PARALLELIZATION TECHNIQUES WITH IMPROVED DEPENDENCE HANDLING

EASWARAN RAMAN

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

JUNE 2009

Abstract

Continuing exponential growth in transistor density and diminishing returns from the increasing transistor count have forced processor manufacturers to pack multiple processor cores onto a single chip. These processors, known as multi-core processors, generally do not improve the performance of single-threaded applications. Automatic parallelization has a key role to play in improving the performance of legacy and newly written single-threaded applications in this new multi-threaded era.

Automatic parallelizations transform single-threaded code into a semantically equivalent multi-threaded code by preserving the dependences of the original code. This dissertation proposes two new automatic parallelization techniques that differ from related existing techniques in their handling of dependences. This difference in dependence handling enables the proposed techniques to outperform related techniques.

The first technique is known as parallel-stage decoupled software pipelining (PS-DSWP). PS-DSWP extends pipelined parallelization techniques like DSWP by allowing certain pipelined stages to be executed by multiple threads. Such a parallel execution of pipelined stages requires distinguishing inter-iteration dependences of the loop being parallelized from the rest of the dependences. The applicability and effectiveness of PS-DSWP is further enhanced by applying speculation to remove some dependences.

The second technique, known as speculative iteration chunk execution (Spice), uses value speculation to ignore inter-iteration dependences, enabling speculative execution of chunks of iterations in parallel. Unlike other value-speculation based parallelization techniques, Spice speculates only a few dynamic instances of those inter-iteration dependences.

Both these techniques are implemented in the VELOCITY compiler and are evaluated using a multi-core Itanium 2 simulator. PS-DSWP results in a geometric mean loop speedup of 2.13 over single-threaded performance with five threads on a set of loops from five benchmarks. The use of speculation improves the performance of PS-DSWP resulting in a geometric mean loop speedup of 3.67 over single-threaded performance on a set of loops from six benchmarks. Spice shows a geometric mean loop speedup of 2.01 on a set of loops from four benchmarks. Based on the above experimental results and qualitative and quantitative comparisons with related techniques, this dissertation demonstrates the effectiveness of the proposed techniques.

Acknowledgments

First, I thank my adviser David August for supporting me in all possible ways throughout my life as a graduate student. His active encouragement, and the insightful discussions we had led me to work on automatic parallelization. He gave me sufficient freedom to come up with my own ideas and solutions. At the same time, he was always available to discuss anything related to my research and provide appropriate guidance. He insulated me from funding concerns, enabling me to focus on my research. Finally, he taught me the importance of presenting my ideas in a simple and lucid manner.

Next, I would like to thank the rest of the members of my Ph.D committee. Doug Clark and Teresa Johnson agreed to spend their valuable time in reading my thesis and providing their feedback. Their useful feedback improved the quality of this dissertation by weeding out many errors and improving the presentation. They, along with the other members of the committee, Vivek Pai and David Walker, provided insightful suggestions during my preliminary FPO that strengthened this dissertation. I would also like to thank the National Science Foundation for supporting my research work.

The Liberty group provided a wonderful team environment for pursuing my research. Spyros Triantafyllis was the principal architect of the VELOCITY compiler used in this dissertation. David Penry and Ram Rangan contributed to the cycle-accurate simulator used in the evaluation. Many of the ideas presented in my dissertation evolved from the stimulating intellectual discussions on automatic parallelization with Guilherme Ottoni, Neil Vachharajani and Matt Bridges. As office-mates for four years, Guilherme and I have had many interesting conversations. Neil's insightful feedback on many of my presentations provided greater clarity to my thoughts. Tom Jablin and Nick Johnson helped in proof-reading my dissertation. In addition to those named above, I thank all other members and alumni of the Liberty group for their help throughout my life as a grad student.

Outside my research group, I want to thank a wonderful set of friends I had during my Princeton years. First and foremost, I'm thankful to my dear friend Ram, who was a major influence in my decision to apply to Princeton and join the Liberty group and also was my apartment-mate for two years. Lakshmi gave many words of wisdom during my first few days at Princeton. Fellow Ph.D students who joined Princeton in 2003 including Rohit, Sunil, Sesh, Amit, Ananya, Eddie and Manish helped me keep sane in times of distress and made my Princeton years memorable. My last year at Princeton was fun thanks to the younger grad students including Arun, Prakash, Tushar, Aravindan, Rajsekar, Badam, Aditya and others. During the last 2 1/2 years, the weekly *satsang*s of PHS served as a time for peace and reflection and I am thankful to everyone who actively participated in those *satsang*s.

Several friends who lived away from Princeton stayed in touch and provided moral support. I thank Chandru, Ramkumar, Karthick, Lakshmi, Meena, Man and all my close undergrad friends in India for all the emails, lengthy phone conversations, the occasional vacations together, love and support.

So many teachers have directly or indirectly contributed to my academic progress. In particular, Priti Shankar, my Master's adviser at IISc, played a major part in my interest in compiler optimization. Matthew Jacob taught me the art of reading, understanding and critiquing technical papers. Above all, my undergrad adviser Ranjani Parthasarathi (RP) played an incomparable role in positively influencing my thoughts and ideas both inside and outside the classroom.

I am thankful to my family and relatives for their help and support. In particular I want to thank my *athai*'s family and my *periappa*'s family. My sisters Latha and Uma and their husbands Rajamani and Sriram have always been a strong source of support and encouragement to me in the past 5 1/2 years. In various difficult situations, my sisters' love and encouragement kept me going.

Finally, I would like to thank my father and my mother, the two most important people in my life. They have made numerous personal sacrifices for my sake. My biggest regret in my life is that my father did not live to see me complete my PhD. My mom always wanted me to excel academically and is perhaps the happiest person to see my complete my Ph.D. No amount of words could fully convey my gratitude to her.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Today's computing systems provide a much richer experience to the end user than those of the past. Improvements in commodity hardware systems, in conjunction with compiler optimization technology, have sustained a high growth rate of application performance on these systems. As a consequence, application writers are able to focus their attention on developing complex software systems with new features that enhance user experience without concerning themselves too much about the performance implications of these features.

Recent trends point to a future where application developers may no longer be able to focus their attention on adding rich features without having to worry about their performance impact. As a case in point, consider the graph showing historical performance trend in Figure 1.1. This graph evaluates the performance of commodity computers from various vendors using the SPEC [60] benchmark suite. The Y axis shows the performance based on a normalized SPEC score metric. From 1993 to 2004, the performance of these machines have shown a steady growth as indicated by the regression line. However, from 2004, the rate of growth in performance has decreased significantly.

To understand the causes behind this performance flattening, it is essential to know the causes of the performance trend until 2004. From the hardware side, there were two main trends. The first was the exponential increase in clock frequencies of processors. For instance, the clock frequencies of the x86 family of processors have increased from 5

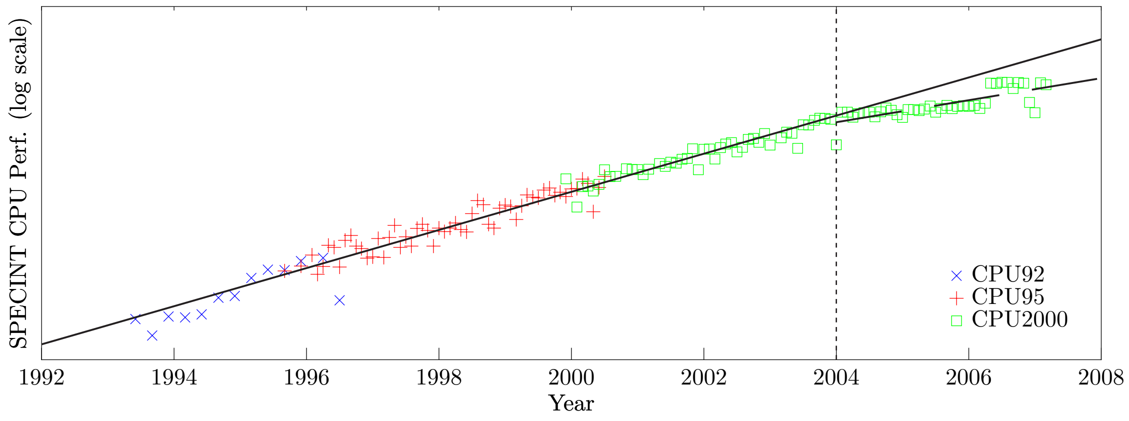


Figure 1.1: Normalized SPEC scores for all reported configuration of machines between 1993 and 2007.

MHz in 8086 to 3.8 GHz in Pentium 4 in less than 30 years. The second was the suite of architectural innovations that made use of the ever increasing transistor counts to reduce execution times. The architectural innovations include improved branch prediction, larger and better caches, multiple functional units, larger issue width, deeper pipelines and out-of-order execution among many others. These architectural innovations are complemented by modern compiler technology that exploits these innovations.

However, a shift in design goals and certain inherent physical limits have significantly impacted the performance growth delivered by the hardware. Power has become a first-class design constraint [41]. This has caused processor manufacturers to scale down the clock frequencies. For instance, the maximum clock frequency of Intel's Pentium M series of processors was 2.26 GHz as against the 3.8GHz of the previous generation Pentium 4 processors. Microarchitectural advancements have also started to give diminishing returns due to design constraints such as power and design complexity.

Amidst this slowdown in performance improvement, the number of transistors available on a die continues to grow at an exponential rate in line with Moore's law [39]. For instance, Figure 1.2 shows how the transistor counts of the current generation Intel processors follow the historical growth trend. The abundance of transistors combined with the inability to leverage them to deliver improved performance has forced a paradigm shift in processor

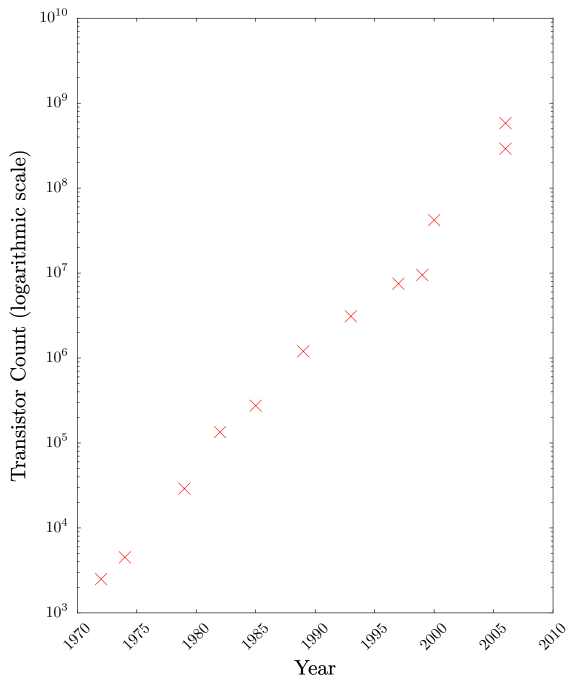


Figure 1.2: Transistor counts for successive generations of Intel processors. Source data from Intel [26].

design. Leading chip manufacturers started to manufacture *multi-core* processors that pack multiple independent processing cores into a single die. Multi-core processors can significantly improve the execution time of certain classes of multi-threaded applications such as servers. However, they do not directly improve the execution latencies of single-threaded applications. Unless the problem of improving the latency of single-threaded applications is addressed, application developers may no longer be insulated from the performance implications of the features they add to enhance end-user experience.

1.1 Approaches to Obtaining Parallelism

There are two predominant approaches to producing multi-threaded applications that deliver good performance on multi-core processors. The first approach is to delegate the task of specifying the parallelism to the programmer. This places a huge burden on the programmer as writing efficient multi-threaded code using existing programming models is much harder than writing efficient single-threaded code. The lock based multi-threaded programming model introduces additional correctness issues such as race conditions, deadlocks, and livelocks. In addition, the programmer has to worry about performance issues such as those associated with locking granularity. The vast body of research trying to address these problems [11, 15, 17, 34, 58] is evidence for the complexity of the task of writing efficient multi-threaded code that places it outside the realm of most programmers. Transactional memory [23, 51] was proposed as an alternative to lock-based multi-threading. Transactional memory provides better composability than locks and eases programmers' efforts in handling concurrency correctly. However, the task of identifying parallel regions is still left to the programmer. This task can be simplified by providing constructs to specify parallelism in programming languages. Languages and language annotations such as High Performance Fortran [33], MPI [40], OpenMP [42], Cilk [19], StreamIt [68] and Atomos [7] provide constructs and annotations to express parallelism. However, these languages allow only *regular* and *structured* parallelism to be easily specified by the programmer. Since many general purpose programs do not exhibit that kind of parallelism, the applicability of these languages is limited.

An alternative approach is to automatically translate single-threaded code written by a programmer into efficient multi-threaded code. This translation requires analyzing large segments of code to identify parallelism and is best done primarily by compilers, with support from runtime systems and hardware. Even if advancements in programming paradigms make the identification and specification of parallelism by the programmer a much easier task, an automatic parallelization solution with comparable performance that insulates the programmer from performance issues is likely to be preferred.

Compiler based automatic parallelization, which is based on the theory of dependences in programs [28], has been used with a high degree of success in the domain of scientific and numerical computation. Several parallelization techniques with varying degrees of applicability and effectiveness have been developed. The DOALL [1, 35] parallelization technique extracts parallelism by executing multiple iterations of a loop in parallel. This technique is limited by the fact that it is applicable only when the loop has no *inter-iteration dependences*. An inter-iteration dependence occurs when the execution of an operation in a loop iteration depends on the execution of an operation in some prior iteration of that loop. DOACROSS [12, 45] also executes multiple loop iterations concurrently but with synchronizations to handle the inter-iteration dependences. These and various other techniques for detecting dependences in loops and transforming them into parallel code were incorporated into research compilers such as Fortran-D Compiler [24], Suif [21, 61], Polaris [4], etc. However, many of these techniques were successful mainly for loops operating on arrays with regular accesses and with very limited control flow. They do not work well for loops in general purpose programs that are characterized by complex or irregular control flow and memory access patterns.

In the last decade, new parallelization techniques were proposed to parallelize general purpose codes with arbitrary control flow and memory access patterns. A vast body of research on speculative multi-threading techniques [3, 22, 27, 36, 59, 63, 70, 78], including thread level speculation (TLS) and other variants, emerged. TLS improves the performance of DOACROSS by adding speculation. A new non-speculative parallelization technique called decoupled software pipelining [56, 44] proposed a different approach to parallelization. DSWP extracted *pipelined parallelism* from a loop by partitioning the body of the loop into pipeline stages which were then executed by different threads. The applicability and performance of DSWP can be extended by applying speculation [72, 73]. While the above techniques have extended the scope of automatic parallelization to general purpose code, the performance evaluation of these techniques indicates there are many applications whose performance do not improve by these techniques. The inability of these techniques to deliver good performance across a wide range of applications is likely to deter automatic parallelization from being widely embraced.

1.2 Contributions

The contributions of this dissertation are two new compiler transformations to parallelize loops in the presence of inter-iteration dependences. These two techniques can extract thread level parallelism from general purpose programs with arbitrary control flow and memory access patterns. By improving the handling of inter-iteration dependences, these two techniques overcome the limitations of several existing techniques. This results in performance advantages that improve the viability of automatic parallelization as a solution to the challenges of the multi-core era.

The first technique is called parallel-stage decoupled software pipelining or PS-DSWP [52] which improves the performance of DSWP by exploiting characteristics of DOALL parallelization in conjunction with pipelined parallelism. DSWP pipelines a loop body by isolating each recurrence of dependences within a pipeline stage and executes each stage using a separate thread. The performance of DSWP depends on how well the dynamic operations of the loop are distributed across the pipeline stages and, in practice, does not scale well as the number of threads increase. A key insight leading to the idea behind PS-DSWP is that the code in certain pipeline stages may be free from inter-iteration dependences with respect to the loop being parallelized. Those stages could therefore be executed by multiple threads, with each thread executing a different iteration of the code inside the stage, similar in spirit to a DOALL execution. As a result, PS-DSWP retains the improved applicability of DSWP, but with better performance obtained as a result of executing iterations concurrently. This dissertation presents the PS-DSWP technique with a description of the key algorithms used in the transformation as well as their implementation details. An extension to PS-DSWP that applies speculation based on the speculative DSWP technique [73] is also presented.

The second technique is a speculative multi-threading technique called speculative parallel iteration chunk execution (Spice) [53]. Spice relies on a novel software-based value prediction mechanism. The value prediction technique predicts the loop live-ins of just a few iterations of a given loop. This breaks a few dynamic instances of inter-iteration dependences in the loop, enabling speculative parallelization of the loop. Spice also increases the probability of successful speculation by only predicting that the values will be used as live-ins in *some* future iterations of the loop. These properties enable the value prediction scheme to have high prediction accuracies while exposing significant coarse-grained thread-level parallelism.

These two techniques are implemented in the VELOCITY automatic parallelization compiler. The two techniques are applied to loops from several general purpose applications and evaluated on a simulated multi-core Itanium 2 processor. PS-DSWP results in a geometric mean loop speedup of 2.13 over single-threaded performance when applied to loops from a set of five benchmarks. The use of speculation improves the performance of PS-DSWP resulting in a geometric mean loop speedup of 3.67 over single-threaded performance when applied to loops from a set of six benchmarks. These results are compared with the performance of DSWP and TLS on these benchmarks. Spice shows a geometric mean loop speedup of 2.01 on a set of loops from four benchmarks. The performance results and the comparison with related techniques demonstrate the effectiveness of the proposed techniques.

1.3 Dissertation Organization

Chapter 2 introduces some of the existing approaches to compiler based parallelization. A simplified analytical model to characterize the performance potential of these techniques is presented. This discussion on current parallelization approaches is important to motivate and understand the contributions of this dissertation. Chapters 3 and 4 discusses the parallel-stage decoupled software pipelining in detail including the code transformation, thread partitioning heuristics and optimizations that enhance the applicability and the performance of the baseline technique. Chapter 5 describes a new approach to value speculation for thread level parallelism and the Spice technique that relies on this value speculation. Chapter 6 presents an experimental evaluation of the techniques proposed in this dissertation. Finally, Chapter 7 discusses avenues for extending the techniques presented in this dissertation and summarizes the conclusions.

Chapter 2

Parallelization Transformations

This chapter presents an overview of compiler-based automatic parallelization transformations. Some general ideas and concepts in automatic parallelization are presented first. This is followed by a discussion on some of the important proposed solutions. The two new techniques presented in this dissertation extend these ideas to overcome some of the limitations of these solutions.

2.1 Overview

Parallelization transformations convert single-threaded code into a semantically equivalent multi-threaded code. The semantic equivalence is guaranteed by ensuring that the multi-threaded execution respects all the dependences present in the single-threaded code. The techniques employ a variety of transformations such as variable renaming, scalar expansion, array privatization and reduction transformations to remove many dependences from the single-threaded code and insert appropriate synchronization to respect the remaining dependences in the multi-threaded code.

While the scope of these transformations can be any arbitrary code region, most of these techniques operate on loops. The iterative nature of the execution of a loop's body and the

```
for(i = 0; i < 10; i ++){
A:    a[i] = b[i-1]+1;
B:    b[i] = a[i]+1;
}
```

Figure 2.1: An example that illustrates the limits of the traditional definition of dependence when applied to loops.

fact that programs typically spend most of their execution time in a few hot loops make loops a suitable candidate for parallelization. Since the techniques discussed in this chapter also operate on loops, the discussion in this chapter is restricted to parallelization of loops.

Traditional definition of dependences between program statements or operations are insufficient in the context of loops. The code example in Figure 2.1 illustrates its limitations. The statement `B` depends on `A` resulting in the dependence arc $A \rightarrow B$. However, `B` in any given iteration is dependent on `A` from the same iteration of the loop and not `A` from earlier iterations. This distinction is crucial for many loop parallelization techniques. The dependence in the above example is called *intra-iteration* dependence. On the other hand, `A` is dependent on `B` from the previous iteration. If a dependence is between a statement in one iteration to a statement in some later iteration, it is said to be an *inter-iteration* or *loop-carried* dependence. The different techniques described in this chapter differ in the way they preserve the dependences in the multi-threaded code.

