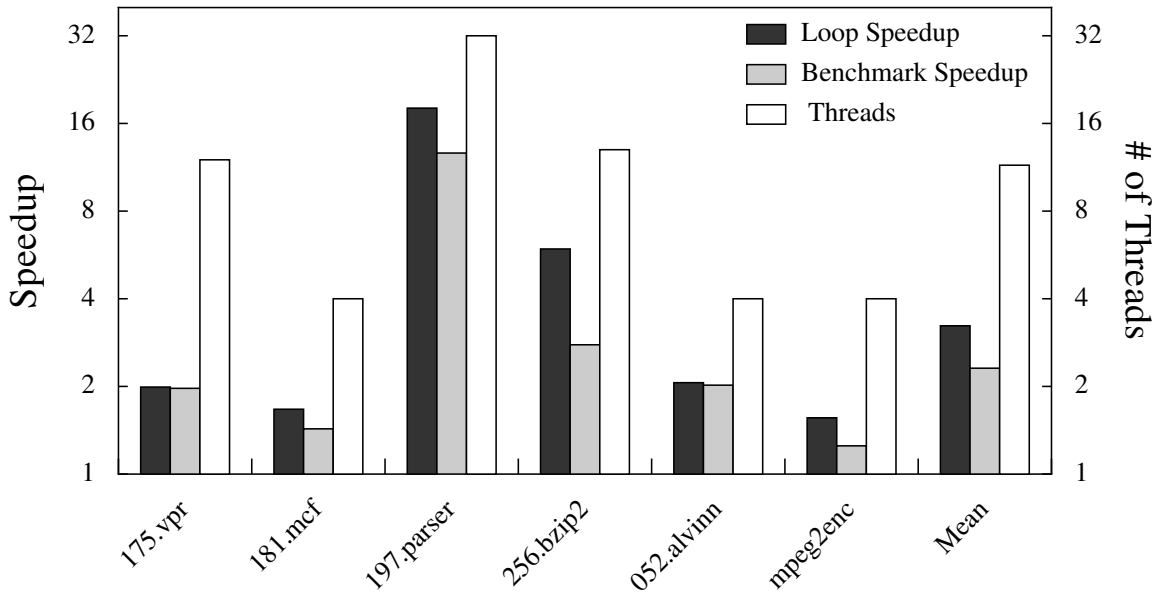


Figure 8.2: Speedup vs. single threaded code.

MTXs.

8.2.1 A Closer Look: Misspeculation

Of the loops that were parallelized, only `mpeg2enc` and `175.vpr` suffered a significant performance loss due to misspeculation. This performance loss is directly attributable to the delay in detecting misspeculation, the significant misspeculation rate, and the cost of recovery. For `175.vpr`, the misspeculation and its penalty are unavoidable for the parallelization chosen by the heuristic (see Section 8.2.3. However, for `mpeg2enc`, the loss may be avoidable.

In `mpeg2enc`, only loop exits were speculated. However, as Table 8.2 shows, `mpeg2enc` has only a few iterations per invocation. This translates to significant misspeculation. As SpecDSWP is currently implemented, upon misspeculation, state is rolled back to the state at the beginning of the iteration, and the iteration is re-executed non-speculatively. However, in the case of a loop exit misspeculation, observe that no values have been incorrectly computed, only additional work has been done. Consequently, provided no live-out values have been overwritten, it is only necessary to squash the speculative work; the iteration

does not have to be rolled back and re-executed. The savings can be significant. For example, in `mpeg2enc`, on average a single-threaded loop iteration takes 120 cycles. The parallelized loop iteration, conversely, takes about 53 cycles. After one misspeculation, due to a cold branch predictor, the recovery iteration took 179 cycles to execute. Reclaiming these 179 cycles would significantly close the performance gap between the iterations without misspeculation and all the iterations. Estimating the resulting speedup by ignoring the last iteration of each loop invocation yields a 2.24x loop speedup corresponding to a 1.45x benchmark speedup. Augmenting the SpecDSWP implementation with this optimization has been left to future work.

8.2.2 A Closer Look: Unspeculation

Recall from Section 5.2 that unspeculation allows SpecDSWP to consider many dependences for speculation and then select only those that are important for a particular parallelization. For the loops parallelized in this evaluation, unspeculation played a particularly important role in `175.vpr` and `181.mcf`. In both of these applications, it was not possible to select alias frequency and branch bias thresholds that would allow the compiler to speculate dependences important for parallelization without causing the compiler to speculate extraneous dependences that lead to degraded runtime performance. In fact, without unspeculation, all values of the parameters either lead to no parallelization opportunity *or* a runtime slow down when compared to the single-threaded execution. Unspeculation, however, allowed the set of speculated dependences to be selected *intelligently* leading to loop speedups of 1.99 and 1.67, respectively. While not shown here, applying SpecDSWP to several SPEC® applications, hand-modified to expose parallelism [14], also necessitated unspeculation to provide the compiler sufficient freedom to speculate dependences inhibiting parallelization while not causing over-speculation.

8.2.3 A Closer Look: Committed Value Speculation

Two of the parallelized loops, `175.vpr` and `197.parser`, benefited from committed value speculation. Recall, this speculation allows a thread to read the value of a memory location from committed state as opposed to using the most recent speculative value. In the context `197.parser`, the main loop reuses several variables from iteration to iteration. However, rather than re-initializing these variables at the beginning of each iteration, the code relies on canceling operations in the previous iteration. For example, each iteration inserts items into a linked list, however each item is removed before the iteration completes. Consequently, at the beginning of each loop iteration the list is empty. However, compiler analyses could not prove the absence of a loop-carried dependence this dependence prevents a large SCC from being allocated to a parallel-stage. However, the loop-carried dependence can be speculatively broken with committed value speculation. Since the state of the list in committed memory is always empty, the speculation never fails and enables `197.parser` to be parallelized with no runtime misspeculation. Notice that alias misspeculation would not suffice because the list's contents do frequently change meaning that corresponding loads and stores do frequently alias.

In `175.vpr`, the parallelized loop is attempting to find the optimal placement of logic blocks in a circuit. The application uses simulated annealing to estimate the optimal placement. In the algorithm, given an initial block placement, two logic blocks are selected at random and their positions are swapped. If the swap improves the quality of the placement, the logic blocks remain in their new positions. If the quality decreases, the algorithm randomly decides whether to keep the blocks in their new positions. The random decision is controlled by "temperature". The higher the temperature, the more likely the algorithm will retain an swap that reduced the placement's quality. This allows the algorithm to escape a local minimum in search of a better global minimum. With each invocation of the loop, the temperature is decreased to ultimately converge to a final placement.

Parallel-stage SpecDSWP was used to parallelize the loop. In the first pipeline stage

in the parallelization, several random numbers are computed. This process is sequential since a pseudo-random number generator is used. The parallel stage was responsible for using the random numbers to swap blocks, check the quality change, and swap back if necessary. Two instances of the parallel stage should not conflict if the blocks selected for swapping were different and did not affect the quality calculation, *or* if the earlier instances of the parallel-stage ends up undoing its swap. Unfortunately, with traditional alias speculation, even if the blocks are swapped and then unswapped, the loads from the subsequent iteration will be flow dependent on the unswapping stores from the previous iteration. Committed value speculation once again allows the later iterations to assume that the earlier ones will not retain their swap thus enabling parallelism. While the temperature is high, the speculation fails frequently leading to little if any parallelism. However, in later loop invocations, when the temperature is low, few swaps are retained leading to significant parallelism.