2.2 Categories of Parallelization Techniques

The different approaches to preserving the original dependences lead to different communication patterns between the threads that execute the parallel code. Many important parallelization techniques can be placed under three broad categories based on the communication pattern between the multiple threads of execution:

Independent Multi-threading (IMT) The IMT techniques are characterized by the absence of any communication between the threads that execute the parallelized loop within the loop body. DOALL is the main parallelization technique in this category.

Cyclic Multi-threading (CMT) Unlike IMT, the parallel threads generated by CMT techniques contain communication operations inside the loop body. If threads are represented as nodes of a "communication graph" and directed edges are used to represent communication between a pair of threads, the threads produced by CMT form a cyclic graph. DOACROSS, and its speculative variant TLS, are the major techniques in this category.

Pipelined Multi-threading (PMT) Like CMT, PMT techniques also generate threads that communicate within the loop body. However, the resulting communication graph is an acyclic graph. DSWP is a major technique in this category.

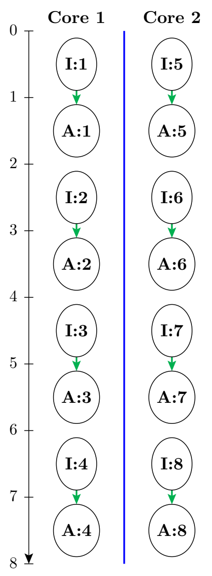
While the above categorization does not exhaustively cover all proposed automatic parallelization techniques, it is sufficient to cover the important techniques closely related to the contributions of this dissertation. In the rest of this section, each of these types of multi-threaded techniques are discussed in detail. An overview of the major representative in each category is first presented. For each category, a simplified analytical model for performance gain is then discussed. This helps in understanding the advantages and limitations of the techniques. Finally, the use of speculation to improve the parallelization is discussed. Speculation is a useful tool available to compiler engineers to overcome the effects of conservativeness in provable static analyses and is an important component in parallelizing general purpose applications.

2.2.1 Independent Multi-threading

One of the earliest proposed IMT technique is DOALL parallelization [35]. IMT techniques are characterized by their ability to extract *iteration level parallelism*. The iterations of the loop are partitioned among the available threads and executed concurrently with no or little restrictions on how the iterations can be partitioned among threads. For instance if a loop executes for 100 iterations and is parallelized into two threads, then a DOALL par-

```
for(i=0; i < N; i++) //I
    a[i] = a[i] + 1;   //A
```

(a) C code



(b) Execution timeline

Figure 2.2: A candidate loop for DOALL transformation and a timeline of its execution after the transformation.

allelization could execute the first 50 iterations in one thread and the next 50 iterations in another thread or execute the odd iterations in one thread and even iterations in the second thread.

This unrestricted iteration level parallelism in IMT requires that the loop has no inter-iteration dependences. Even if the original loop has inter-iteration dependences, transformations such as induction variable expansion, array privatization and reduction transformations can be used to remove or ignore inter-iteration dependences between threads.

Figure 2.2(a) is an example of a loop that can be parallelized using DOALL. The only inter-iteration dependence is the increment of the index `i` in every iteration. This dependence can be ignored by initializing the value of `i` in each thread appropriately so that an iteration does not depend on a prior iteration from a different thread. Figure 2.2(b) shows the execution schedule of a DOALL parallelization of this loop for a total of 6 iterations using 2 threads. The first thread executes the first 3 iterations and the second thread executes the next 3 iterations. The body of the loop executed by both the threads is identical. The initial value of `i` in the second thread is set to `N/2` so that the loop carried dependence can be ignored.

Analytical model

The simple analytical model for measuring DOALL's performance assumes that all iterations of the loop have the same execution latency. In practice, there is some variability in execution latencies of the iterations that could affect the performance of DOALL. Let L_i be the latency to execute an iteration of the loop and let n be the number of iterations. The sequential execution time of the loop is $n \times L_i$. If the loop is parallelized using m concurrent threads, then the execution time of the parallelized loop is L_i if $m > n$ and $\frac{n \times L_i}{m}$ if $m \leq n$. Thus, the speedup obtained by DOALL parallelization is $\frac{n \times L_i}{(\frac{n \times L_i}{Min(m,n)})}$ which is equal to $Min(m, n)$.

Performance characteristics

In the foreseeable future, the number of cores in a chip is likely to increase at an exponential rate. Hence a good parallelization technique should be scalable to a large number of cores. As can be inferred from the analytical model, the speedup of DOALL is linear in the number of cores available as long as there are more iterations than the number of available processors. Since the iteration count of many loops is often determined only by the size of the input set, DOALL scales well as the input size increases. The second advantage offered by DOALL is that its performance is independent of the communication latency between cores since the threads do not communicate within the loop body. Finally, from a compilation perspective, code generation in DOALL is simple since the body of the loop executed by different threads is virtually identical.

```
while(ptr = ptr->next) //LD
    ptr->val += 1;  //A
```

Figure 2.3: A loop with inter-iteration dependence.

The main drawback of IMT is that it is very limited in applicability. The fact that most inter-iteration dependences are precluded by IMT makes it inapplicable to most loops. As an example, consider the loop in Figure 2.3. In terms of functionality, the loop is similar to the one in Figure 2.2(a) since both the loops increment a list of integers. The only difference is that the list is implemented as an array in Figure 2.2(a) and as a linked list in Figure 2.3. The linked list implementation contains an inter-iteration dependence due to the pointer chasing load that cannot be ignored making DOALL inapplicable to that loop.

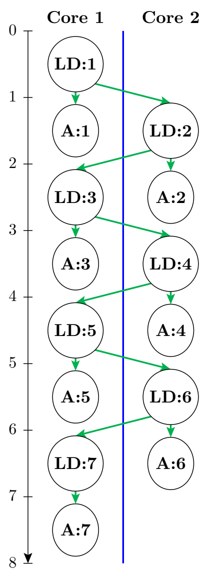
Application of speculation

Static analysis techniques have very limited success in classifying dependences as intra- or inter-iteration dependences. Most of the techniques presented in the literature work only when the array indices are simple linear functions of loop induction variables and conservatively assume inter-iteration dependences for complex access patterns [1]. Hence speculation of inter-iteration dependences provides a way to improve the applicability of DOALL.

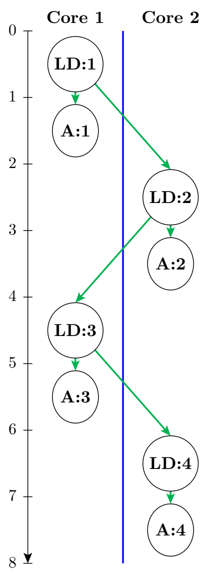
The LRPD test [57] and the R-LRPD test [13] speculatively partition the iterations into threads to execute in a DOALL fashion. Mis-speculation detection and recovery are done purely in software by making use of shadow arrays and status arrays. However this technique is limited to loops with array accesses and does not handle arbitrary loops. Zhong et al. [77] showed that a significant fraction of loops in many programs are speculative DOALL loops particularly after application of several classical transformations.

2.2.2 Cyclic Multi-Threading

Even in the presence of inter-iteration dependences, it is possible to extract a restricted form of iteration-level parallelism by synchronizing the inter-iteration dependences. This is the approach used by the CMT techniques. DOACROSS [12, 45] is an important technique in this category. IMT techniques have *restricted* iteration level parallelism. The parallelism in DOACROSS is also obtained by executing iterations concurrently across threads, but the mapping from iterations to threads is restricted by the presence of synchronization and communication.



(a) Original schedule



(b) Communication latency increased by 1 cycle

Figure 2.4: DOACROSS execution schedule for the loop in Figure 2.3.

Figure 2.4(a) shows the DOACROSS execution schedule for the loop in Figure 2.3. The DOACROSS schedule has the following characteristics: All the odd iterations of the loop are executed by the first thread and the even iterations by the second thread. Synchronization is inserted to respect the inter-iteration dependence due to the pointer chasing load `LD`. Both the threads receive and send synchronization tokens from the other thread resulting in cyclic communication between the threads.

Analytical Model

Let n be the number of iterations of the loop and m be the number of threads used to parallelize the loop. Let us assume that there is only one inter-iteration dependence that needs to be synchronized and let **P** and **C** be the produce and consume points of that synchronization. Let $C(i, j)$ and $P(i, j)$ denote the consume and produce points in iteration i executed by thread j. In each iteration, let $t1$ be the time from the start of the iteration to the consume point, $t2$ be the time between the consume point **C** and the produce point **P** within a single

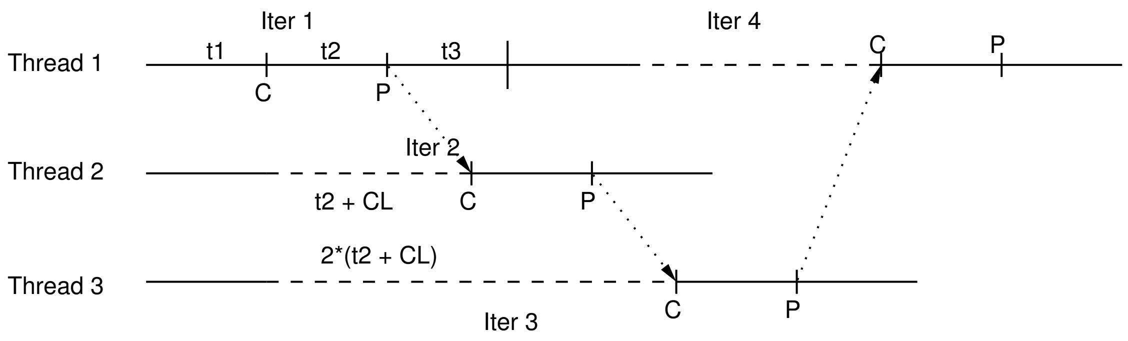


Figure 2.5: This figure illustrates synchronization and associated stalls in DOACROSS for a loop with one inter-iteration dependence. **C** and **P** are consume and produce points of the dependence.

iteration and let $t3$ be the time between the produce point till the end of the iteration. Let CL be the communication latency between two threads.

Figure 2.5 shows the timeline of a DOACROSS execution under these assumptions, with $m = 3$. The solid lines represent execution of loop body, the dashed lines represent stall cycles waiting for synchronization and the dotted lines represent communication between threads. Let SC be the stall cycles incurred by thread 1 between iterations i and $i + m$ which are any two consecutive iterations executed by thread 1. SC is thus the difference between the time when the synchronization token for consume point $C(i + m, 1)$ is available and the earliest time when $C(i + m, 1)$ can execute. Thus

$$SC = Max((C_{synch}(i + m, 1) - C_{earliest}(i + m, 1)), 0)$$

The chain of synchronizations from $C(i, 1)$ to $C_{synch}(i + m, 1)$ contains m segments of length CL (the communication segments) and m segments of length $t2$ (the execution segments). Hence $C_{synch}(i + m, 1)$ can be expressed in terms of $C(i, 1)$ as follows

$$C_{synch}(i + m, 1) = C(i, 1) + m \times (CL + t2)$$

Similarly, $C_{earliest}(i+m,1)$ can be expressed as

$$C_{earliest}(i+m,1) = C(i,1) + t2 + t3 + t1$$

Using the above two expressions, the expression for the stall cycles SC can be simplified to

$$SC = Max(m \times CL + (m-1) \times t2 - t3 - t1, 0) \tag{2.1}$$

Since SC is the number of cycles stalled for every m iterations of the loop, the total stall cycles during the entire execution of the loop is $\frac{n \times SC}{m}$. If $L_i = (t1 + t2 + t3)$ is the sequential execution time of an iteration of the original loop, then the total time to execute the parallelized loop L_{par} is given by

$$L_{par} = \frac{n}{m} \times (L_i + SC)$$

and hence the speedup obtained is given by

$$\begin{aligned} speedup &= \frac{n \times L_i}{L_{par}} \\ &= \frac{n \times m \times L_i}{n \times (L_i + SC)} \\ &= \frac{m \times L_i}{L_i + Max(m \times CL + (m-1) \times t2 - t3 - t1, 0)} \end{aligned}$$

While this analysis has assumed the presence of only one inter-iteration dependence that needed to be synchronized, it can be easily extended for the general case by considering the dependence which has largest value of $t2$ since the synchronization of other dependences will be subsumed by this dependence.

Performance characteristics

DOACROSS gives a linear speedup with m threads provided there are no stall cycles. As the number of stall cycles increase, the speedup deviates farther from the ideal speedup of m. Several factors influence the number of stall cycles SC.

In DOACROSS parallelization, the synchronization of dependences becomes part of the critical path when it could not be completely overlapped with the rest of the computation. Under this scenario, as the value of CL increases, so does the value of SC. This is illustrated by the execution timeline in Figure 2.4(b). As the number of cores on a chip increase, on-chip memory interconnection networks are likely to be more complex with an increased end-to-end latency. This will critically impact DOACROSS performance. A major consequence of this is that DOACROSS becomes unprofitable when applied to hot loops whose per-iteration execution latencies are nevertheless low.

Another factor that contributes to an increase in SC is the value of $t2$. Consider an optimal placement of the produce and consume points such that the produce is inserted immediately after the source of the dependence is executed and the consume is inserted immediately prior to the destination of the dependence. Since an inter-iteration dependence is usually part of a cycle in the dependence graph[1], the value of $t2$ can be approximated by the length of the cycle. Thus the speedup of DOACROSS is limited by the length of the longest dependence cycle. Finding the longest cycle in a graph is an NP complete problem [20], and in practice an optimal placement of produce and consume points is not possible. For instance, if the source and destination of the inter-iteration dependence are nested within some inner function in the presence of complex control flow, the produce and consume points have to be inserted conservatively causing a further increase in the value of SC. Heuristics have been proposed to eliminate redundant synchronizations and improve the placement of synchronization points [8, 50].

[1]Otherwise a transformation like loop rotation [74] can be used to eliminate the dependence.

The last major factor contributing to the value of SC is the number of threads available for parallelization. Note that in Equation 2.1, m is a multiplicative factor to CL and $t2$. As the number of threads increase the stall cycles also increase. This acts as an inherent limitation to the scalability of DOACROSS.

Despite these limitations, DOACROSS may still be a viable technique if the value of various parameters are such that the stalls due to synchronization are contained. Under those circumstances, DOACROSS achieves a linear speedup under the ideal conditions assumed in the analytical model. From the code generation perspective, DOACROSS code generation is not complex since the body of the loop is identical across all threads.

Application of speculation

```
while(ptr = ptr->next) { //LD
    ptr->val += inc;  //A1
    if(foo(ptr->val)) //IF
        inc++;        //A2
}
```

Figure 2.6: A loop with infrequent inter-iteration dependence.

Most TLS techniques [22, 36, 59, 63] are speculative versions of DOACROSS. Speculation is a useful tool in eliminating hard-to-disprove and infrequent inter-iteration dependences. While DOACROSS can parallelize loops that contain inter-iteration dependences, speculating inter-iteration dependences can result in significant reduction of stall cycles thereby improving the performance of DOALL.

Figure 2.6 shows a code example that demonstrates the advantage of applying speculation to DOACROSS. The loop increments the elements of a linked list by `inc`, similar to the loop in Figure 2.3. However the increment amount is neither constant nor loop-invariant. Instead it gets incremented whenever a list element satisfies a complex condition given by the function `foo`. Thus the loop has two inter-iteration dependences: one due to the pointer chasing load `LD` and another from `A2` to `A1`. If `foo` has a very long latency, then the synchronization of the dependence from `A2` to `A1` will be the bottleneck contributing to the stall cycles. However, if A2 is infrequently executed, then A1 can speculatively use the previous value of inc instead of synchronizing the dependence. In that case, only the self dependence between LD needs to be synchronized resulting in a significant reduction of stall cycles leading to a better performance.

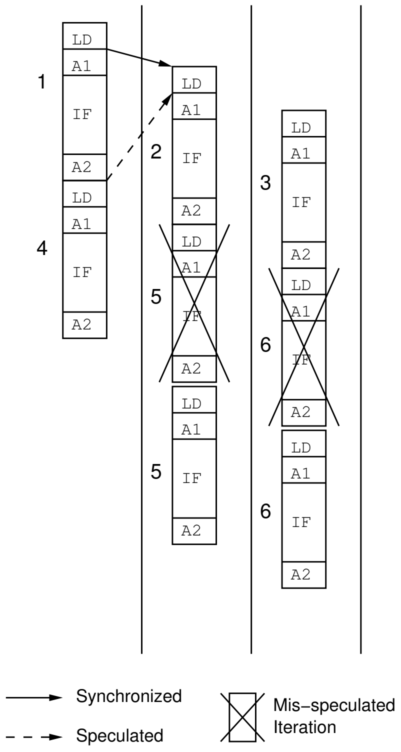
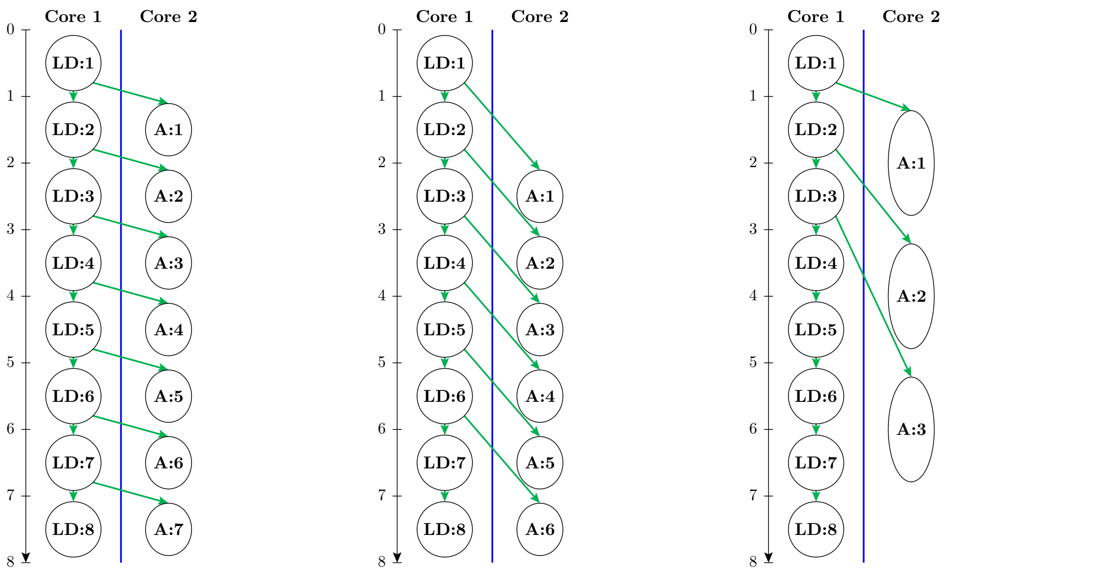


Figure 2.7: The execution schedule of the loop in Figure 2.6 parallelized by TLS.

Figure 2.7 shows a simplified execution schedule of a TLS parallelization of the loop in Figure 2.6. Each iteration of the loop is represented by a rectangular box with each of the statements demarcated. The self dependence on the LD statement is synchronized and the dependence from A2 to A1 is speculated. As long as the speculation is successful, the execution schedule is identical to that of DOACROSS. If a speculation is unsuccessful in an iteration, the execution of all later iterations are squashed and re-executed. In the example, the speculation of the dependence from iteration 4 to 5 turns out to be unsuccessful. This causes iteration 5 in thread 2 and iteration 6 in thread 3 to be squashed and re-executed after the detection of mis-speculation.

The TLS execution model requires hardware support to detect mis-speculations and recover from them. The cache coherence protocol is used to identify if a memory location is written to by an iteration *after* some later iteration has read from it. The updates from speculative iterations are buffered in private caches and written to shared caches or main memory only after it is guaranteed that the iteration cannot suffer any further mis-speculation. The hardware support required for TLS is described in detail by Steffan [62].

2.2.3 Pipelined Multi-threading



(a) Original schedule (b) Communication latency increased by 1 cycle (c) Execution time of A increased by 1 cycle

Figure 2.8: The execution schedule of the loop in Figure 2.3 parallelized by DSWP.

Pipelined multi-threading is another technique for parallelization in the presence of inter-iteration dependences. The first proposed PMT technique is DOPIPE [14]. DOPIPE is restricted to only loops with limited control flow. Decoupled software pipelining or DSWP [56, 44] is a more general PMT technique to extract pipelined parallelism from loops with arbitrary control flow. DSWP partitions the body of the loop into a sequence of pipeline stages. Each stage is then executed by a separate thread. The threads communicate either through special hardware structures such as synchronization array [56] or through memory. Figure 2.8(a) shows the DSWP execution schedule of the loop in Figure 2.3. The body of the loop is divided into a two stage pipeline: the first stage executes the pointer chasing load `LD` and the second stage executes the addition `A`. Parallelism is achieved by overlapping an earlier iteration of the second stage with a later iteration of the first stage.

Pipeline formation

Unlike other techniques discussed so far, DSWP partitions the body of the loop across different threads. While the details on how the code is partitioned across threads can be found elsewhere [44], a brief overview is given here to understand the performance implications of the code partitioning. The goal of the thread partitioning is to ensure that the threads form a pipeline and the work done by the different threads are balanced so that no thread ends up doing most of the work. The DSWP technique first constructs the program dependence graph(PDG [18]) of the loop and operates on the PDG. It then identifies the set of strongly connected graphs in the PDG. All operations in an SCC must be allocated to the same thread as otherwise there will be cyclic communication between the threads that execute the operations of the SCC. To ensure this, the PDG is transformed into another graph in which each SCC in the PDG are represented by a single node. The resulting graph is a DAG. Once the DAG is formed, the nodes of the DAG can be mapped into threads by assigning nodes to threads in a topological order. All these steps ensure that the resulting partition forms a pipeline. To address the problem of ensuring a balance among threads, a bin-packing-like heuristic is used; finding an optimal solution is NP complete [44].

Analytical model

Let $\Pi_1, \Pi_2 \ldots \Pi_k$ be the set of SCCs in the PDG of the loop. Let $L(\Pi_i)$ be the latency to execute the set of operations in the SCC Π_i. Let $\Pi_{T_{j,1}}, \Pi_{T_{j,2}} \ldots$ be the SCCs that are mapped to the j^{th} thread and let m be the total number of threads. Let $C_1, C_2 \ldots C_m$ be the set of communication operations that are inserted in each of the threads to communicate and synchronize the dependences between the threads. The latency of execution of the j^{th} thread is given by

$$L(T_j) = L(\Pi_{T_{j,1}} \cup \Pi_{T_{j,2}} \ldots \cup \Pi_{T_{j,k_j}} \cup C_j)$$

Assuming no variation in the execution latencies across loop iterations, the overall execution time of the parallelized loop is simply the execution time of the thread that takes the longest time:

$$\begin{aligned} L_{par} &= Max_j(L(T_j)) \\ &= Max_j(L(\Pi_{T_{j,1}} \cup \Pi_{T_{j,2}} \ldots \cup \Pi_{T_{j,k_j}} \cup C_j) \end{aligned}$$

If L_{seq} is the sequential execution time of the loop, the speedup obtained by applying DSWP is given by

$$speedup = \frac{L_{seq}}{Max_j(L(\Pi_{T_{j,1}} \cup \Pi_{T_{j,2}} \ldots \cup \Pi_{T_{j,k_j}} \cup C_j)} \tag{2.2}$$

Performance characteristics

Equation 2.2 helps to understand the factors that affect the performance of DSWP. The first observation is that DSWP is not affected by communication latency between cores in the asymptotic case. This naturally follows from the fact that the communication is always unidirectional in DSWP and it is pipelined. The only impact of communication latency is that it increases the "fill" cost of the pipeline which is significant only when the loop has a low iteration count. This is illustrated by Figure 2.8(b). In this execution schedule, the communication latency is increased by one more cycle. This increases the fill cycle in thread 2 by 1. However, after the first iteration is completed, in each cycle an iteration of the loop completes execution. Thus, the asymptotic execution latency is 1 cycle per iteration which is twice as fast as the single-threaded case.

While communication latency is not a factor in the expression for speedup, the latency of executing the communication operations C_i affects the speedup. If a value is communicated immediately after it is generated, the number of communication operations is proportional to the number of inter-thread dependences. However, strategies can be employed to group multiple communication operations together [54] so that the number of communication operations is proportional to the iteration count of the loop.

The factor that significantly affects the performance of DSWP is the execution latencies of the strongly connected components in the PDG. From the expression for $speedup$, it is clear that the speedup is limited by the execution latency of the thread that takes the longest time to execute. This is lower bounded by the execution latency of the largest SCC as the thread containing that SCC cannot run any faster than that SCC. Thus, if Π_{max} is the largest SCC, then the maximum speedup obtainable by PS-DSWP is $\frac{L_{seq}}{L(\Pi_{max})}$. Thus a fundamental difference between DSWP and the other techniques described earlier is that DSWP parallelization does not scale with the iteration count of the loop and hence the size of the input to the program.

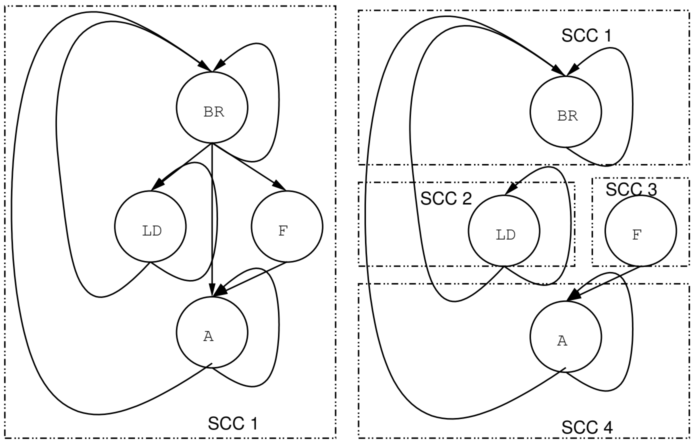
Application of speculation

Since the speedup obtainable by DSWP is fundamentally limited by the execution time of the largest SCC in the PDG, speculation can be used as a tool to break large SCCs. This allows the DSWP partitioning algorithm to form more balanced partitions with better performance characteristics. The speculative version of DSWP was first proposed by Vachharajani et al. [73].

Figure 2.9 illustrates how speculation can be used to enhance pipelined parallelism. Consider the program dependence graph for the loop in Figure 2.9a. The entire PDG is one single SCC and hence prevents DSWP from being applied to this loop. Speculation can be applied by observing that the loop exit branch `BR` is highly biased in one direction. In fact, during an invocation of the loop, the branch causes the loop to be exited only

```
while(ptr &&
   sum < MAX_SUM) { //BR
  val = foo(ptr->val)//F
  sum += val //A
  ptr = ptr->next //LD
}
```

(a) Loop with an infrequent dependence.



(b) Original PDG and SCCs (c) Speculative PDG and SCCs

Figure 2.9: This figure shows how speculation can enable the application of DSWP to a loop with an infrequent dependence.

during the last iteration and in the rest of the iterations it always results in control being transferred to the statement F. Hence, the control dependences originating from BR can be speculatively ignored. Figure 2.9(c) shows the resultant *speculative PDG*. Once the dependences originating from the branch BR are removed, the PDG decomposes into 4 different strongly connected components allowing DSWP to be applied.

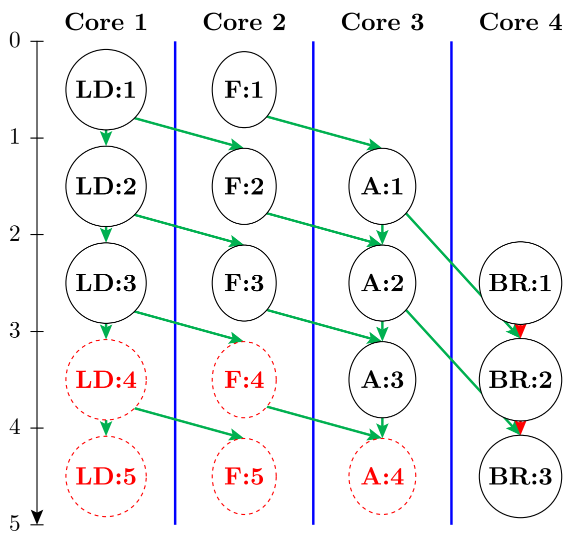


Figure 2.10: Execution timeline of the loop in Figure 2.9 parallelized by speculative DSWP.

Figure 2.10 shows a possible four stage speculative DSWP pipeline of the loop in Figure 2.9. The loop is assumed to have iterated for 3 iterations. The dashed circles denote statements that are mis-speculated. For instance, `LD:4` and `LD:5` are shown inside dashed circles, denoting the mis-speculation of the statement `LD` in the fourth and fifth iterations. A separate thread called *commit thread* detects mis-speculation and orchestrates the recovery of the correct state. The recovery involves undoing the effects of the mis-speculated statements and sequential re-execution of the mis-speculated code. To recover from the effects of speculation on memory state, a special form of transactional memory known as *multi-threaded transactions* [72] must be supported in the hardware.

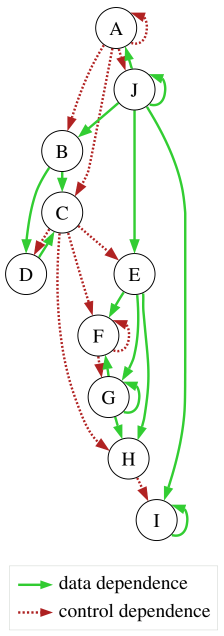
Chapter 3

Parallel-Stage Decoupled Software Pipelining

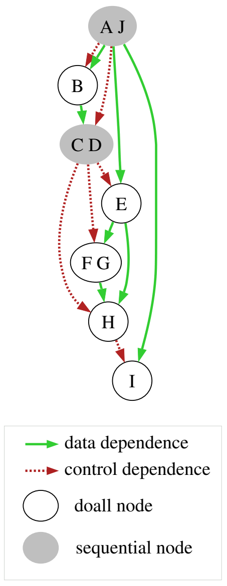
This chapter presents the parallel-stage decoupled software pipelining (PS-DSWP) technique, first motivating the technique with code examples and then describing the technique's details. Finally, the use of speculation to improve the performance gains of PS-DSWP is discussed.

```
        p = list;
        sum = 0;
A:      while (p != NULL) {
B:        id = p->id;
C:        if (!visited[id]) {
D:          visited[id] = true;
E:          q = p->inner_list;
F:          while (q != NULL && !q->flag)
G:            q = q->next;
H:          if (q != NULL)
I:            sum += p->value;
          }
J:        p = p->next;
        }
```

Figure 3.1: Motivating example for PS-DSWP.



(a) PDG



(b) DAG$_{SCC}$

Figure 3.2: PDG and DAG$_{SCC}$ for the loop in Figure 3.1.

3.1 Motivation

Parallelizing the loop in Figure 3.1 motivates the PS-DSWP technique. Consider the parallelization of this loop by DSWP. Figure 3.2 shows the PDG and the DAG_{SCC} for the loop in Figure 3.1. As discussed in the previous chapter, the performance of DSWP is limited by the execution latency of the largest strongly connected component. There are a total of 7 SCCs in this loop, each represented by a DAG_{SCC} node in Figure 3.2(b). Out of these 7 SCCs, the SCC FG represents the entire inner loop formed by the statements F and G. Assuming that the inner loop has a sufficiently high iteration count, the SCC FG is likely to be the largest SCC and the performance of DSWP is limited by its execution time.

A key observation about the loop in Figure 3.1 is that the execution of different *invocations* of the inner loop are independent. If DSWP is enhanced by allowing concurrent execution of these invocations, the performance of DSWP will no longer be bound by the total execution time of the inner loop. PS-DSWP enables such a concurrent execution by "replicating" the SCC FG so that multiple threads concurrently execute the code repre-

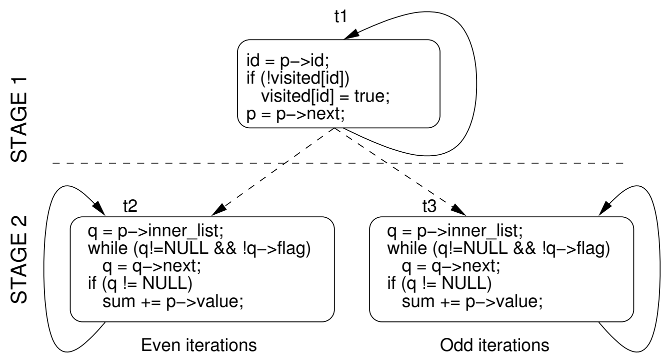
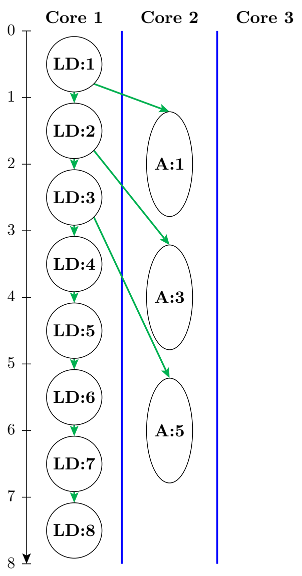


Figure 3.3: PS-DSWP applied to the loop in Figure 3.1.

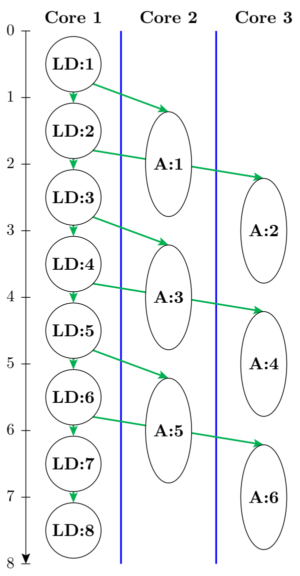
sented by this SCC in different iterations of the outer loop. Replicating a stage results in the extraction of data parallelism since all replicated copies of a stage execute the same code, but on different pieces of data. Any stage in the DSWP pipeline that is replicated is called a *parallel stage*. This replication is possible because this SCC is created by dependences carried[1] only by the inner loop, and not the outer loop that is parallelized by DSWP. In other words, only dependences carried by the loop being parallelized by DSWP prevent the formation of parallel stages.

In this example, there are dependences carried by the outer loop in the SCCs AJ, CD, and I. While the dependences in the first two of these SCCs cannot be ignored, the dependence in the third SCC (I) can be ignored by applying sum reduction [1], allowing the SCC I to be replicated. Thus, one possible PS-DSWP partition of the DAG$_{SCC}$ in Figure 3.2(b) is as follows: a first, sequential stage containing SCCs AJ, B, and CD, and a second, parallel stage containing the remaining SCCs. In this partition, the parallel stage can be replicated and concurrently executed in as many threads as available, with the performance limited only by the number of iterations of the outer loop and the slowest stage of the pipeline. Figure 3.3 sketches the code that PS-DSWP generates for the loop in Figure 3.1, with two threads executing the parallel stage. While not shown in this figure, the actual transformation generates code to communicate the control and data dependences appropriately, and to perform the sum reduction.

[1]A dependence is said to be carried by a loop if it is inter-iteration with respect to that loop.



(a) DSWP schedule



(b) PS-DSWP schedule

Figure 3.4: The execution schedule of the loop in Figure 2.3. Execution latency of `LD` is one cycle and `A` is two cycles.

Figure 3.4 revisits the loop in Figure 2.3 and uses it to contrast DSWP and PS-DSWP. For the purpose of this example, it is assumed that the list traversed by the loop is acyclic. The DSWP schedule for the loop is shown in Figure 3.4(a). DSWP is unable to make use of the third core as the loop cannot be partitioned into more than two stages. Since the increment of the two nodes in an acyclic list can be done in parallel, PS-DSWP can be used to replicate the second stage of the pipeline. The resulting PS-DSWP schedule is shown in Figure 3.4(b).

3.2 Communication Model

The threads created by PS-DSWP communicate values between themselves inside the loop body. For this purpose, PS-DSWP assumes the presence of a set of point-to-point communication queues between the threads. The interface to these queues consists of send and receive primitives. The send primitive produces the value in a register to a communication queue and the receive primitive consumes the value at the head of a communication queue into a register. The queues are addressed using a register relative address mode. A special register called *queue base* register (QB) and an immediate value are added together to get the actual queue number. This mode of addressing the queues allows the same copy of the code in a parallel stage to be shared by all the threads executing that stage, and yet be able to communicate with different threads. PS-DSWP does not rely on any specific implementation of the communication queues for its correctness. The queues could be implemented by dedicated hardware structures such as synchronization array [56] and scalar operand networks [67], or by using the memory subsystem [55].

3.3 PS-DSWP Transformation

Algorithm 1 PS-DSWP algorithm

```
        PS-DSWP (loop $L$)
(1)       $G \leftarrow$ build_dependence_graph$(L)$
(2)       $SCCs \leftarrow$ find_strongly_connected_components$(G)$
(3)       if $|SCCs|$ = 1 then return
(4)       $DAG_{SCC} \leftarrow$ coalesce_SCCs$(G, SCCs)$
(5)       $\mathcal{A} \leftarrow$ assign_threads$(DAG_{SCC}, L)$
(6)       if $|\mathcal{A}|$ = 1 then return
(7)       generate_code$(L, \mathcal{A})$
```

Algorithm 1 shows the pseudo-code for the PS-DSWP transformation. It takes a loop L as its input and parallelizes the loop. The rest of this subsection describes each step of the algorithm in detail, focusing on the extensions to the DSWP algorithm that enable the creation of parallel stages. The loop in Figure 3.1 is used as a running example to illustrate the steps of the algorithm.

3.3.1 Building the Program Dependence Graph

The Program Dependence Graph (PDG) [18] is used to represent the body of the loop L. The nodes of the PDG represent operations contained in the body of the loop. An edge $u \rightarrow v$ in the PDG indicates that the operation represented by v is dependent on the operation represented by u. A dependence arc can represent either a data dependence through a register,[2] a data dependence through memory, or a control dependence. For registers, only true dependences are represented in the PDG [44]. The dependence arcs in the PDG are annotated with a flag indicating whether the dependence is inter-iteration or not. Inter-iteration dependences are identified as follows:

- For data dependences through memory, if array dependence analysis [1] could be applied, it is used to determine if the dependences are inter-iteration dependences. Otherwise, a dependence is conservatively treated as an inter-iteration dependence.
- For data dependences through registers, data flow analysis is used to determine if they are loop-carried. Consider a dependence arc $n_1 \rightarrow n_2$. Let r be the register written by the operation corresponding to n_1. If the definition of r by n_1 reaches the loop header, and the use of r by n_2 is upwards-exposed at the loop header, only then the dependence is an inter-iteration dependence.
- For control dependences, a simple graph reachability check is used. If $n_1 \rightarrow n_2$ is a control dependence and all paths from n_1 to n_2 in the control-flow graph contain the loop backedge, then the control dependence is considered as an inter-iteration dependence.

Irrespective of the above tests, the dependences between operations that can be subjected to reduction transformations and the self-dependences involving induction variables are not considered to be inter-iteration, since suitable transformations can be applied to enable these operations to be executed in a parallel stage.

One limitation of using the PDG representation is that it forces procedures that are called within the loop to be treated as one indivisible unit, unless they are already inlined. As a consequence, if there is an inter-iteration dependence between two operations deep inside a called procedure, then it prevents the SCC containing the caller from being

[2]In this discussion, registers denote virtual registers, which are nothing but scalar variables whose addresses are never taken.

replicated. The use of system dependence graph (SDG) [25] overcomes this problem by exposing the operations inside procedures to the dependence graph representation.