The performance of `175.vpr` could be improved further if the architecture allowed a thread to push a cache line to another thread. Recall from Section 6.1.3 that alias and committed value misspeculation is detected in software. This means for each dynamic speculation, two threads will load the same address; one thread will speculatively load a value and then another will non-speculative reload the value to confirm the speculation. In `175.vpr`, the second load was almost always an L2 cache miss slowing the thread verifying speculation. However, if the thread executing the first load were to "prefetch" the value for the second thread, then this penalty could be avoided. Figure 8.3(c) shows the performance of `175.vpr` with and without this optimization. The points labeled "Base" do not use the optimization, while the points labeled "Push" measure the performance had the optimization been available. On average, this optimization improved the performance by 5% over single-threaded execution. Other applications were not affected by this optimization since the cache miss in the misspeculation detection code represented a small fraction of the execution time of the corresponding thread.

8.2.4 A Closer Look: Memory Versioning

While all parallelized loops rely on MTXs to rollback speculative stores, the loops from `052.alvinn`, `197.parser`, `256.bzip2`, and `181.mcf` benefit from the effective privatization provided by MTXs. In `052.alvinn`, MTXs were used to privatize several arrays. The parallelized loop contains several inner loops, and each inner loop scans an input array and produces an output array that feeds the subsequent inner loop. While the loop is not DOALL (dependences exist between invocations of the inner loops), each inner loop can be put in its own thread. Each stage in the SpecDSWP pipeline, therefore, is one of the inner loops. However, privatization is needed to allow later invocations of an early inner loop to execute before earlier invocations of a later inner loop have finished. Note that `052.alvinn` did not require any true speculation. False dependences broken by memory versioning were sufficient to achieve parallelism.

The privatization necessary in `052.alvinn` could be achieved statically without sophisticated array dependence analysis. In `256.bzip2`, however, static privatization is not as simple. In `256.bzip2`, each loop iteration reads data from the input file, run-length encodes (RLE) it storing the results to a global array, and then applies the remainder of the bzip2 compression algorithm to the data stored in the array. Since the RLE phase can produce variable sized output, the amount of data stored in the array is tracked by a separate global variable, `last`. Later phases of the compression algorithm iterate over the entire contents of the array using `last` as the upper bound for the iteration. Simple points-to analysis concludes that accesses to the array from later iterations of the loop *could* read values stored into the array from earlier iterations. Identifying that this is impossible is necessary for privatization, and therefore also necessary to allow for many SCCs in the dependence graph to be replicated with parallel stage SpecDSWP. Unfortunately, proving this impossibility statically would require whole-program array dependence tests. MTXs provide the privatization dynamically without relying on such analyses. The pattern exhibited by `197.parser` is similar.

Finally, in 181.mcf a linked data structure was privatized by MTXs. In the parallelized loop, execution of the function refresh_potential could be overlapped with other code within the primal_net_simplex loop by speculating that refresh_potential would not change a node's potential (using silent store speculation). MTXs allow subsequent iterations of the loop to modify the tree data structure used by refresh_potential without interfering with refresh_potential's computation. Due to the complexity of the left-child, right-sibling tree data structure, it is unlikely that compiler privatization could ever achieve the effects of the memory versioning provided by MTXs.