3.3.2 Obtaining the DAG_{SCC}

Once the PDG or the SDG of the loop is formed, the strongly connected components (SCCs) in this dependence graph are then identified [66] and a directed acyclic graph of them, the DAG_{SCC}, is constructed. Each node of the DAG_{SCC} represents a strongly connected component in the original dependence graph. The DAG_{SCC} for the PDG in Figure 3.2(a) is shown in Figure 3.2(b). If there is only one node in the DAG_{SCC}, PS-DSWP cannot parallelize the loop. If none of the edges in a strongly connected component are labeled as loop-carried dependences, the corresponding node in the DAG_{SCC} is labeled as a doall node. All other nodes are labeled as sequential nodes. If a node is labeled doall, the operations belonging to the SCC corresponding to that node from two different iterations can be executed concurrently.

3.3.3 Thread Partitioning

Let $D = \{d_1, d_2, \ldots d_k\}$ be the set of nodes in the DAG_{SCC}. Let the number of target threads be denoted by n, and the set of threads be $T = \{t_1, t_2, \ldots t_n\}$. The thread partitioning algorithm takes D and T as input and produces the following as output:

- A partition $P = \{B_1, B_2 \ldots B_l\}$ of the nodes in the DAG_{SCC}. Each element of P corresponds to one of the l stages of the pipeline.
- An assignment $A = \{(B_1, T_1), (B_2, T_2) \ldots (B_l, T_l)\}$, which maps the blocks[3] in the partition P to subsets T_i of T. The T_is partition the thread set T.

[3]An element of a partition is called *block*. Thus, in this dissertation, a block refers to a set of nodes in the dependence graph. Block does not refer to a basic block unless explicitly mentioned as such.

To be valid, an assignment A obtained as above must respect the following property:

Property 1 (Valid Assignment)**.** *The assignment A is valid iff it satisfies the following conditions:*

1. *For $i \neq j$, if there is some dependence from B_i to B_j, then there are no blocks $B_{k_1}, B_{k_2} \ldots B_{k_n}$, where $k_1 = j$ and $k_n = i$, such that there is some dependence arc from every B_{k_l} to $B_{k_{l+1}}$. In other words, the dependence arcs between the blocks in the partition do not result in a cycle.*
2. *For every (B_i, T_i), with $|T_i| > 1$, the DAG_{SCC} nodes in B_i must be doall nodes, and none of the dependence arcs among the nodes in B_i is an inter-iteration dependence.*

The first condition ensures that the blocks B_i can be mapped to threads that form a pipeline. The second condition ensures that only doall nodes are present in a parallel stage and that there are no loop-carried dependences within a parallel stage. In addition to the above two conditions, a third condition is imposed in the implementation described in this work to simplify code generation: for every (B_i, T_i), with $|T_i| > 1$, $|T_i| = k$. In other words, every parallel stage in a loop is executed by the same number of threads.

An ideal solution to the thread partitioning problem is one that minimizes the overall execution time of the parallelized loop. However, the execution times of the different operations in the loop are not known *a priori* during compilation time. Further discussion on the problem modeling and different thread partitioning algorithms is postponed to Chapter 4.

3.3.4 Code Generation

After the partition and assignment are chosen, multi-threaded code is generated using an extension of the MTCG algorithm [44]. The algorithm described here assumes that the dependence graph is a PDG. The code generation algorithm for SDG is very similar to this with some additional changes to partition the procedures contained within the loop. The code generation extensions specific to the partitioning of SDG are discussed by Bridges [5].

Thread procedures

Let $P = \{B_1, B_2 \ldots B_l\}$ be the partition obtained by thread partitioning. The instructions in the blocks $B_2 \ldots B_l$ are removed from the original loop and placed into separate procedures F_2 to F_l. Placing the operations involves appropriately creating the required control flow in these procedures. This part of the code generation is identical to the original MTCG algorithm and hence is not described here in detail.

If B_i forms a parallel stage, then it is executed by multiple threads. All these threads execute the same procedure F_i. However, certain registers, such as loop induction variables, need to be initialized differently by each of the threads. Hence each F_i is passed a parameter that distinguishes the different threads that execute a given stage.

Inter-thread communication

Once the operations are placed in their respective procedures, dependences are communicated between the procedures. Based on the position in the CFG where the communication operations are inserted, inter-thread communication can be classified as follows:

Communication inside the loop: Consider two blocks B_i and B_{i+1} produced by the thread partitioning algorithm. Let P_{B_i} and $P_{B_{i+1}}$ be the PDG nodes contained in the two blocks. Consider the set of PDG edges $E = \{(u, v) \mid u \in P_{B_i}, v \in P_{B_{i+1}}\}$. E is the set of edges whose source node belongs to P_{B_i} and the destination node belongs to $P_{B_{i+1}}$. The dependences represented by E need to be communicated from the thread executing the operations in B_i to the thread executing the operations in B_{i+1}. The dependences communicated include both data and control dependences.

Register dependences are communicated by sending the value of the register immediately after the operation at the source of the dependence. In the receiving thread, this value is received at the corresponding program point. Memory dependences are synchronized using send/receive pairs. The communication primitives used for memory synchronization also act as memory barriers by implementing the release/acquire semantics.

Control dependences are communicated by replicating the relevant branches. Consider the control dependence $n_1 \rightarrow n_2$. The node n_1 must correspond to a conditional branch instruction. This dependence is communicated by replicating that branch in the thread containing n_2. The direction of execution of the replicated branch should always mirror the original branch. This requires communicating the branch direction to the other thread, which can be viewed as a data dependence communication. Since all the operations inside the loop are transitively control dependent on the loop exit branch, the loop branch is replicated in all the threads. This recreates the loop structure in all the threads.

Communication between sequential and parallel stages is handled differently. Consider a dependence $n_1 \rightarrow n_2$ from a sequential stage S to a parallel stage P. Let p be the number of threads that execute the parallel stage. Then, during the i^{th} iteration of the loop, the dependence is communicated to the $(i \ mod \ p)^{th}$ thread executing the parallel stage P. Initially, all such dependences are communicated as if the dependence exists between two sequential stages. Then the total number of communication queues used Q_{used} is computed. The total number of available queues Q_{max} is then divided into a set of Q_{used} contiguous queues. Each set of Q_{used} is called a *queue-set*. Let $t_0 \ldots t_{k-1}$ be the threads that execute the parallel stage P. Each t_i is assigned a unique queue-set and it uses only the queues in that queue set. The thread t_i initializes its `QB` register to $i \times Q_{used}$ before entering the loop. The sequential stage S sets the value of `QB` to ($Q_{used} \times j$) mod ($k \times Q_{used}$) at the beginning of the j^{th} iteration so that the values it produces are communicated to thread $t_{j \ mod \ k}$.

Figure 3.5 shows the communication of dependences for the loop in Figure 3.1. For the purpose of this example, a two stage partition of the DAG_{SCC} in Figure 3.2(b) is assumed. The first stage is a sequential stage with SCCs AJ, B and CD and the second stage a parallel stage with the rest of the nodes in the DAG_{SCC}. Thread t0 executes the sequential stage and threads t1 and t2 execute the parallel stage. Both t1 and t2 execute the same copy of the loop. The communications within the loop occur from t0 to t1 and t2, alternately. The communications between t0 and t1 are shown in dashed lines, and the communications

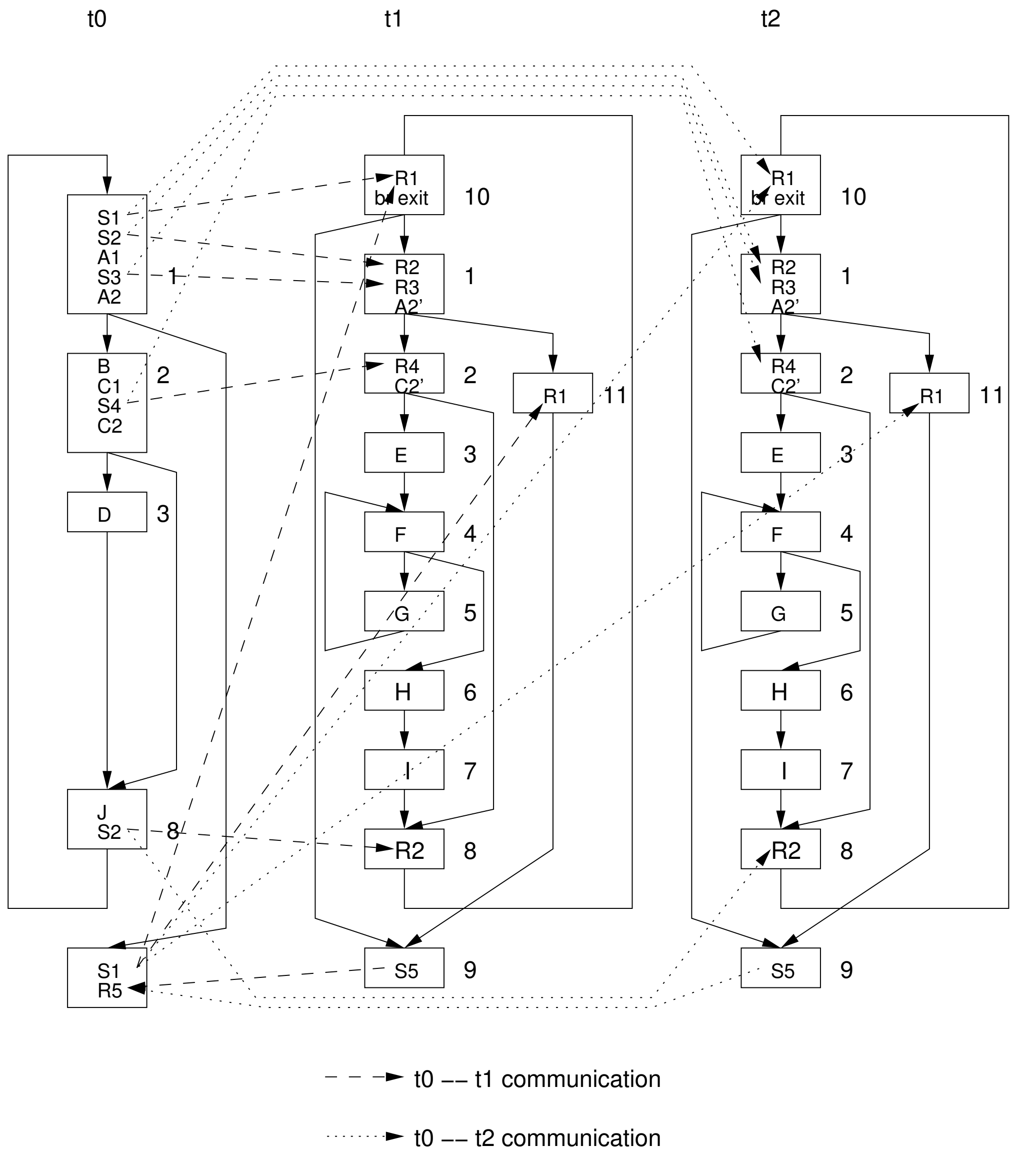


Figure 3.5: Inter-thread communication in PS-DSWP for the loop in Figure 3.1. A two stage partition of the DAG_{SCC} in Figure 3.2(b) is assumed with the parallel second stage executed by two threads. The (S1,R1) and (S3,R3) pairs communicate the loop exit condition, (S2,R2) pair communicates the variable `p`, (S4, R4) pair communicates the branch condition inside the loop and the (S5, R5) pair communicates the loop live-out `sum`.

between t0 and t2 are shown in dotted lines. The loop exit branch A is split into two statements A1 and A2, where A1 computes the branch condition and A2 is the actual branch. Thread t0 executes A1 and communicates the value to t1 and t2 by means of the send/receive pair (S3, R3). The branch A2 is replicated in the parallel stage as A2', thereby completing the communication of the control dependence. This creates the loop structure in the parallel stage. The branch C is similarly communicated to t1 and t2. The data dependence from J that defines the variable p is communicated using the send/receive pair (S2, R2).

Communication of live-ins: If a register defined outside the loop is used within the loop in the newly created threads, then the value of this register just before entering the loop needs to be communicated. Consider a procedure F_i that uses a value v defined outside the loop. If F_i executes a sequential stage, then the value v is communicated to F_i at the pre-header of the loop. If F_i executes a parallel stage, then the location of communication depends on whether the value is loop-invariant or not. If the variable containing v is loop-invariant, then the value is communicated at the pre-header. If the variable is not loop invariant, communicating the value in the pre-header will result in incorrect execution. To see this, consider the variable p in Figure 3.1, which is defined both outside and inside the loop. Let p be communicated to the parallel stage outside the loop at the pre-header. Since all threads executing the parallel stage execute the same thread procedure, all of them would be initialized with the value of p outside the original loop. Only the first thread executing the parallel stage — the one that executes the first iteration of the original loop — has to be initialized with this value. It is incorrect to initialize p with this value in the rest of the threads executing the parallel stage. For instance, in the second parallel thread, the variable p needs to be initialized with the value of p produced by the statement J in the first iteration of the loop, and not the value from outside the loop. Hence, if a variable is defined both inside and outside the loop, its value must be communicated at the loop header, instead of the pre-header.

In Figure 3.5, the communication of the live-in variable `p` is shown by the send/receive pair (S2, R2) in basic block 1. Note that `p` is also communicated within the loop in basic block 8. In this example, the (S2, R2) pair in basic block 8 can be removed since the R3 in block 1 redefines `p`.

Communication of live-outs: All the variables that are defined inside the sequential stages, and live out of the loop are communicated back to the main thread. In the parallel stages, for each register that is not control dependent on any branch within the loop, only the thread that executes the last iteration of the loop sends the value of the register. If the variable is an accumulator, all the threads send the value to the main thread where the values are added together. For conditionally defined variables and variables subjected to min/max reduction, all threads executing the parallel stage send both the variable and also the iteration count when the variable was last written. This iteration count acts as a timestamp and its use is is discussed in Section 3.3.4. The only live-out variable in the example loop is the accumulator `sum`, whose communication is shown by the (S5, R5) pair.

Loop termination

The loop exit branches are replicated in all thread procedures to satisfy control dependences. Exit branch replication enables the threads executing sequential stages to properly terminate the loop. However, replicating the exit branches is not sufficient to terminate the parallel stages, since only one of the threads assigned to a parallel stage executes the last iteration of the loop, and only that thread exits the loop by taking the original loop exit. Two different approaches are used to terminate the rest of the threads executing the parallel stages, depending on where a parallel stage is located in the pipeline. If a parallel stage is the first stage of the pipeline, it implies that the loop is counted. This follows from the fact that all operations are control dependent on all loop exit branches, and the loop exit is part of a non-trivial SCC in an uncounted loop. In the case of a counted loop, each thread executing the parallel stage counts the number of iterations they need to run and exit based on that. If a parallel stage is not the first stage, the first (sequential) thread, which is guaranteed to execute the loop exit branch, explicitly communicates loop termination information to the threads executing the parallel stage. The first sequential thread sends a `true` token at the loop header to the thread that is going to execute the current iteration. On loop termination, it sends a `false` token to all the threads. Since the parallel-stage thread that has exited the loop by taking the loop exit branch also receives a `false` token, it has to consume and discard that token after exiting the loop. The communication of this exit token is shown by the (S1, R1) pair in Figure 3.5. Basic block 10 in t1 and t2 has a branch that exits based on the value received by R1. This token is sent by t0 in basic block 1 within the loop, and basic block 9 outside the loop. Either t1 or t2 exits the loop by executing the original loop exit branch and consumes this token in basic block 11.

Loop induction variables

When a loop induction variable is assigned to a parallel stage, it needs to be suitably initialized at the beginning of each thread. Consider a basic induction variable of the form $i = i + k$. Let i_0 be the initial value of i before entering the loop. Let $0, 1, \ldots, p - 1$ be the thread identifiers of the threads executing the parallel stage that increments the induction variable. In the parallel stage, the variable i is initialized to $i_0 + id \times k$ at the loop's pre-header, where id is the thread identifier passed as a parameter to the thread procedure. Thus each thread executing the parallel stage initializes the variable with a different value. Inside the loop, the value of i is incremented every iteration by $p \times k$.

Loop live-outs

If a loop live-out is an accumulator, all threads executing the parallel stage send the values to the main thread at loop termination, and the main thread sums them up. For other live-outs that are not conditionally defined, only the thread that executes the last iteration of the

```
if(cost < mincost){
  mincost = cost;
  minnode = node;
}
```

(a) Original code

```
if(cost < mincost){
  ic = curr_iteration;
  mincost = cost;
  minnode = node;
}
```

(b) Code in parallel stages

```
receive(mincost1)
receive(minnode1)
receive(ic1);
receive(mincost2)
receive(minnode2)
receive(ic2);
mincost = mincost1;
minnode = minnode1;
if(mincost2 < mincost){
    mincost = mincost2;
    minnode = minnode2
}
else if(mincost2 == mincost){
    if(ic2 < ic1)
      minnode = minnode2;
}
```

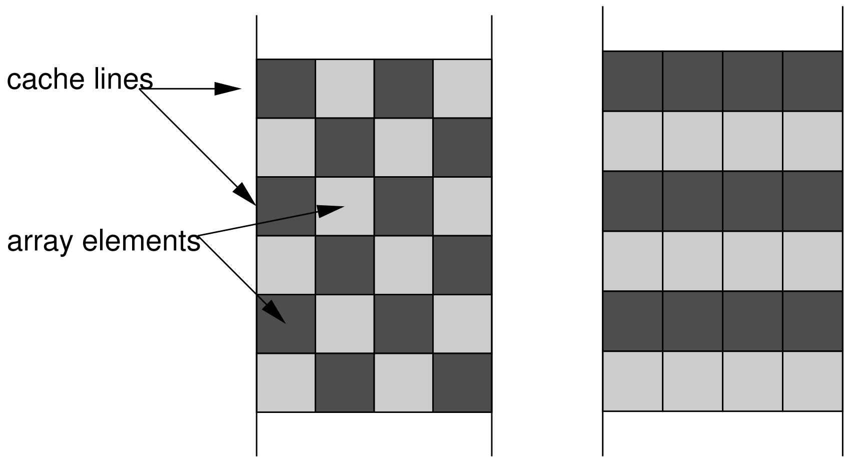
(c) Merging of the results

Figure 3.6: Merging of reduction variables defined in parallel stages using iteration count as the timestamp.

loop sends the value to the main thread. Conditionally defined live-outs pose an additional problem as shown by the code fragment in Figure 3.6(a). Assuming that the computation of `mincost` and `minnode` can be subjected to reduction, this code fragment can be part of a parallel stage. If that parallel stage is executed by two threads, one of them computes the minimum of the `cost` variable and the associated `node` in all even iterations, and the other thread computes the minimal in all odd iterations. To correctly obtain the value of `minnode`, it is essential to keep track of the iteration in each of the threads that finally defines the `mincost` variable. Figure 3.6(b) shows that a variable `ic` is assigned the current iteration count whenever `mincost` is assigned. Figure 3.6(c) shows how the main thread merges the two values. If `mincost` produced by one thread is less than the `mincost` produced by the other thread, the lesser value and the associated `minnode` is chosen. If the values are equal, then iteration count determines the correct value of `minnode`.

```
        int a[N], b[N], c[N];
A:      for(i = 1; i < N; i++){
B:        a[i] = a[i]*b[i];
C:        c[i] = c[i-1]+a[i];
        }
```

(a) Loop affected by false sharing



(b) Without chunking (c) With chunking

Figure 3.7: Memory access pattern with and without iteration chunking.