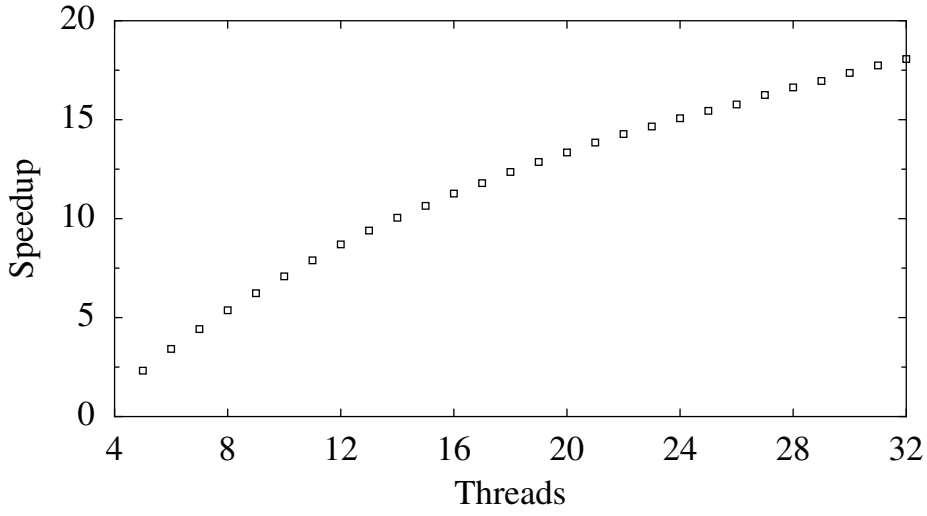
8.2.5 A Closer Look: Scalability

Of the application loops parallelized, 197.parser, 256.bzip2, and 175.vpr benefitted from having a stage of the pipeline replicated by *parallel-stage* SpecDSWP. Consequently, while Figure 8.2 showed the speedup for a fixed number of threads, fewer or more threads could be used depending on the system being targeted. Figure 8.3 shows the performance across a wider range of threads.

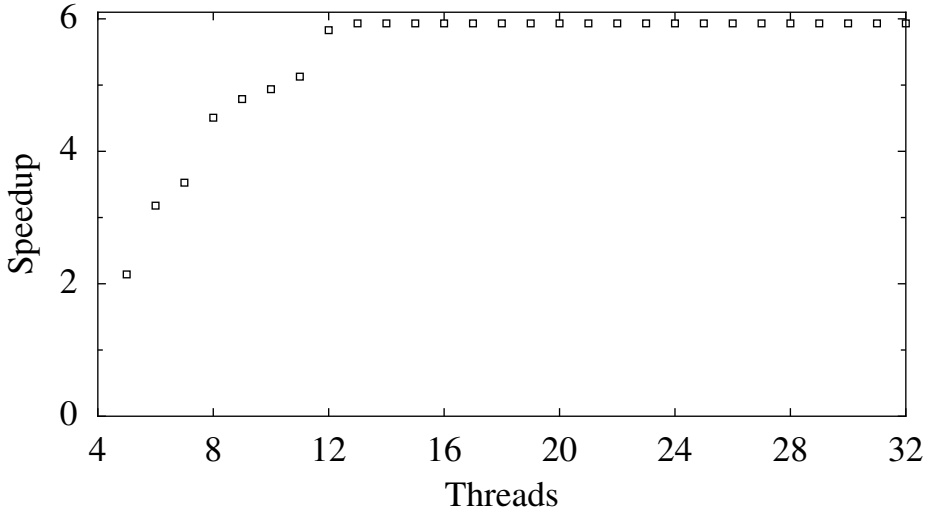
Each of the three applications exhibits different scalability behavior. For 197.parser more threads continues to improve the performance of the application all the way up to 32 threads. The speedup attained is not linearly related to the number of threads due to imbalance in the workload. There are several loop iterations which take 10 to 100 times longer to execute than the other iterations. If each of these iterations is executed by a different thread, the benefit of an additional thread diminishes.

For 256.bzip2, after 12 threads, the performance saturates. Looking at Table 8.2, this is not surprising as, on average, the loop only iterates 12 times per invocation. Consequently, additional threads never receive any work. Larger input sets would be able to leverage many more threads.

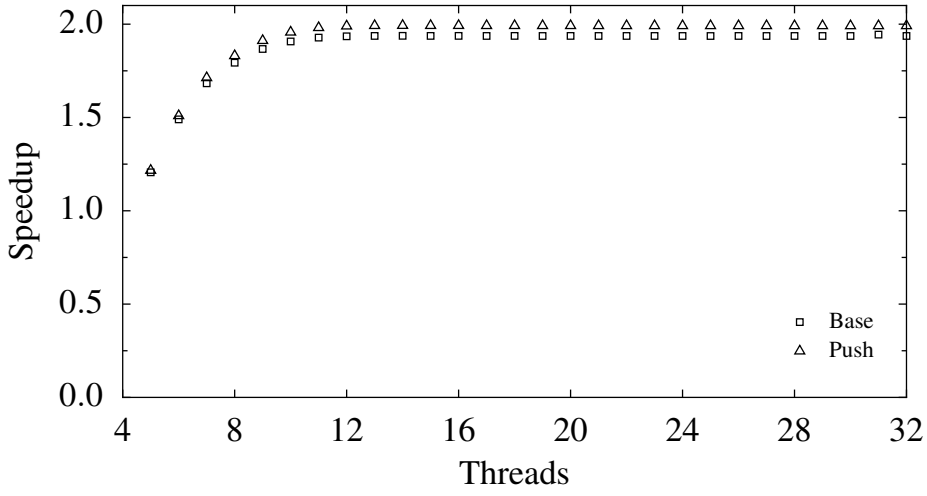
Finally, for 175.vpr, the performance also peaks after 12 threads. Unlike in 256.bzip2, this is not due to small input size, but rather due to misspeculation. As was mentioned



(a) 197.parser



(b) 256.bzip



(c) 175.vpr

Figure 8.3: Scalability of parallel-stage SpecDSWP.

above, in early invocations, nearby iterations are frequently dependent leading to a high misspeculation rate. This eliminates the benefits of replicating a stage using parallel-stage SpecDSWP. In later invocations, on average 30 iterations can execute before a misspeculation. However, after 12 threads, Amdahl's law predicts the diminishing returns; improving the performance of later invocations does not significantly reduce the application's runtime because the early invocations dominate.

In addition to these applications, Bridges et al. have shown that with several small modifications to the application, parallel-stage SpecDSWP is a promising approach for extracting scalable parallelism [14, 15].

Chapter 9

Future Directions and Conclusions

The shift to multi-core processors has marked a major upheaval in the computing landscape. Previously, newer microprocessors delivered increased computing power improving the performance of existing applications and enabling new ones. Compilers smoothed the transition from one microarchitecture to the next by tuning the compiled code for the idiosyncrasies of each processor. However, with multi-core processors, the hardware continues to provide additional compute power, but existing applications do not directly benefit. To leverage this additional compute power, applications must be multi-threaded. Unfortunately, as decades of experience has shown, manually writing correct and well-performing multi-threaded applications is tedious and difficult. Additionally, outside of certain niche domains, compilers have been unable to help programmers automatically translate their applications to leverage multiple processors.

Since pushing the burden of parallelization on to developers would dramatically slow the pace of innovation in computing applications, compilers must overcome the myriad of challenges and bridge the gap between simple programming models understandable by humans and the multi-core programming model exposed by modern hardware. Building on recent promising research, this thesis pushes the state of the art in automatic parallelization forward by introducing powerful speculative mechanisms to extend the reach and appli-

cability of pipelined parallelism transformations. Before concluding the dissertation, Section 9.1 discusses several promising areas of future research. Then Section 9.2 summarizes the contributions of this work.

9.1 Future Directions

This thesis has demonstrated the utility of Speculative DSWP in automatically parallelizing applications. While the groundwork for the technique is presented here, there remain several challenges for this technology to be adopted by main stream developers. Here we summarize the most important future research directions.

- *Dynamic Compilation and Efficient Profiling* - The techniques presented in this dissertation heavily leverage profiling to enable the compiler to optimize for the common case (while preserving correctness) rather than pessimistically assuming the worst case manifests frequently. However, profiling requires application developers to find representative input data to drive the feedback directed optimization. In certain domains, finding representative input is difficult or simply can not be found because many different usage patterns exist.

  Applying the techniques discussed in this dissertation *dynamically* is a promising approach to mitigating the problem of finding representative inputs. However, to enable applying SpecDSWP dynamically, novel low-cost methods to profile running applications are necessary. In particular, low cost techniques to profile memory dependences and value locality are essential.