3.3.5 Optimization: Iteration Chunking

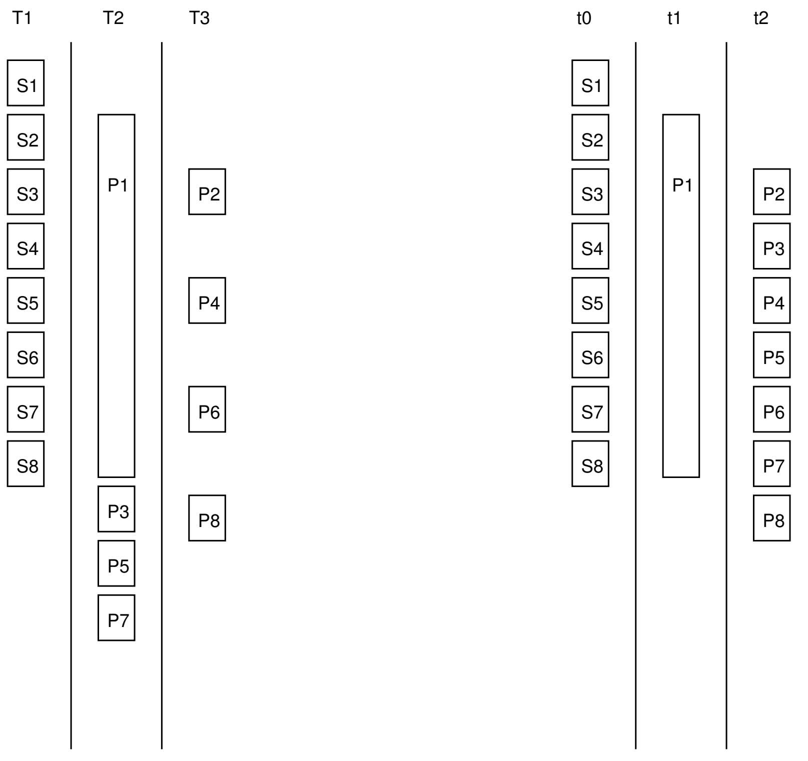
The PS-DSWP transformation causes the iterations of the parallel stage to be executed in a round-robin fashion by the threads assigned to that stage. One of the drawbacks of such a round-robin execution is that it could exacerbate the problem of *false sharing*, thereby negating the benefits of parallelization. False sharing can occur on multi-processors that implement an invalidation-based cache-coherence protocol when two or more different processors alternately write to *different* locations in the *same* cache line.

Consider the loop in Figure 3.7. PS-DSWP can parallelize this loop into a first parallel stage containing statements A and B executed by two threads followed by the sequential stage executing C. The first thread executing the parallel stage writes to array elements `a[1]`, `a[3]`, `a[5]` and so on, and the second thread writes to even elements of the array `a`. This is depicted in Figure 3.7(b). Each row in the figure represents a cache line, and each cell in a row represents an array element. For the sake of this discussion, assume that the cache-line size is 4 times the size of an element, and that the array is aligned to the cache-line boundary. The two different colors used in the cells represent the two different processors that execute the parallel stage, and hence write to the cache line. Since both threads write to the same cache line, the ownership of the cache lines alternate between the two processors. This increases the latency of every access to `a`. If this increase in latency is huge, it could negate the benefits of executing the pipeline stage in parallel.

In this example, false sharing can be eliminated by applying *iteration chunking*. With iteration chunking, each of the threads executing the parallel stage executes chunks of contiguous iterations, in a round-robin fashion. For the example in Figure 3.7(a), assuming that the two parallel threads execute chunks of 4 iterations, the writes to array `a` now exhibit a pattern shown in Figure 3.7(c). Each cache line is written by only one processor, and the two processors write to alternate cache lines, thereby eliminating false sharing. However, chunking could reduce the amount of parallelism. In the extreme case, the chunk size can equal the total number of iterations of the loop in a given invocation, which is equivalent to treating that stage as a sequential stage. The chunk size can be specified as a parameter by the user to the PS-DSWP transformation and defaults to 1. Chunking requires some additional changes to PS-DSWP code generation in the areas of induction-variable handling and the manipulation of the `QB` register.

3.3.6 Optimization: Dynamic Assignment

In PS-DSWP, the i^{th} iteration of a parallel stage is executed by thread $t_{i \ mod \ k}$ where k is the replication factor of that parallel stage. Such a static assignment of iterations to threads can result in severe performance degradation due to variability in iteration execution times. This is illustrated in Figure 3.8(a). The figure shows the execution schedule of a loop after PS-DSWP transformation. There are two stages in the pipeline: a sequential stage followed by a parallel stage which has a replication factor of two. Consider a case where the first iteration of the parallel stage takes much longer than the rest of the iterations. As



(a) Static Assignment (b) Dynamic Assignment

Figure 3.8: Comparison of execution schedules with static and dynamic assignment of iterations to threads.

a consequence, thread t1 takes much longer to complete executing its share of iterations than thread t2, while t2 is idle for many cycles between useful work. These idle cycles can be eliminated by *dynamically* assigning parallel-stage iterations to the threads that execute that stage. Figure 3.8(b) shows a dynamic assignment where thread t1 executes just the first iteration of the parallel stage and t2 executes the rest of the iterations, thereby reducing the total run-time.

Two different implementations of dynamic assignment are discussed in this section. Both these implementations assume that the first stage is not a parallel stage as that implies a fixed partition of iterations to threads. The first implementation follows a work-queue model where the threads executing the parallel stage themselves decide the mapping. Let P be a parallel stage and S be the sequential stage that immediately precedes it in the pipeline. At the beginning of each iteration, the thread executing stage S writes the queue-set used in that iteration to the head of a shared work queue accessed by P. The work queue is accessed by all threads executing P and is guarded by a single lock. The thread that locks the queue consumes the value from the head of the queue and uses it to set the `QB` register. For this implementation of dynamic assignment to work, the communication model has to be more general than the model described in Section 3.2. The queues in a queue-set have to be accessible to multiple processor cores that execute the threads associated with P. Memory based implementations of the communication queues as proposed by Rangan *et al.* [55] can provide such a functionality. Another drawback of this scheme is that during each iteration, many threads contend for a single lock which could degrade the performance.

The second implementation does not require changes to the communication model and retains the one-to-one mapping from threads to queue-sets. Let S_1 be the first stage of the pipeline, which is a sequential stage. In this implementation, S_1 decides the mapping from iterations to threads. Let i be the current iteration and k be the number of threads assigned to each parallel stage. Under static assignment, iteration i gets mapped to thread t_j, where $j = (i \ mod \ k)$. Let t_r be the candidate thread for this iteration, where $r \neq j$. The thread executing S_1 determines the candidate based on processor load or some other related metric that can be obtained from hardware performance counters. Dynamic assignment is achieved by treating the current iteration as if it is iteration l, where $l \ mod \ k = r$, so that it gets assigned to thread t_r. To realize this, S_1 inserts e empty loop iterations into the pipeline, where $e = (r - j) \ mod \ k$. An empty iteration is inserted to a pipeline stage by issuing an `EARLY_EXIT` at the beginning of that stage. When a thread receives that token, it skips that loop iteration by jumping to the end of the iteration. When an empty iteration is inserted, threads executing the sequential stages, including S_1, increment the `QB` register so that it points to the next queue-set. After inserting e empty iterations, the `QB` register in the sequential stages would point to the queue-set used by t_r. At this point S_1 executes the original iteration i normally.

As an example, consider a two stage pipeline with a first sequential stage and a second parallel stage with a replication factor of 3. Let PT_0, PT_1 and PT_2 be the threads that execute the parallel stage. Iterations numbered 0 to 4 are assumed to be assigned to the threads in a round robin fashion similar to static assignment. When the sequential stage enters iteration 5, it decides which parallel stage is best suited to execute that iteration. Based on some heuristic, it decides to assign iteration 5 to PT_1. It then sends `EARLY_EXIT` tokens to PT_2 and PT_0, thereby inserting two empty iterations. As a result, it can treat the original iteration 5 as if it is iteration 7 and gets assigned to PT_1.

3.4 Speculative PS-DSWP

Like other non-speculative parallelization techniques, the performance potential of PS-DSWP is limited by the conservative nature of the underlying static analyses. In particular, the conservative results produced by the memory dependence analysis are often a crucial limiting factor. Even if the analyses are accurate, they cannot ignore the dependences that infrequently manifest and those dependences that are highly input dependent. Speculation can be used in those situations to improve the effectiveness of PS-DSWP. The application of speculation to PS-DSWP is along the lines of speculative-DSWP [72, 73].

Speculative PS-DSWP transformation involves the following steps:

1. Perform loop-aware memory distance profiling and annotate the program with the profile information.

2. Construct the PDG (or the SDG) for the loop to be parallelized.

3. Form the speculative PDG by speculating dependences in the original PDG.

4. Construct the DAG_{SCC} and apply a PS-DSWP thread partition algorithm that generates a valid assignment.

5. Compute the set of speculations to retain based on the partitioning.

6. Apply the necessary speculations on the original single-threaded code and insert code for mis-speculation detection.

7. Apply the PS-DSWP code generation algorithm on the speculated single-threaded code.

8. Generate code for mis-speculation recovery.

A detailed description of these steps can be found in the Ph.D dissertation of Vachharajani [72]. The rest of this section describes loop-aware profiling and some details of mis-speculation recovery that are specific to speculative PS-DSWP but not relevant to speculative DSWP.

3.4.1 Loop Aware Memory Profiling

The application of speculation to PS-DSWP is helpful in two ways. Speculation can cause large SCCs to be broken down into smaller ones. This increases the likelihood of a more balanced partitioning of the loop into pipeline stages. This benefit is the primary motivation for speculative DSWP and does not depend on the presence of parallel stages. The second benefit of speculation is specific to PS-DSWP. Speculation can cause a node originally labeled sequential to be labeled doall. This is possible when an inter-iteration dependence is speculatively treated as intra-iteration. Recall that the classification of intra- and inter-iteration dependences is possible using static analysis only for memory dependences involving affine array accesses. All other memory dependences are conservatively assumed to be inter-iteration dependences. Instead of such a conservative classification, the loop aware memory profiling (LAMP) described in this section can be used to speculatively identify intra-iteration dependences, thereby potentially causing many DAG_{SCC} nodes to be treated as doall nodes.

Profile information

Just like other profilers, LAMP gives runtime dependence information between store/load pairs. Unlike other profilers, this dependence information between store/load pairs is not just a count indicating the number of times the store and the load alias. Instead, LAMP outputs a set of triples LP = $\{$ (H_l, $dist$, $count$) $\}$, where H_l is a loop header, $dist$ is dependence distance and $count$ is profile count. A triple (H_l, $dist$, $count$) is to be interpreted as follows: with respect to a loop whose header is given by H_l, the number of times the `load` reads a value written by the store that executed $dist$ iterations ago is $count$. As a concrete example, the presence of a tuple (H_{loop1}, 1, 10) in LP indicates that both the load and store are inside some loop $loop1$ and with respect to that loop, load has read the value written to by the store in the previous iteration 10 times. Note that the same pair of operations may also have an intra-iteration dependence if LP contains a triple (H_{loop1}, 0, $count_{intra}$) as well as a dependence with respect to a different loop $loop2$ if LP contains a triple (H_{loop2}, $dist$, $count$).

The number of inter-iteration dependences between stores and loads is obtained by adding all values of $count$ in triples of the form (H_{loop}, $dist$, $count$), where $loop$ is the loop being parallelized by PS-DSWP and $dist > 0$. If this value is below a threshold, the dependence can be speculated as an intra-iteration dependence.

Profiler implementation

The program to be profiled is instrumented with calls to the LAMP library. First, unique identifiers are assigned to all memory operations such as loads, stores and external library calls as well as all the loops in the program. Then all memory operations are instrumented with calls to the LAMP API. These calls take the identifier of the memory operation, the address accessed by the memory operation, and the size of the load or store. The preheader and the loop exits are instrumented to indicate the start and end of loop invocations. The loop headers are instrumented to indicate the beginning of a new iteration of a loop.

The LAMP library maintains several data structures to collect loop-aware profile information. The data structures and how they are used are described below:

Timestamp A global timestamp counter that is incremented whenever a LAMP API is called.

Loop nest stack The loop nesting information is maintained as a stack. Whenever the `LAMP_invocation_start` API call is made, the identifier of the loop and the current time stamp are pushed to the stack and when the `LAMP_invocation_end` call is made the top element of the stack is popped out.

Loop iteration queues For each loop in the loop nest stack, a circular queue is maintained. Whenever a `LAMP_iteration_start` API call is made, the current time stamp is inserted to the tail of the circular queue corresponding to the current loop (i.e the top element of the loop nest stack).

Memory writer map A map is maintained between each memory location and the store operation that last wrote to that location. Whenever a `LAMP_store` API call is made, a map is created between the locations that are written by the store and the tuple (id, current_time_stamp).

LAMP arcs map This is the main data structure that contains the final profile information. This is a map between (store_id, load_id, loop) triples and a list of (iteration_distance, count) tuples. This map is updated whenever a `LAMP_load` call is made. The location that is read by the load is looked up in the the memory writer map to get the (store_id, time_stamp) tuple. The timestamps of the entries in the loop nest stack are scanned from top to bottom to find the innermost loop that encloses both the load and the store. The loop iteration queue for this loop is scanned to find the iteration distance based on the iteration in which the store happened and the current iteration. If $dist$ is the iteration distance, then the list in the LAMP arcs map corresponding to the triple (store_id, load_id, this_loop) is scanned to see if there is a tuple for this iteration distance, and if so, the corresponding *count* is incremented. If no such tuple is found, a new (dist, 1) tuple is added to the list. If the dist is non-zero (i.e the dependence is an inter-iteration dependence), the process is continued for outer loops in the loop nest. If the iteration distance is 0, then the scanning of the entries in the loop nest stack stops since the iteration distance with respect to any outer loop will also be 0. When the program terminates, the contents of the LAMP arcs map are dumped to a file which is later used to annotate the program

The length of the circular queues determines the maximum dependence-distance that can be tracked by LAMP. If l is the length of each circular queue, any dependence with distance greater than l cannot be distinguished from a dependence with distance l and hence are merged together. For the purpose of speculative PS-DSWP, it is sufficient to distinguish dependences with distance 0 (intra-iteration) and greater than 0 (inter-iteration). Hence the circular queue can be replaced by a scalar that stores the time stamp of the last iteration of the loop.

Among the different data structures, the loop related structures have negligible memory usage. The size of LAMP arcs map in practice turns out to be small, since the set of dependences that manifest in practice are small [38]. The size of the memory writer map is proportional to the working set of the application. Its size can be reduced by keeping information at word level instead of byte level. Various performance optimization techniques including caching and paging have ensured that profile times are of the order of a few minutes for applications in the SPECInt 2000 benchmark suite.

3.4.2 Mis-speculation Detection and Recovery

Speculative DSWP uses a hybrid software-hardware approach to mis-speculation detection and recovery. The hardware support is in the form of multi-threaded transactions (MTX) which is only used to manage speculative updates to the memory, with software taking

```
op1: ld r1 = M[r2]
 op1': ld r1' = M[r2]
 op3: br r1 == r1', L
 //Mis-speculation
L:
...
op2: st M[r3] = r4
```

Figure 3.9: Application of memory alias speculation in speculative PS-DSWP.

care of the rest of the mis-speculation detection and recovery. Each stage contains code to detect violation of any speculated dependence in that stage. It then communicates that information to a separate thread called the *commit thread* that orchestrates the recovery process. The commit thread gathers the set of live-in registers from the individual stages at the beginning of each iteration. When it receives mis-speculation information from a thread, it restores the memory state to the state prior to the mis-speculation, re-executes the mis-speculated iteration sequentially, and sends back the correct set of live-in registers to all the threads. The hardware support and the steps involved in mis-speculation detection and recovery are discussed in detail by Vachharajani [72].

The integration of PS-DSWP with speculative DSWP is almost seamless requiring very minor changes to the two component techniques. Because of the presence of parallel stages, the commit thread maintains multiple copies of the live-in registers that need to be restored in each thread executing a parallel stage after recovery from mis-speculation. It manipulates the queue base register to receive iteration live-in values from each thread in a round-robin fashion. The queue base register in each thread itself is saved and restored after mis-speculation to allow for correct communication among the different threads after mis-speculation.

The addition of memory alias speculation forces the last stage of a PS-DSWP partition to be a sequential stage because of the mis-speculation detection mechanism used in speculative DSWP. In speculative DSWP, all speculations are first applied to single-threaded code before performing the partitioning. The application of memory speculation leads to the insertion of new operations that must be appropriately allocated to a pipeline stage.

In Figure 3.9, the dependence between the store op2 and the load op1 needs to be speculated. In the single-threaded code, the load op1 is first cloned to create another load operation op1′. Then, the original load value r1 and the cloned loaded value r1′ are compared. Then, during partitioning, the dependence between op2 and op1 is ignored in the PDG but a dependence is created between op2 and op1′. Furthermore, the branch op3 is speculated to be always taken and control speculation is applied to the branch. If the branch is not taken, it implies that r1 and r1′ are not equal. Since the dependence between op2 and op1′ is enforced, r1 must have the incorrect value, implying the misspeculation of op1.

If the dependence between op2 and op1 is loop-carried, then the dependence between op2 and op1′ is also loop-carried. Hence, op1′ has to be placed in a sequential stage and the stage containing op1′ must come after the original store and load. Thus, due to the use of alias speculation in speculative PS-DSWP, the last stage of the pipeline must be a sequential stage.

3.5 Related Work

Davies [14] proposed an extension to DOPIPE that applies DOALL parallelization to a stage if the stage comprises only an inner loop. The technique is also not general enough to be applicable to loops with arbitrary control flow. PS-DSWP is applicable to loops with arbitrary control flow and parallel stages are not limited to inner DOALL loops. Takabatake et al. [65] proposed the *Do-sandglass* parallelization based on the observation that a loop can be divided into tasks with inter-iteration dependences and tasks that do not have inter-iteration dependences and they can be executed as a pipeline. However, they do not propose any automatic transformation to achieve the same.

When a loop to which DOALL cannot be applied contains some statements that do not have any inter-iteration dependences, loop distribution [1] can be applied to isolate those statements into separate loops. Then, those loops can be subjected to DOALL parallelization and the rest of the original loop can be executed sequentially. This approach results in computational resources being unused when the sequential portions execute whereas PS-DSWP overlaps the execution of the sequential and DOALL portions to extract pipelined parallelism. Furthermore, loop distribution can be applied only to counted loops.

Some of the advantages of PS-DSWP can be obtained by applying loop unrolling followed by DSWP. If a loop is unrolled twice, an SCC in the original loop body that does not contain an inter-iteration dependence with respect to the loop being parallelized is also replicated twice and forms two separate SCCs in the unrolled loop. Hence a DSWP partition of the unrolled loop could execute the two SCCs in two different threads. This unroll-and-DSWP approach has some significant drawbacks when compared to PS-DSWP. The first big disadvantage is the increase in code size due to unrolling, especially when applied to outer loops. To gain the same benefits of applying PS-DSWP inter-procedurally, it is not just enough to unroll the outer loops but also recursively inline the functions in the loop. This causes significant code growth for large unroll factors. Even if the code growth of the final binary can be tolerated, this code growth affects the size of PDG and other data structures in memory, thereby increasing cost of analyses and the transformation. The second major drawback is that this does not allow for dynamic assignment of iterations to threads described in Section 3.3.6. There are also other limitations such as the fact that the replication factor of a parallel stage has to be determined earlier since unroll factor determines the replication factor, whereas in PS-DSWP it is decided after the formation of the PDG.

Chapter 4

Thread Partitioning

This chapter discusses the PS-DSWP thread partitioning problem in detail. The goal of thread partitioning is to obtain a valid thread assignment that minimizes the execution time of the parallelized loop. However, the actual execution time depends on the input set and no compile time algorithm can minimize the execution time for all possible inputs. In this chapter, the problem is simplified and modeled using metrics available at compile time. The hardness of obtaining an optimal solution under this model is then shown, followed by a discussion on three different heuristic solutions. This chapter presents empirical data comparing the three different heuristics and based on the empirical data, a heuristic called *doall-and-pipeline* is found to outperform others. The purpose of this empirical analysis is to identify the heuristic that is likely to have a good *average* performance on a large number of loops, so that it can be used by a compiler implementing PS-DSWP.

4.1 Problem Modeling and Challenges

To model the solution for the thread partitioning problem, the following simplifying assumptions are made:

- The execution time of each pipeline stage remains unchanged across different iterations of the loop.
- The target processor has a single-issue in-order architecture with each operation taking 1 clock cycle to execute.