- *Static Analysis* - While profiling is essential to avoid optimizing for the worst case, profiling is often also used to overcome the limitations of conservative analysis. Just in the applications parallelized in the experimental evaluation of SpecDSWP, there were several instances where speculation was used to break a dependence, when,

in reality the dependence did not really exist. Improving the accuracy of analysis, particularly memory dependence analysis, can reduce the dependence on profiling and potentially improve execution efficiency. If fewer dependences need be profiled, the cost of profiling can be reduced significantly, allowing the profiling to occur at runtime. Similarly, knowing that a dependence is absent avoids introducing code and burdening limited hardware resources to detect and recover from misspeculation. For example, informing the compiler that a false memory dependence does not exist may allow it to not create a new memory version to isolate the two instructions. Research in memory shape analysis [31, 32] seems to be a promising approach to improve memory dependence analysis.

- *Software MTXs* - SpecDSWP relies on MTXs to enable its speculation. Unfortunately, support for even single-threaded transactions has not yet appeared in mainstream microprocessors since introducing support will not benefit any existing applications. Similarly, developers and commercial compilers are not using transactional memories due to the lack of hardware support. Breaking this cycle relies on an efficient software implementation. While a software implementation of MTXs may not perform as well as a hardware one, it can certainly provide the impetus for compiler writers and application developers to begin targeting MTXs. The experience gained with this initial deployment of MTXs will allow hardware manufacturers to identify the degree of hardware support necessary. Ideally, little or no hardware support would be incorporated to directly support MTXs, but instead more general mechanisms upon which fast MTXs as well as other yet to be conceived techniques could be built.

- *Programmer Support* - Finally, this dissertation focused on parallelizing applications automatically, without any assistance from the application developer. While this exercise was enlightening and demonstrated that parallelization is possible in

many cases, limited support from the application developer is likely to extend the applicability of the techniques discussed in this work. For example, work by Bridges et al. has shown that two simple annotations can enable a SpecDSWP compiler to extract parallelism from applications where the techniques described in this dissertation would prove insufficient [14]. Enabling wide-spread use of such annotations requires further research into their general applicability, usability studies to understand how developers would use and misuse these annotations, and new compiler analyses to protect programmers from unintended consequences of such annotations.

9.2 Conclusions

This dissertation extends the state-of-the-art in automatic parallelization by bringing speculation to pipeline parallelism. Past work has demonstrated the virtues of pipeline parallelism. In particular, how it gracefully maps dependences within a loop to acyclic inter-thread communication. This allows for efficient execution even in the face of long inter-core communication latency and dynamic variability in thread performance. This thesis extends the pipeline parallelism paradigm with speculation technology that breaks critical path dependences improving the scalability and applicability of pipeline parallelism. This dissertation demonstrates how pipeline parallelism offers the flexibility to speculate only those dependences which are predictable *and* will yield significant improvements in performance. Further, the dissertation introduced a heuristic algorithm for selecting these dependences to speculate. This algorithm relied on *conditional analysis*, a novel analysis approach, introduced in this thesis, that allows data flow analysis to be performed on many related CFGs simultaneously.

To support SpecDSWP, this dissertation also introduces multi-threaded transactions. Transactional memories, in general, have emerged as a promising platform for multi-threaded programming since they provide a foundation for programmers and compilers to

optimize their applications while ignoring infrequent dependences that inhibit parallelization. MTXs are a generalization of conventional transactions that allows multiple threads to cooperatively *or* speculatively participate in a single transaction. While designed with SpecDSWP as its initial application, MTXs provide a unified platform for code written in a transactional programming model and for programs automatically parallelized using speculation. MTXs open the door for wider-scale adoption of transactional programming by lifting the barriers imposed by the single-threaded atomicity problem. With MTXs, automatic parallelization frameworks are free to balance the cost of buffering speculative state with the cost of misspeculation recovery. Further, with MTXs, compilers, application writers, and library writers are free to parallelize functions nested within atomic regions in transactional code providing modularity and composability to parallel programs.