Under these assumptions, the execution time of the original operations in a stage S_i (which is the same as the operations in block B_i of the partition P) can be approximated by W_i, the profile weight of those operations. The profile weight W_i is simply the sum of dynamic execution frequencies of all the operations in that stage. In addition, after the PS-DSWP transformation, each stage executes some additional send/receive operations, whose cost is denoted by C_i. Finally the number of iterations executed by the loop is assumed to be much larger than the number of threads n so that all threads executing a parallel stage performs roughly the same amount of work.

Based on the above approximation, an expression for the execution time of the parallelized loop can be derived. The execution time of a sequential stage is just the sum of its W_i and C_i. If a stage S_i is a parallel stage, its work is equally divided across $|T_i|$ threads and hence its execution time is $\frac{(W_i+C_i)}{|T_i|}$. Furthermore, the latency of a pipeline is determined by the latency of its slowest stage. Thus, the execution time of the parallelized loop is given by

$$E = MAX_i(\frac{W_i + C_i}{|T_i|}). \tag{4.1}$$

For this simplified model of the problem, the optimal solution is one that minimizes $MAX_i(\frac{(W_i+C_i)}{|T_i|})$. Minimizing E can be shown to be NP-hard. The proof is by a reduction from the *bin packing* problem, a known NP-complete problem [20].

Theorem 1. *Thread partitioning is NP-hard under the simplified execution model.*

Proof. The input to the bin packing problem is a sequence of numbers S, a number of bins b, and the capacity c of each bin. The problem is to decide if there exists an assignment of the numbers in S to the b bins such that the sum of the numbers in each bin is at most c.

The reduction to the thread partitioning problem instance is as follows. For each number S_i in S, a DAG_{SCC} node is created whose profile weight is set to S_i. All nodes are labeled sequential. No edges are created in the DAG_{SCC} leading to the absence of any communication between the nodes. The number of threads n is set to b, the number of bins.

The solution to this thread partitioning instance returns an assignment $A = \{(B_1, T_1), (B_2, T_2) \ldots (B_l, T_l)\}$, whose execution time E is given by $MAX_i(\frac{(W_i+C_i)}{|T_i|})$. Since no stage can be a parallel stage and there is no communication between the stages, the number of stages l becomes the number of bins b and E reduces to $MAX_i(W_i)$. Consider a bin packing where all numbers corresponding to the DAG_{SCC} nodes in stage S_i are packed into a unique bin. If the solution to the thread partitioning problem is optimal, then the solution to bin packing can be obtained by merely checking if $MAX_i(W_i) < c$.

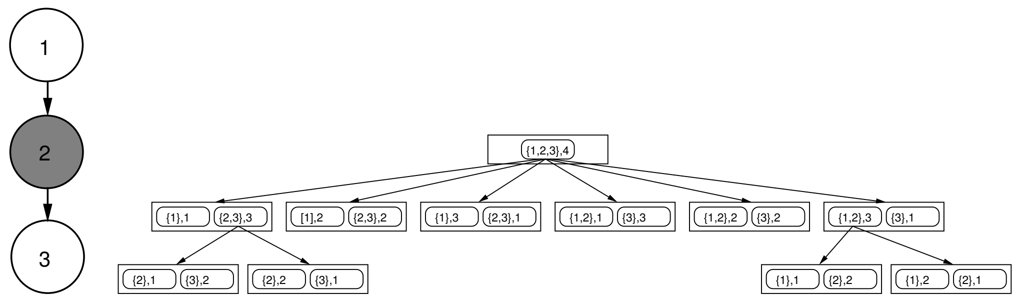
$\square$

Since the size of the DAG_{SCC} runs into thousands of nodes when constructed from a system dependence graph, an exact solution is impractical. Hence, the rest of this chapter presents three different *heuristic* algorithms for the thread partitioning problem.

4.2 Evaluating the Heuristics

Since all the three algorithms are heuristic solutions, empirical results are used to compare them and find the algorithm that is better on the average. The algorithms are evaluated on a total of 251 loops from 29 benchmarks that include the SPEC2000 and SPEC2006 integer benchmarks [60] written in C, the Mediabench benchmarks [29] and a few Unix applications. Only those loops that account for at least 10% of the total execution time in these benchmarks are used for the comparison.

The system dependence graphs of these loops are first constructed. Then, the following criteria are used to speculate infrequent dependences in the SDG:



(a) DAG_{SCC} (b) Search tree

Figure 4.1: Search tree traversed by the exhaustive search algorithm.

- If a branch is highly biased ($> 95\%$), then the control dependences from the branch to all the operations it controls are speculated.
- Dependences between operations that are infrequent with respect to the execution frequency of the outer loop are speculated.
- Memory dependences that never manifested during the profiling run are speculated.

The DAG_{SCC} is constructed from the speculated SDG and the partitioning algorithms are applied on the DAG_{SCC}. For each loop, an estimated speedup metric is computed based on the parallel execution time in Equation 4.1 and its sequential execution time. This speedup estimate does not take into account the cost of mis-speculation detection and recovery.

4.3 Algorithm: Exhaustive Search

The first approach is an exhaustive search algorithm that is based on the assumption that while the PS-DSWP partitioning problem is NP-hard, certain instances may still be tractable if the edges in the DAG_{SCC} highly constrain the set of legal solutions. For such instances, the algorithm searches all possible legal partitions to find the best partition. For other instances, a time budget is used to stop the search-space prematurely and return the best solution found so far.

The search of solution space is illustrated in Figure 4.1. Figure 4.1(a) shows a simple straight-line DAG with three nodes and Figure 4.1(b) shows the corresponding search tree. Each node of the search tree corresponds to a dependence graph G that needs to be partitioned and the number of threads available to partition the graph. The root of the tree contains the entire dependence graph, represented by the set of nodes $\{1, 2, 3\}$ and the number of threads given to the partitioner, which is 4 in this case. All nodes, other than the root node, consist of two sub-nodes. This denotes a cut of the dependence graph associated with its parent into two parts. For instance, the leftmost child of the root node corresponds to cutting the graph into a subgraph $G1$ consisting of just node 1, and a subgraph $G2$ consisting of nodes $\{2, 3\}$. The two subgraphs are represented by the two sub-nodes in the left child of the root. The sub-nodes also contain the information that one thread is available to partition the subgraph $G1$ and three threads to partition $G2$. A pipeline for G with 4 threads can be obtained by first obtaining a pipeline of $G1$ with one thread, a pipeline of $G2$ with three threads and then by appending the second pipeline to the first. By cutting G in different ways and by assigning different number of threads to the cuts, multiple feasible solutions for G are obtained. The best among them — the one that maximizes the estimated performance — is then chosen. The solution for each of the parts in a given cut can be similarly obtained recursively. Thus, each of the *sub-nodes* has a set of children in the search tree. A sub-node has no children if the corresponding subgraph is a single-node subgraph in which case it cannot be further partitioned. A node is also a leaf node if it has less than three threads as otherwise its partition cannot contain a parallel stage.

The pseudocode for the SEARCH routine is given in Figure 2. It contains several optimizations to prune the search tree. It first has a series of `if` statements that terminate a branch of the search tree. First, it checks if the partition for the given graph for the given number of threads has been already computed and returns it from a table. The search also terminates when there is just one thread available, in which case there is nothing left to partition. If the entire graph is a DOALL graph, then again no partitioning of the subgraph

Algorithm 2 SEARCH

```
Input: $DAG\ G$
Input: #Threads $n$

if Table[$G$, $n$] is not empty then
    return Table[$G$, $n$]
else if n == 1 then
    return [($G$, 1)]
else if $G$ is a $DOALL$ graph then
    return [($G$, $n$)]
else if n == 2 then
    return DSWP_PARTITION($G$, $n$)
else if Time $>$ TimeBudget then
    return DSWP_PARTITION($G$, $n$)
else if not HAS_PROFITABLE_PARALLEL_STAGE($G$, $n$) then
    return DSWP_PARTITION($G$, $n$)
else
    BEST_WT = $\infty$
    $CUTS$ = PROFITABLE_LEGAL_CUTS($G$)
    for each cut $(G1, G2)$ in $CUTS$ do
        for m = 1 to $n - 1$ do
            $P_1$ = SEARCH($G1$, $m$)
            $P_2$ = SEARCH($G2$, $n - m$)
            $WT$ = ESTIMATE_WEIGHT($G1$, $m$, $G2$, $n - m$)
            if $WT > BEST\_WT$ then
                $BEST\_WT = WT$
                $BEST\_CUT = APPEND(P_1, P_2)$
            end if
        end for
        if Time $>$ TimeBudget then
            break
        end if
    end for
    Table[$G$, $n$] = $BEST\_CUT$
    return $BEST\_CUT$
end if
```

is required. Each of the next three conditions, if satisfied, causes the routine to obtain a partition without any parallel stages by calling the DSWP_PARTITION routine. The first of these is when there are only two threads available, which rules out the possibility of a partition with at least one sequential stage and a parallel stage. If the time taken to search the tree so far has already exceeded the time budget, a DSWP pipeline of the subgraph is returned. Finally, the DSWP_PARTITION routine is invoked if it determines that the profile weight of the nodes that can be assigned to a parallel stage is low.

The `else` part of the outermost `if` statement contains the recursive portion of the SEARCH routine. First, the set of legal cuts of the graph is obtained. A cut is simply a partition of the nodes of the graph into top and bottom parts and it is legal if there is no arc from the bottom part to the top part. Given a legal cut, the top and bottom parts could be independently partitioned and fused together. From the set of legal cuts, cuts that are deemed unprofitable are removed.

A cut could be unprofitable in two ways. A cut could be *lopsided* if the weight of one part is much much lower than the other leading to a poor load balance. A cut C_1 of G could also be unprofitable if it is subsumed by another cut C_2. This happens when exploring C_2 is likely to get a partition of G that is at least as good as any partition obtained by exploring C_1. As an example, consider a graph G that contains a sequential node n_1 of weight 0.3, and two doall nodes n_2 and n_3 of weights 0.1 and 0.15. A cut with n_2 alone in one part and the rest of the nodes in the other part is subsumed by a cut that has n_2 and n_3 in the same part since the part containing n_1 is the bottleneck.

Figure 4.2 show the relative speedup when the exhaustive search is used. The time budget was set to 1200 seconds. Only 7% of the loops considered for partitioning exceeded the time budget. The partitions obtained by exhaustive search result in a 240% geometric mean estimated-speedup over single-threaded execution under the execution model described in Section 4.1. On an average, the algorithm takes 211.8 seconds to partition a loop.

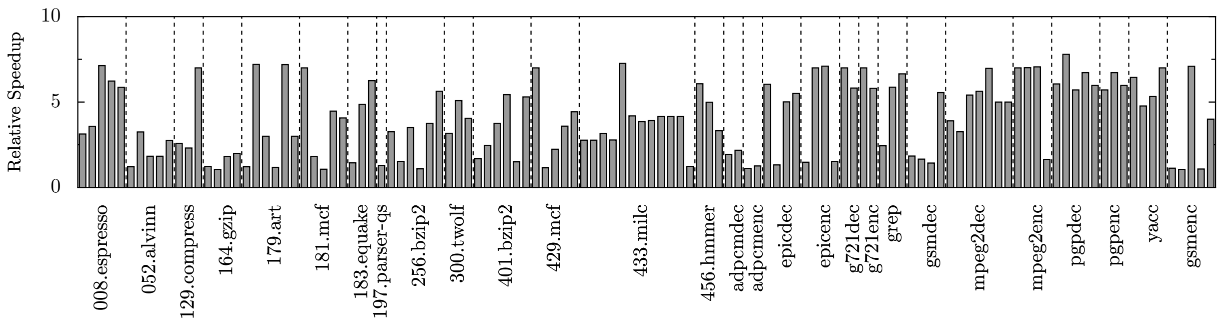
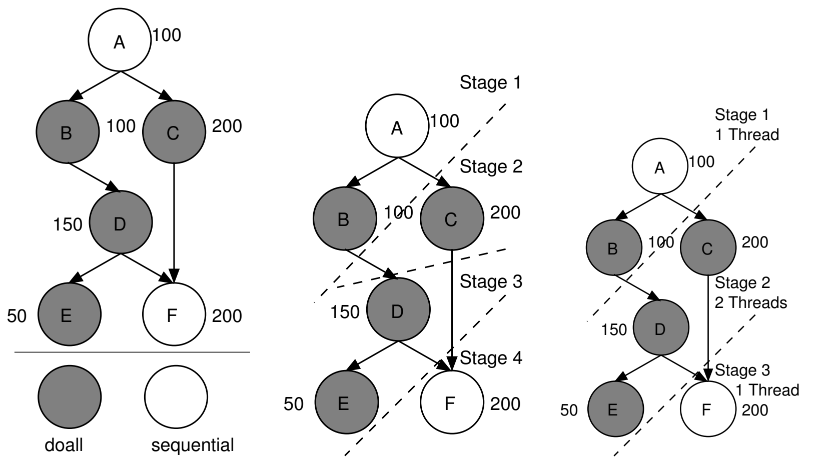


Figure 4.2: Estimated speedup of exhaustive search. Estimated geometric mean speedup is 3.4.



(a) DAG_{SCC} (b) Pipeline (c) Pipeline with parallel stages

Figure 4.3: Example illustrating the pipeline-and-merge heuristic.

4.4 Algorithm: Pipeline and Merge

The second approach to thread partitioning is to obtain a DSWP pipeline and then perform a post-processing pass that merges consecutive parallel stages to obtain larger parallel stages. This algorithm is referred to as *pipeline-and-merge*. The DSWP pipeline is obtained by using the pipeline formation algorithm originally proposed by Ottoni et al. [44]. The algorithm attempts to balance the weights of the stages in the pipeline.

After forming the DSWP pipeline, a post-processing phase repeatedly merges stages to reduce the depth of the pipeline and to maximize the work done by parallel stages. This phase merges only two parallel stages since if a parallel stage is merged with a sequential

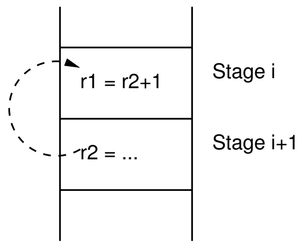


Figure 4.4: Parallel stages containing operations with loop carried dependence.

stage, the resulting stage can only be sequential. Furthermore, only consecutive stages can be merged to prevent cyclical dependences between pipeline stages. The presence of loop-carried dependences can prevent two consecutive parallel stages from being merged. Consider two consecutive stages S_i and S_{i+1} in Figure 4.4. S_i contains an operation that uses a register $r2$ defined in stage S_{i+1}. Note that there can not be any dependence from S_i to S_{i+1} as that violates the requirement that there be no cyclic dependences between pipeline stages. If S_i and S_{i+1} are merged together, then the resulting stage cannot be a parallel stage because it contains a loop-carried dependence. If there is no loop-carried dependence between operations in S_i and S_{i+1}, the stages are merged together. The number of threads allocated to the resulting stage equals the sum of threads allocated to those two stages. It is always profitable to merge two parallel stages instead of keeping them as two separate stages. Let stage S_i have a weight of W_i and assigned n_i threads and stage S_{i+1} have a weight of W_{i+1} with n_{i+1} threads. The weight of the merged stage is at most W_i + W_{i+1} since communication operations needed to communicate between S_i and S_{i+1} are no longer needed and no additional communication operations are inserted in the pipeline. Clearly, $\frac{W_i+W_{i+1}}{n_i+n_{i+1}}$ is less than $\frac{W_i}{n_i} + \frac{W_{i+1}}{n_{i+1}}$.

Figure 4.3 illustrates this heuristic. The original DAG_{SCC} is shown in Figure 4.3(a). The shaded nodes are doall nodes and the numbers adjacent to the nodes are the node weights. Figure 4.3(b) shows a pipeline with four stages. Excluding the communication cost, the pipeline is perfectly balanced. Stages 2 and 3 are parallel stages and the edges connecting the nodes in those stages are not loop carried. Hence they can be merged to form a single large parallel stage that is executed by two threads as shown in Figure 4.3(c).

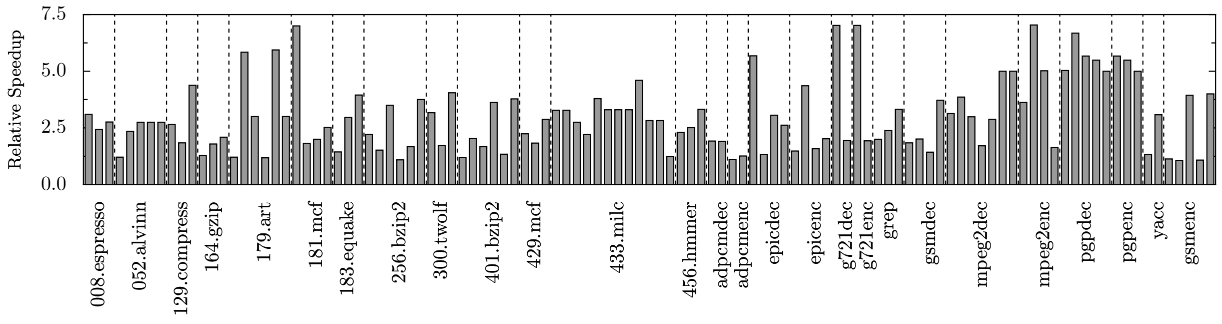


Figure 4.5: Estimated speedup of pipeline-and-merge heuristic over single-threaded execution. Geometric mean of the estimated speedup is 2.65.

Figure 4.5 shows the estimated speedup of various loops using the pipeline and merge heuristic. For the loops that are considered, the pipeline and merge heuristic results in a 165% geometric mean estimated-speedup over single-threaded execution. This compares poorly against the speedup obtained by the exhaustive search algorithm. However, the pipeline-and-merge algorithm takes only 4.8 seconds to partition a loop.

Randomization

One drawback with the pipeline-and-merge approach is that a stage formed by the DSWP partitioning algorithm may comprise of mostly doall nodes, but a few sequential nodes in that stage may prevent it from being merged with another stage to form a larger parallel stage. Randomization can be used to overcome this problem. There are two ways in which the use of randomization can aid in bringing more operations to parallel stages:

- DSWP partitioning algorithm traverses the DAG_{SCC} in topological order to form pipeline stages. There are many legal topological orderings of a DAG and DSWP explores only one of them. Randomization helps in exploring other valid topological orderings.
- While the DSWP partitioning tries to exactly divide the work among pipeline stages, allowing some slack in load balancing could result in more parallel stages being merged together in the post-processing pass.

The randomized version of the pipeline-and-merge algorithm conducts a large number of random trials, where each trial corresponds to choosing one particular partitioning of the DAG_{SCC}. While the weight of each stage in a trial may be unbalanced, the algorithm attempts to achieve load balancing based on the expectation of the weights of the stages over a large number of trials.

Algorithm 3 shows how a pipeline stage is formed. For each stage, the algorithm generates a Gaussian random number R. This number determines the weight of the operations in that pipeline stage, normalized with respect to the total weight of all the operations in the DAG_{SCC}. If the stages are perfectly balanced, this number has to be $\frac{1}{n} + \delta$, where n is the number of threads or pipeline stages and δ is the estimated communication cost per stage. By generating R with a mean μ of $\frac{1}{n} + \delta$, the algorithm ensures that over a large number of trials, it generates balanced pipeline stages. The standard deviation σ of the Gaussian distribution is set to $\frac{\mu}{6}$. Since more than 99% of the trials under a normal distribution lie in the range $\mu \pm 3\sigma$, the value of σ chosen by the algorithm ensures that most of the numbers fall in the range $\frac{\mu}{2} - \frac{3\mu}{2}$.

This algorithm also maintains a work list and picks a random node from the work list to add to the current pipeline stage S_i. If the weight of a stage exceeds the Gaussian random number generated for that stage, then it stops adding operations to S_i and starts the new stage S_{i+1}, unless the current value of i is n, in which case all the remaining operations are added to S_n. In the post-processing phase, consecutive parallel stages that do not have any inter-iteration dependences are merged and the threads are added up. Since the number of pipeline stages created by the randomized partitioner could be less than the available number of threads, some unused threads may remain. These are proportionately allocated to the parallel stages based on the weight of the stages. The algorithm repeats this procedure a large number of times and retains the best partition.