In conclusion, this thesis demonstrates that Speculative DSWP is a promising tool in a compiler writer's tool box. Similarly, a runtime system equipped with MTXs opens the door to many potential parallelization techniques. This dissertation demonstrated the promise of these technologies through an implementation of Speculative DSWP in the VELOCITY compiler running on an MTX enabled multi-core processor. The evaluation demonstrated a mean 2.31x speedup across the applications parallelized with misspeculations rates ranging from 0% to only 16%. The techniques presented in this dissertation provide a solid foundation for future work in unlocking the potential of multi-core processors and larger scale parallel systems.

Bibliography

[1] IBM XL C/C++ for Transactional Memory for AIX. http://www.alphaworks.ibm.com/tech/xlcstm/.

[2] Intel C++ STM Compiler, Prototype Edition 2.0. http://softwarecommunity.intel.com/articles/eng/1460.htm.

[3] Intel Threading Building Blocks 2.1 for Open Source. http://www.threadingbuildingblocks.org.

[4] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–37, New York, NY, USA, 2006. ACM.

[5] K. Agrawal, J. Fineman, and J. Sukha. Nested parallelism in transactional memory. In *Proceedings of the Second ACM SIGPLAN Workshop on Transactional Computing*, August 2007.

[6] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[7] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 226–236, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[8] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[9] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–11, New York, NY, USA, 1988. ACM.

[10] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[11] D. I. August. *Systematic Compilation for Predicated Execution*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois, 2000.

[12] U. Banerjee. *Loop Parallelization*. Kluwer Academic Publishers, Boston, MA, 1994.

[13] A. Bhowmik and M. Franklin. A general compiler framework for speculative multi-threading. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*, pages 99–108, 2002.

[14] M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August. Revisiting the sequential programming model for multi-core. In *Proceedings of the 40th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 69–81, December 2007.

[15] M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August. Revisiting the sequential programming model for the multicore era. *IEEE Micro*, January 2008.

[16] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transaction on Computers*, C-35(8):677–691, August 1986.

[17] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[18] P.-S. Chen, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee. Interprocedural probabilistic pointer analysis. volume 15, pages 893–907, Piscataway, NJ, USA, 2004. IEEE Press.

[19] B. Cheng and W. W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–69, June 2000.

[20] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 13–24, New York, NY, USA, 2000. ACM Press.

[21] M. Cintra and J. Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 43. IEEE Computer Society, 2002.

[22] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, 1996.

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

[24] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software: Practice and Experience*, 29(7):577–603, 1999.

[25] P. A. Emrath and D. A. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 89–99, 1988.

[26] S. Eranian. Perfmon: Linux performance monitoring for IA-64. http://www.hpl.hp.com/research/linux/perfmon/, 2003.

[27] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, July 1987.

[28] Z. Fu and S. Malik. Solving the minimum-cost satisfiability problem using SAT based branch-and-bound search. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 852–859, November 2006.

[29] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, 1979.

[30] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *ACM Transactions on Architecture Code Optimization*, 2(3):247–279, 2005.

[31] R. Ghiya and L. J. Hendren. Is it a Tree, DAG, or Cyclic Graph? In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1996.

[32] B. Guo, N. Vachharajani, and D. I. August. Shape analysis with inductive recursion synthesis. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 256–265, June 2007.

[33] L. Hammond, B. D. Carlstrom, V. Wong, M. Chen, C. Kozyrakis, and K. Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24(6), Nov-Dec 2004.

[34] 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.

[35] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, New York, NY, USA, 1998. ACM Press.

[36] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 253–262, New York, NY, USA, 2006. ACM.

[37] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[38] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 59–70, 2004.