Figure 4.6 shows the results for the randomized version of the pipeline and merge heuristic. The algorithm uses a total of 1000 trials and chooses the best of those 1000 trials.

Algorithm 3 Randomized Pipeline Formation

```
Input: $DAG_{SCC}$
Input: $n$, number of threads

$\mu = \frac{1}{n} + \delta$
$\sigma = \frac{\mu}{6}$
$R = N(\mu, \sigma)$
**while** worklist not empty **do**
   $r$ = random element from worklist
   $S_i = S_i \cup r$
   $W_i$ = weight($S_i$)/weight($DAG_{SCC}$)
   **for** each successor $s$ of $r$ **do**
      **if** all predecessors of $s$ are processed **then**
         add $s$ to worklist
      **end if**
   **end for**
   **if** $W_i > R$ and $i <= n$ **then**
      i = i+1
      $S_i = \emptyset$
      $R = N(\mu, \sigma)$
   **end if**
**end while**
```

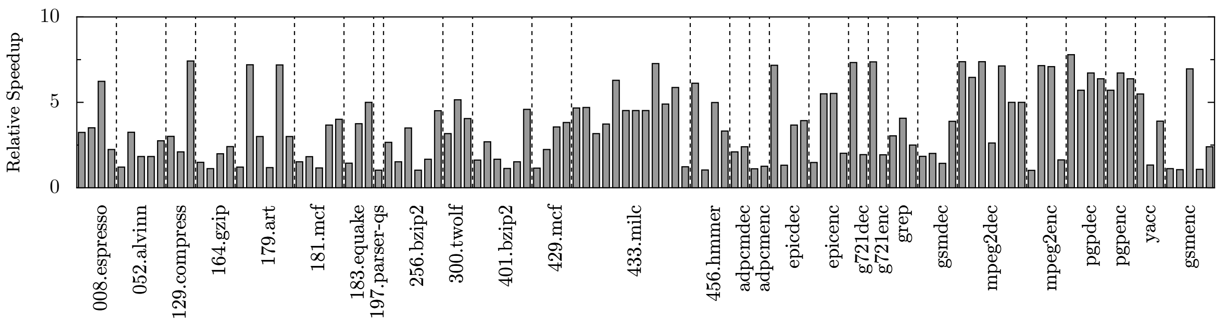


Figure 4.6: Estimated speedup of randomized pipeline-and-merge heuristic over single-threaded execution. Number of trials is 1000 and the estimated geometric mean speedup is 2.99.

The geometric mean estimated-speedup obtained by this algorithm is 199%. While, this speedup is significantly better than the deterministic version, it is still less than the speedup obtained by the exhaustive search algorithm. Compared to the deterministic version, the freedom given by randomization in slicing the graph in different ways results in large parallel stages to be extracted. However, the average time to partition a loop has increased from 4.8 seconds in the case of the deterministic version to 95 seconds for the randomized version. Increasing the number of trials does not significantly improve the performance, but significantly increases the partitioning time.

4.5 Algorithm: Doall and Pipeline

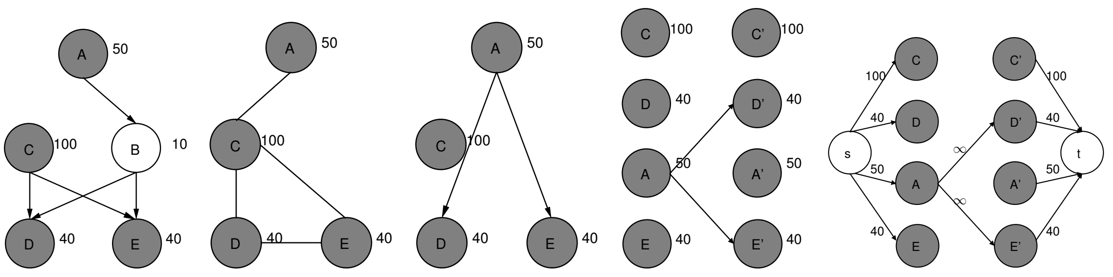
The doall-and-pipeline algorithm is based on the hypothesis that executing more code as part of parallel stages is more important than getting a balanced pipeline. Executing most of the code in a parallel stage reduces the depth of the pipeline, thereby reducing the communication overhead. The doall-and-pipeline heuristic is shown in Algorithm 4. It first finds the largest subgraph in the DAG_{SCC} that can be assigned to a single parallel stage. Then it estimates if it is profitable to assign the nodes of that subgraph to a parallel stage. It is profitable to assign a set of operations to a parallel stage only if the execution time of those operations is significant enough that at least two threads can be assigned to execute that parallel stage. If there is no profitable parallel stage, then it just invokes the DSWP pipeline algorithm. Otherwise, it divides the rest of the DAG_{SCC} into two subgraphs: a predecessor graph P from which dependence edges reach the parallel stage and a successor graph S that is fed by the nodes of the parallel stage. The rest of the nodes that do not have any edges incident on the parallel stage are added to either the predecessor part or the successor part based on a load balancing heuristic. Then, it assigns pt threads to the parallel stage, where pt ranges from 2 to t, the total number of threads available. For each assignment of pt, the algorithm is recursively applied to the two subgraphs P and S, which are allotted all possible combinations of the remaining $t - pt$ threads. The combination with the best speedup estimate is chosen.

Algorithm 4 DOALL_AND_PIPELINE

```
Input: $DAG_{SCC}$ $G$
Input: #Threads n
**if** $G$ is a $DOALL$ graph **then**
   return [($G$, $n$)]
**end if**
$PS$=LARGEST_PARALLEL_STAGE($G$, $n$)
ps_graph=SUBGRAPH($G$, $PS$)
**if** NOT_PROFITABLE(PS) **then**
   return DSWP_PARTITION($G$, $n$)
**else**
   pred_graph = FIND_PRED($G$, $PS$)
   succ_graph = FIND_SUCC($G$, $PS$)
   best_speedup = 0
   **for** pt = MIN_THREADS(ps_graph) to n **do**
      **for** st1 = MIN_THREADS(pred_graph) to n **do**
         st2 = n - (pt + st1)
         part1 = DOALL_AND_PIPELINE(pred_graph, st1)
         part2 = DOALL_AND_PIPELINE(ps_graph, pt)
         part3 = DOALL_AND_PIPELINE(succ_graph, st2)
         speedup = ESTIMATE_SPEEDUP(part1, part2, part3)
         **if** speedup $>$ best_speedup **then**
            best_speedup = speedup
            best_part = APPEND(part1, part2, part3)
         **end if**
      **end for**
   **end for**
   return best_part
**end if**
```

Largest Parallel Stage in a DAG_{SCC}

The main component of the doall-and-pipeline algorithm is an optimal algorithm to compute the subgraph that can be assigned to a parallel stage and has the maximum profile weight among all such graphs. In other words, the algorithm finds the subset PS of the nodes of DAG_{SCC} that satisfies the following conditions:



(a) DAG_{SCC} (b) Mergeability graph (c) NM (d) B_{NM} (e) F_{NM}

Figure 4.7: Example illustrating the steps involved in obtaining the largest parallel stage from a DAG_{SCC}.

- All nodes in PS are doall nodes.
- It is legal to place any pair of nodes of PS in the same parallel stage.
- There is no other subset PS' of DAG_{SCC} nodes satisfying the above two conditions such that $W(PS') > W(PS)$.

Figure 5 outlines the algorithm to compute such PS. The rest of this section explains the algorithm in detail. The algorithm is illustrated using the example in Figure 4.7. Figure 4.7(a) shows the DAG_{SCC} that is to be partitioned. Node B is the sole sequential node and the rest of the nodes, shaded in gray, are all doall nodes.

Algorithm 5 Largest parallel stage

```
Input: $DAG_{SCC}$

Construct the non-mergeability graph $NM$
Construct the bipartite graph $B_{NM}$ from $NM$
Construct the flow graph $F_{NM}$ from $B_{NM}$
$MC$ = MINCUT($F_{NM}$)
Obtain the largest parallel stage from $MC$
```

Non-mergeability graph

The information on the pairs of doall nodes in a DAG_{SCC} that can be merged together can be represented using a *mergeability graph*. Figure 4.7(b) gives the mergeability graph for the DAG_{SCC}, which contains only the doall nodes of DAG_{SCC}. Mergeability is not a transitive relation. Hence, in any parallel stage, the nodes in that stage must be pairwise mergeable. In other words, the set of nodes in any parallel stage forms a clique in the mergeability graph. Hence, finding the largest parallel stage involves solving the maximum clique problem which is NP complete. However, the structure of the *complement* of the mergeability graph — the non-mergeability graph — can be used to solve the problem optimally in polynomial time. Finding the maximum clique in a graph is equivalent to finding the maximum independent set in its complement graph, the non-mergeability graph. Finding the maximum independent set is NP complete for a general graph, but has a polynomial time solution on a directed acyclic graph [2].

The nodes of the non-mergeability graph NM are again the doall nodes of DAG_{SCC}, but an edge between two nodes indicates that they cannot be merged and assigned to the same parallel stage. The edges can be assigned directions based on the partial order of the nodes in the DAG_{SCC}. This follows from the fact that if two nodes n_1 and n_2 in the DAG_{SCC} are not comparable, then they can be merged since doing so cannot create a cycle. Conversely, if n_1 and n_2 cannot belong to the same parallel stage, then they must be comparable and hence either there is a directed path from n_1 to n_2 or from n_2 to n_1 in DAG_{SCC}. Thus, the non-mergeable graph is a transitive closure of the DAG_{SCC} after removing edges between any two nodes that can be merged. Figure 4.7(c) shows the non-mergeability graph for the DAG_{SCC} in Figure 4.7(a). Node C can be merged with any of the other doall nodes and hence there is no edge incident on it in NM. Node A cannot be merged with nodes D and E and in the DAG_{SCC}, there is a directed path from A to D and from A to E. Hence NM contains the edges $A \rightarrow D$ and $A \rightarrow E$

Maximum independent set

Given the non-mergeability graph, the goal is to find the weighted maximum independent set. A weighted maximum independent set of a graph is a set of nodes I such that

1. No two elements of I are connected by an edge.

2. There is no other I' that meets condition 1 such that $W(I') > W(I)$.

An algorithm to compute the maximum weighted independent set on a DAG was proposed by Berenguer *et al.* [2]. A brief overview of that algorithm is presented below and illustrated with an example. Further details and the proof of correctness of the algorithm can be found in Berenguer *et al.* [2].

The first step of the algorithm involves constructing a bipartite graph B_{NM} from the non-mergeability graph NM. Let $NM = (V, E)$. Then, $B_{NM} = (L \cup R, E')$, where

- $L = \{v_i^l | v_i \in V\}$ and $R = \{v_i^r | v_i \in V\}$,

- $E' = \{(v_i^l, v_j^r) | v_i^l \in L, v_j^r \in R \quad s.t \quad (v_i, v_j) \in E\}$

In other words, L and R are both copies of V and if there is an arc $v_i \rightarrow v_j$ in NM, an edge is added from the copy of v_i in L to the copy of v_i in R. The weights of the nodes in the original graph are preserved in the two copies. Figure 4.7(d) shows the bipartite graph for the running example. For each node n in NM, B_{NM} has two nodes n and n'. Since NM has the arcs $A \rightarrow D$ and $A \rightarrow E$, B_{NM} has $A \rightarrow D'$ and $A \rightarrow E'$.

The next step is to construct the flow graph F_{NM} from the bipartite graph B_{NM}. Formally, the flow graph $F_{NM} = (V'', E'')$ is specified by

- $V'' = L \cup R \cup \{s\} \cup \{t\}$

- $E'' = E \cup \{(s, l) | l \in L\} \cup \{(r, t) | r \in R\}$

- $C(e) = \infty, \forall e \in E'$

- $C(e) = W(l), \forall e = (s, l) \quad s.t \quad l \in L$

- $C(e) = W(r), \forall e = (r, t) \quad s.t \quad r \in R$

F_{NM} has two nodes in addition to the nodes in B_{NM}: a source node s and a sink node t. Edges are added from the source node to each of the nodes in L and from each of the nodes in R to the sink node t. Capacities are assigned to the edges in F_{NM}. For all the edges in F_{NM} that are also in B_{NM}, the edge capacities are set to infinity. For an edge connected to the source node or the sink node, the edge capacity is set to the weight of the other node incident on the edge. The flowgraph for the running example is shown in Figure 4.7(e). The next step is to apply a maximum flow algorithm on F_{NM} to get the min-cut MC. Since the edges connecting the nodes in the left and right parts of the bipartite graph have a capacity of ∞, those edges are never a part of the min-cut. Thus the edges in the min-cut must have the source or the sink node as one of the end points and a node from B_{NM} as the other end point. Let MCN be the set of nodes from B_{NM} that are incident on the edges of the min-cut. Berenguer *et al.* [2] show that MCN cannot contain both a v_i^l and v_i^r. In other words, both the left and the right copies of the same node in V cannot be contained in MCN. Furthermore, $V - MCN$ gives the maximum weighted independent set in NM. For the example graph, MC contains just the edge (s, A) and so $MCN = \{A\}$ and the maximum weighted independent set is $\{C, D, E\}$ which constitutes the largest parallel stage.

Maximum flow implementation

The algorithm uses the Edmonds-Karp algorithm [16] for maximum flow computation. This algorithm has a complexity of $O(|V||E|^2)$ since there are $O(|V||E|)$ augmenting paths in the worst-case and the worst case complexity of finding an augmenting path is $O(|E|)$. For the flow graph F_{NM}, there are at most $O(|V|)$ augmenting paths, resulting in a worst case complexity of $O(|V||E|)$. Even then, the running time is prohibitively expensive when the flow graph has thousands of nodes. The running time can be further reduced by making use of the following observation. In F_{NM}, any augmenting path must have a prefix $s \rightarrow l \rightarrow r$, where $l \in L$ and $r \in R$. So, the algorithm pre-computes all such valid path prefixes $s \rightarrow l \rightarrow r$ and maintains a prefix list PL. Whenever an augmenting path has to

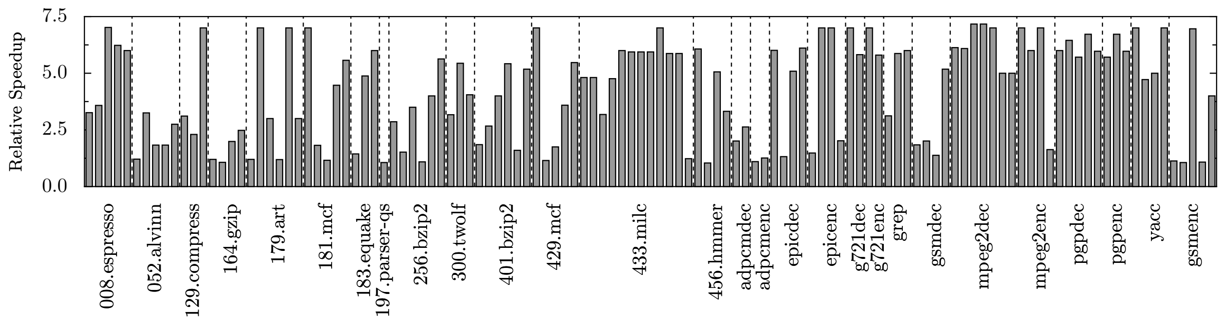


Figure 4.8: Estimated speedup of doall-and-pipeline heuristic relative to single-threaded execution. Estimated geometric mean speedup is 3.57.

be found, the algorithm starts the BFS from r for every $s \to l \to r$ in PL, instead of the source s. Since all $r \in R$ have an edge to t, the first step of the BFS results in the formation of the path $s \to l \to r \to t$. If the edges on this path are not saturated, an augmenting path has been found in constant time. Once an augmenting path is computed, the prefix list is updated to remove those prefixes that contain a saturated edge.

Evaluation

Figure 4.8 shows the estimated speedup when the doall-and-pipeline heuristic is used for thread partitioning. The geometric mean of the estimated speedup over single-threaded execution is 257%, which is an improvement over the other heuristics proposed in this chapter. Furthermore, this algorithm takes an average of just 20.7 seconds to partition a loop, which is more than ten times faster than the next best performing heuristic, the exhaustive search heuristic. The combination of a good average expected speedup and a low average processing time makes doall-and-pipeline a good candidate for thread partitioning in PS-DSWP.

Chapter 5

Speculative Parallel Iteration Chunk Execution

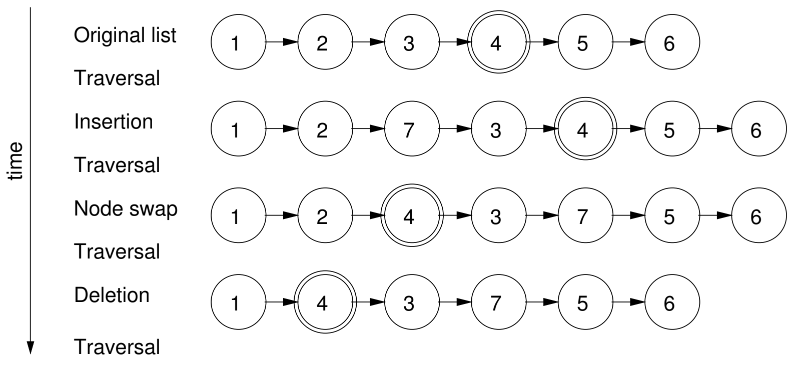
This chapter presents speculative parallel iteration chunk execution (Spice), a parallelization transformation that uses a new approach to value speculation to obtain thread level parallelism. The value speculation used in Spice is discussed first using a motivating example. This is followed by a detailed description of the implementation of this value speculation and the parallelization transformation.

5.1 Value Speculation and Thread Level Parallelism

Thread level speculation, a technique discussed in Chapter 2, speculates infrequently manifested dependences to concurrently execute loop iterations. The most common form of speculation used in TLS is memory alias speculation. In this form of speculation, a load in a given iteration is assumed not to access the same location as a store in some earlier iteration. Memory alias speculation works well as long as the conflict between loads and stores in different iterations are infrequent. If a memory dependence manifests frequently, alias speculation suffers from high mis-speculation rates leading to poor performance. Hence

```
1  c = cm->next_cl;
2  while(c != NULL){
3    int w = c->pick_weight;
4    if (w < wm) {
5      wm = w;
6      cm = c;
7    }
8    c = c->next_cl;
9  }
```

(a) Traversal code



(b) Modifications to linked list

Figure 5.1: A list traversal example that motivates the value prediction used in Spice.

TLS systems typically synchronize store-load pairs that conflict frequently. The drawback of synchronization is a reduction in the available parallelism. An alternative approach to synchronization is to apply *value prediction* [30, 31] and speculatively execute the future iterations with the predicted values.

Several TLS techniques [10, 32, 37, 64] have proposed the use of value prediction to minimize synchronization when alias speculation is not beneficial. TLS techniques with value prediction typically use some value predictor originally proposed to improve instruction level parallelism (ILP). Spice is based on a new value prediction technique with the goal of breaking inter-iteration dependences in a TLS system. The value predictor in Spice is based on two insights into value prediction in the context of thread level parallelism.

The first insight is that, *to extract thread level parallelism, it is sufficient to predict a small subset of the values produced by a static operation.* To see this, consider the loop in Figure 5.1(a) from the benchmark `otter` that traverses a linked list. For the sake of this discussion, it is assumed that the `if` condition in line 4 is rarely true. In that case, a

TLS system can speculate that the read of `wm` in the next iteration does not conflict with the write of `wm` in the current iteration. It can then apply value prediction to predict the value of `c` at the beginning of the next iteration and use the predicted value of `c` to run that iteration simultaneously with the current iteration. Alternatively, it can also speculatively parallelize the loop by only predicting the value of `c` every tenth iteration, instead of every iteration. In that case, the TLS system can speculatively execute *chunks of 10 iterations* in parallel. If the predictions are highly accurate, then it is sufficient to predict only as many values as the number of speculative threads. Predicting more values does not increase the amount of TLP. This is in contrast to the use of value prediction to improve instruction level parallelism. ILP value prediction techniques predict values of a long latency operation to speculatively execute the dependent operations. For every dynamic instance of that operation that is not predicted, the dependent operations stall in the pipeline, thereby reducing the ILP.