[39] D. Kim and D. Yeung. A study of source-level compiler algorithms for automatic construction of pre-execution code. *ACM Transactions on Computing Systems*, 22(3):326–379, 2004.

[40] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.

[41] K. M. Lepak and M. H. Lipasti. Silent stores for free. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 22–31, New York, NY, USA, 2000. ACM Press.

[42] X. Y. Li. *Optimization Algorithms for the Minimum-Cost Satisfiability Problem*. PhD thesis, North Carolina State University, 2004.

[43] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 226–237, December 1996.

[44] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, September 1996.

[45] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. Posh: a tls compiler that exploits program structure. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 158–167, 2006.

[46] D. B. Loveman. Program improvement by source-to-source transformation. *Journal of the ACM*, 24(1):121–145, 1977.

[47] G. R. Luecke, Y. Zou, J. Coyle, J. Hoekstra, and M. Kraeva. Deadlock detection in MPI programs. *Concurrency and Computation: Practice and Experience*, 14(11):911–932, 2002.

[48] V. J. Marathe and M. Moir. Toward high performance nonblocking software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 227–236, New York, NY, USA, 2008. ACM.

[49] P. Marcuello and A. González. Clustered speculative multithreaded processors. In *Proceedings of the 13th International Conference on Supercomputing*, pages 365–372, New York, NY, USA, 1999. ACM Press.

[50] P. Marcuello, J. Tubella, and A. Gonzalez. Value prediction for speculative multithreaded architectures. In *Proceedings of the 32nd annual ACM/IEEE International Symposium on Microarchitecture*. ACM Press, 1999.

[51] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265, Feb. 2006.

[52] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, pages 96–103, February 1979.

[53] E. M. Nystrom, H.-S. Kim, and W.-M. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proceedings of the 11th Static Analysis Symposium*, August 2004.

[54] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, page 303, Washington, DC, USA, 1999. IEEE Computer Society.

[55] G. Ottoni and D. I. August. Communication optimizations for global multi-threaded instruction scheduling. In *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–232, March 2008.

[56] 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*, pages 105–116, November 2005.

[57] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for accurate and efficient simulation. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 318–319, New York, NY, USA, 2003. ACM.

[58] E. Raman, G. Ottoni, A. Raman, M. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the 2008 International Symposium on Code Generation and Optimization*, 2008.

[59] R. Rangan. *Pipelined Multithreading Transformations and Support Mechanisms*. PhD thesis, Department of Computer Science, Princeton University, Princeton, NJ, United States, June 2007.

[60] 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.

[61] 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.

[62] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197, New York, NY, USA, 2006. ACM.

[63] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[64] J. W. Sias, S.-Z. Ueng, G. A. Kent, I. M. Steiner, E. M. Nystrom, and W. mei W. Hwu. Field-testing IMPACT EPIC research results in Itanium 2. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 26–37, June 2004.

[65] J. D. Silva and J. G. Steffan. A probabilistic pointer analysis for speculative optimizations. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 416–425, New York, NY, USA, 2006. ACM Press.

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

[67] Standard Performance Evaluation Corporation (SPEC). http://www.spec.org.

[68] 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.

[69] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 1–12, June 2000.

[70] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *Proceedings of the 8th International Symposium on High Performance Computer Architecture*, pages 65–80, February 2002.

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

[72] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of call statements. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, pages 176–185, New York, NY, USA, 1986. ACM Press.

[73] J. 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.

[74] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, September 2007.

[75] T. Vijaykumar, S. Gopal, J. E. Smith, and G. Sohi. Speculative versioning cache. *IEEE Transactions on Parallel and Distributed Systems*, 12(12):1305–1317, 2001.

[76] 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.

[77] 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.

[78] A. Zhai. *Compiler Optimization of Value Communication for Thread-Level Speculation*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, United States, January 2005.

[79] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *International Symposium on High-Performance Computer Architecture*, 2008.

[80] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 85–96, November 2002.