The second key insight is that *the probability of predicting that an operation will produce a particular value* some time *in the future is higher than predicting that that value will be produced at a* specific time *in the future.* Predicting that a value will appear some time in the future may be sufficient for the purposes of extracting TLP. To give an analogy, the likelihood that the Dow Jones index will be a particular value X exactly a year from now is much lower than the likelihood that it will be X *some day* within the next 2 years. To give a more concrete example, consider Figure 5.1(b) which shows the linked list and how it gets modified and accessed over time. Consider a simple predictor that predicts that node 4 is present in the list. In this example, the prediction is always true, even though the relative position of node 4 from the head of the list changes over time. Thus, such a predictor is likely to have a higher prediction rate than a predictor that predicts that node 4 is the fourth element from the head of the list.

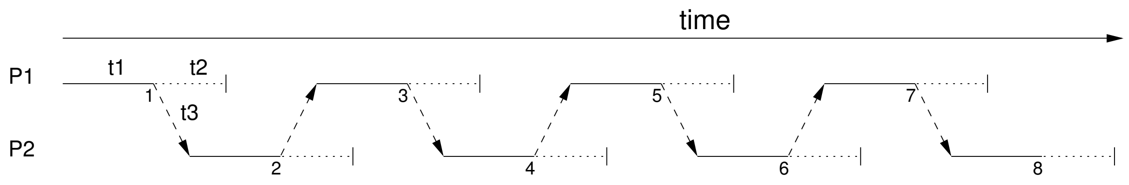


Figure 5.2: Execution schedule of the loop in Figure 5.1(a) parallelized by TLS.

5.2 Motivation

This section motivates Spice by looking at TLS parallelizations of the code in Figure 5.1(a) and illustrates how the insights from the previous section can be used to arrive at a better parallelization.

5.2.1 TLS Without Value Speculation

Figure 5.2 shows how the loop from Figure 5.1(a) is executed on two processor cores by existing TLS techniques that do not employ value speculation. The solid lines represent the execution of the code that performs the list traversal which is synchronized between the iterations. The dotted lines represent the code corresponding to the computation of the minimum element, and the dashed lines represent the forwarding of values from one thread to another thread. The numbers below the lines are the iteration numbers. Let t_1, t_2 and t_3 respectively denote the latency of each of the above three parts of the execution in an ideal execution model when there is no variance in the execution time of these three components due to microarchitectural effects such as cache misses and branch mis-predictions. Let the total number of iterations of the loop be $2n$.

The total time taken to execute the loop by TLS depends on the relation between t_1, t_2 and t_3. If $t_2 > t_1 + 2 \times t_3$, then the computation of `cm` and `wm` lies on the critical path. In that case, the total execution time is roughly equal to $n \times (t_1 + t_2)$, resulting in a speedup of two over single threaded execution. On the other hand, if the computation of `cm` and `wm` is not on the critical path, then the list traversal becomes part of the critical path. If $t_2 \leq t_3$, then the total execution time becomes $2n \times (t_1 + t_3)$. This results in a speedup of $\frac{t_1+t_2}{t_1+t_3}$,

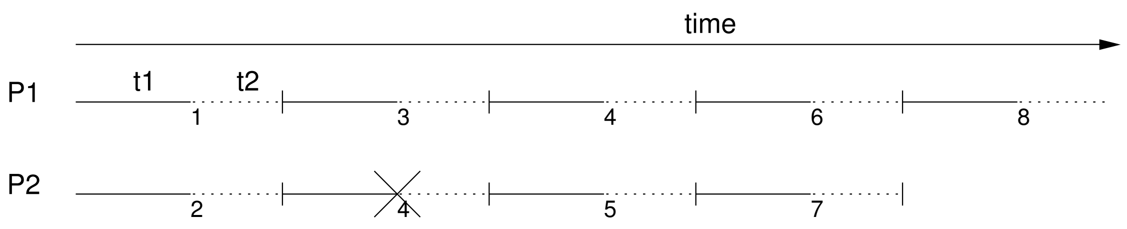


Figure 5.3: Execution schedule of the loop in Figure 5.1(a) parallelized by TLS using value prediction.

which is always less than two, over single-threaded execution. For the loop in Fig 5.1, the list traversal is likely to be on the critical path. This is because of the high cache miss rate of executing the pointer chasing load and also because of our assumption that the `if` statement mostly evaluates to `false`. Thus, the expected speedup of this loop in the ideal case is $\frac{t_1+t_2}{t_1+t_3}$.

5.2.2 TLS With Value Speculation

As an alternative to synchronization, some TLS techniques employ value prediction to predict the value of `c`, eliminating the value forwarding of `c` between iterations. Figure 5.3 shows an execution schedule of the loop under TLS with value prediction. In the example, the prediction of iteration 4 is shown to be wrong, causing iteration 4 to be mis-speculated and re-executed. Let p denote the probability that a given value prediction is correct. Assuming that the probability of a prediction being successful is independent of other predictions, the expected speedup is $\frac{2n(t_1+t_2)}{(n+(1-p)n)(t_1+t_2)}$ or $\frac{2}{2-p}$. If *all* the values of `c` are successfully predicted, then TLS results in a speedup of two over single-threaded execution. Successful prediction depends both on the code sequence that produces the values and the predictor that is used. In this example, the values produced are the addresses of the nodes of a linked list. In `otter`, between successive invocations of the loop in Figure 5.1(a), the minimum element found by the loop is removed from the list and some other nodes are inserted into the list. In this scenario, many proposed value predictors have a limited accuracy:

- The simplest value predictor is the *last value predictor* that predicts that an instruction will produce the same value that it produced in its previous dynamic instance. Obviously, such a predictor cannot predict the address of the nodes of a linked list.

- Another common predictor is the stride predictor. While this is most suited for predicting array addresses, it can also predict linked list nodes as long as the nodes are allocated contiguously in the heap and the order of the nodes seen during the traversal matches the order in which the nodes are allocated. However, in this example, even if the nodes are allocated contiguously, a stride predictor cannot successfully predict all the values of `c` since the insertions and deletions cause the traversal order to be different from the allocation order.

- Some TLS techniques use trace based predictors instead of instruction based predictors. Instead of predicting a value based only on the instruction that produces the value, these predictors try to exploit the correlation of values produced by different instructions in an instruction trace. Marcuello *et al.* [37] proposed the use of trace-based predictors for TLS. They proposed a predictor called *increment predictor* which is a trace based equivalent of a stride predictor. The traces used are loop iteration traces, which are unique paths taken by the program within a loop iteration. There are two paths in the example loop and in both these paths, the value `c` is produced only once, by the same instruction. Hence, for this particular example, a trace based predictor is no more accurate than an instruction based predictor.

Thus, even if value prediction is employed, it is unlikely that existing TLS techniques would significantly improve the performance of this loop. In general, for loops with irregular memory accesses and complex control flow, conventional value predictors fail to do a good job. The next subsection describes how Spice predicts values by *memoizing* the values seen during the previous invocation of the loop to break previously hard-to-predict dependences with very low mis-speculation rates.

```
c = cm->next_cl;
mispred = 1;
while(c != NULL){
  int w = c->pick_weight;
  if (w < wm) {
    wm = w;
    cm = c;
  }
  c = c->next_cl;
  if(c == predicted_c) {
      mispred = 0;
      break;
  }

}
if(!mispred){
    receive(thread2, wm2);
    receive(thread2, cm2);
    if(wm2 < wm){
        wm = wm2;
        cm = cm2;
    }
}
else{
    squash_speculative_thread();
}
```

(a) Non-speculative thread (Thread 1)

```
c = predicted_c;
while(c != NULL){
  int w = c->pick_weight;
  if (w < wm) {
    wm = w;
    cm = c;
  }
  c = c->next_cl;
}
send(thread1, cm);
send(thread1, wm);
```

(b) Speculative thread (Thread 2)

Figure 5.4: Loop in Figure 5.1 parallelized by Spice.

5.2.3 Spice Transformation With Selective Loop Live-in Value Speculation

In the TLS example in Figure 5.3, the iterations of the loop are executed alternately by the two threads necessitating the prediction of `c` in every iteration. The Spice transformation avoids the prediction of `c` in every iteration in by executing a set of contiguous iterations in each thread. Figure 5.4 shows the loop in Figure 5.1 after applying Spice. This example shows parallelization with two threads, but it can be generalized to any number of threads. If there are only two threads, only one value of `c` has to be predicted since there is only one speculative thread. The predicted value is assumed to be present in the variable `predicted_c`. Section 5.3 explains how this prediction is made. Both threads execute the original loop, but with certain differences. The main non-speculative thread contains a check at the end of each loop iteration that checks if the current value of `c` equals the predicted value. When the values are equal, the main thread sets a flag indicating a suc-

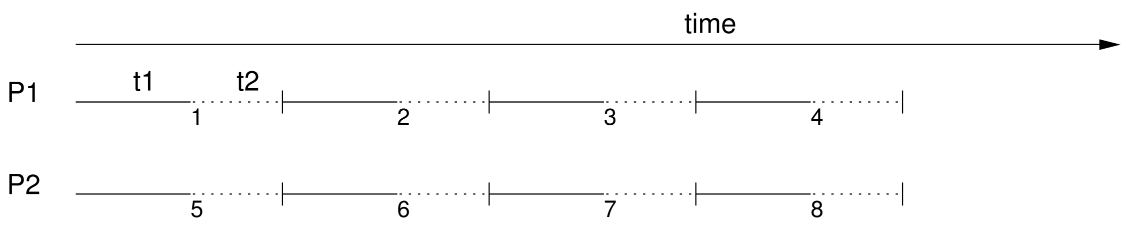


Figure 5.5: Execution schedule of the loop in Figure 5.1(a) parallelized by Spice.

cessful speculation and exits the loop. Outside the loop, the main thread checks if the flag indicating successful speculation is set. If it had exited the loop because of a successful speculation, the main thread receives the `wm` and `cm` values from the speculative thread and computes the minimum among the two `wm`s and the corresponding `cm`. If mis-speculation is detected, the speculative thread is squashed. In the speculated thread, the value of `c` is initialized to `predicted_c`. If the thread exits the loop without getting squashed, the speculative thread sends the `cm` and `wm` values to the main thread.

Figure 5.5 shows the execution schedule for this transformed code. Instead of alternating the iterations among the two processor cores, Spice splits the iteration space into two halves and executes both the halves concurrently in two different cores. Assuming again that the probability of a given prediction being successful is p, and that the predicted value splits the list in the middle, applying Spice results in an expected speedup of $\frac{2}{2-p}$. The expression for the expected speedup of Spice is the same as that of TLS with value speculation. The difference is that Spice requires only a *few* values to be predicted with a high degree of accuracy to achieve good speedup. Consider a simple value prediction strategy where on every loop invocation, the value of `c` in the middle of the list is remembered and used as the predicted value in the following invocation. For the example in Figure 5.1(a), this strategy is likely to result in a high prediction rate since only one node is deleted from the list after each invocation and hence the probability of the remembered node being removed from the list is low.

In general, given n processor cores, the loop in Figure 5.1a(a) can be parallelized into n parallel speculative threads by predicting only $n-1$ values of `c` in one invocation of the loop[1]. Each of these threads executes a chunk of iterations with a low mis-speculation rate. Each Spice thread is long-running compared to iteration-granular TLS. Consequently, Spice does not incur high thread management overhead.

5.3 Compiler Implementation of Spice

This section describes how Spice is implemented as an automatic compiler transformation. Algorithm 6 outlines the Spice transformation. The inputs to this algorithm are the loop to be parallelized and the number of available threads. The algorithm first computes the set of live-ins that require value prediction. This set is obtained by first computing the set of all inter-iteration live-ins. Those live-ins in this set that can be subjected to reduction transformations [1] such as sum reduction or MIN/MAX reduction do not require prediction. The rest of the loop carried live-ins require value prediction. If this set of live-ins contains memory variables that cannot be register-promoted, the transformation cannot be applied to the loop. Otherwise, the compiler then performs the following steps:

Algorithm 6 Spice transformation

```
1: Input: Loop $L$, number of threads $t$
2: Compute inter-iteration live-ins $Liveins$
3: Compute reduction candidates $Reductions$
4: Live-ins to be speculated $S = Liveins - Reductions$
5: **if** $S$ contains memory dependences **then**
6:     return
7: **end if**
8: Create $t - 1$ copies of the body of $L$ into separate procedures
9: Insert communication for non-speculative loop live-ins and live-outs
10: Generate code to initialize speculative live-ins $S$
11: Generate recovery code in speculative threads
12: Insert code for mis-speculation detection and recovery in the non-speculative thread
13: Insert value predictor
```

Thread creation: The compiler replicates the loop $t - 1$ times and places the loop copies in separate procedures. Each of these procedures is executed in a separate thread. To

[1]The loop-carried dependence for `wm` can be eliminated by applying reduction.

avoid spawning these threads before every invocation of the loop, threads are pre-allocated to the cores at the entry to the main thread.

Communication of live-ins and live-outs: The compiler identifies the set of register live-ins to the loop that needs to be communicated to the speculative threads. All live-ins except the speculative live-in set S and the set of accumulators are communicated. Variables used as accumulators are initialized to 0 in the speculative threads.

Value speculation: The compiler creates a global data structure called the *speculated values array* of size $(t-1) \times m$, where m is the number of live-in values that require speculation. The compiler initializes the speculative live-ins of the loop in thread i with the values from the $(i-1)^{th}$ row of the speculated values array. The process of obtaining the contents of this array are obtained is discussed later.

Recovery code generation: The compiler creates a recovery block for each speculative thread and generates the code to perform the following actions:

1. Restore machine specific registers including the stack pointer, frame pointer, etc. Registers used within the loop that is parallelized are simply discarded.

2. Rollback the memory state if the loop contains stores.

3. Inform the main thread that the recovery is complete.

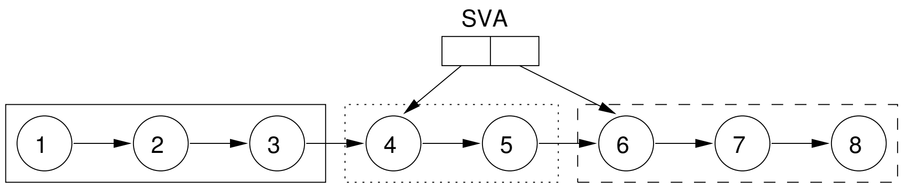
4. Exit the recovery block and jump to the program point where it waits for the main thread to send a token that denotes the beginning of the next invocation

Mis-speculation detection and recovery: Mis-speculation detection is done in a distributed fashion. Mis-speculation detection is first discussed in the context of only one speculative thread (a total of two threads) and then generalized for t threads. Let S be the set of all the loop live-in registers that need speculation. At the beginning of each loop iteration, the non-speculative thread 1, which is also referred to as the *main thread*, compares its values of the registers in S with the values used to initialize those live-in registers in thread 2. If the values of all the registers in S match at the beginning of some iteration j, it implies that thread 2 started its execution from iteration j of the loop with correctly speculated live-in values. In that case, thread 1 stops executing the loop before executing iteration j and waits for thread 2 to communicate its live-outs at the end of the loop. On the other hand, if the values never match, thread 1 eventually exits the loop by taking the original exit branch of the loop. Since it has executed all the iterations of the loop and exited the loop normally, it concludes that thread 2 has mis-speculated. In that case it executes a `resteer` instruction to redirect thread 2 to its mis-speculation recovery code.

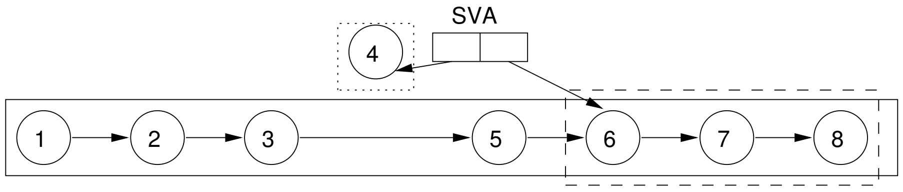
Mis-speculation detection and recovery is now generalized for t threads. Thread i is responsible for detecting whether thread $i + 1$ has mis-speculated in a loop invocation. The compiler generates code in thread i at the beginning of each loop iteration to compare the values of all the registers in set S with the initial values of thread $(i + 1)$'s live-ins. Thread $(i + 1)$'s initial live-in values are loaded from the i^{th} row of the speculated values array. It then inserts code that sets a flag indicating successful speculation, followed by a branch to exit the loop, if the values match. Outside the loop, the compiler emits code to check this flag and take the necessary recovery action.

To recover from mis-speculation, the compiler generates code to perform the following actions. The thread detecting mis-speculation, if it is not the main thread, communicates this information to the main thread. The main thread re-steers all mis-speculated threads to execute their recovery code. It then waits for an acknowledgment token from each of the speculative threads indicating that they have successfully rolled back the memory state. Finally, after all the tokens are received, the main thread commits the current memory state.

Mis-speculation detection is illustrated in Figure 5.6. Figure 5.6(a) shows the traversal of a list with eight nodes. Three threads participate in the traversal of this list. The first thread traverses the nodes enclosed by the solid rectangle, the second thread traverses the nodes enclosed by the dotted rectangle and the nodes of the last thread are enclosed by the dashed rectangle. SVA denotes the speculated values array whose elements contain the



(a) Original value prediction



(b) After removal of node 4 from the list

Figure 5.6: Figure illustrating the effects of the removal of a node on Spice value prediction.

addresses of list nodes which are live-in to the list. Assume that after the first invocation, node 4 is removed from the list as in Figure 5.6(b). The SVA entry still points to the removed node. During the next invocation, it compares the nodes it traverses with the node in the SVA. Since it never finds that node, it traverses the entire list. The second thread starts from the removed node and, depending on the contents of its next pointer field, will either stop the traversal, loop forever, or cause memory faults by accessing some invalid memory location. Thread 3 starts from node 6 and traverses till the end of the list, repeating the work done by thread 1. When thread 1 reaches the end of the list, it concludes that thread 2 has mis-speculated and squashes threads 2 and 3. Note that if thread 1 compares its live-in registers with the speculated live-ins of both threads 2 and 3, then it needs to squash only thread 2 and not thread 3. However, this increases the overhead in thread 1 due to the additional comparisons and hence the compiler limits the comparisons to only one set of live-ins. If thread 2 goes into an infinite loop, the `resteer` issued by thread 1 makes it jump to the correct recovery code. The recovery code rolls back its memory state to undo speculative updates to its memory state.

Value predictor insertion

The compiler inserts a software value predictor that fills the contents of the speculated values array on every loop invocation. A simple value prediction strategy is to *memoize* or remember the live-ins from $t - 1$ different iterations during the first invocation. These $t - 1$ set of live-ins can be used as the predicted values in all subsequent invocations. This approach does not adapt to change in program behavior and hence is not effective. For instance, if the values that are memoized are addresses of the nodes of a linked list, and if a memoized node is removed from the list, all subsequent invocations will mis-speculate even if there are no further changes to the list.

A better approach is to make the predictor memoize the values not just during the first invocation, but on every invocation of the loop. Values memoized in one invocation are used as predicted values only in the next invocation. This approach adapts itself better to changes to the loop structure. Moreover, if the list grows or shrinks, nodes can be memoized in such a way that the work done by speculative threads in the next invocation is balanced.

The value predictor has two components. The first component of the value predictor writes to the speculated values array, and its implementation is distributed across all the threads. To implement this component, the compiler first creates a set of data structures used by the predictor. A list called `svai` is created per thread. The entries of this array for thread i contain the indices of the speculated values array (`sva`) to which thread i should write to. Another per-thread list called `svat` contains work thresholds that determine which values are memoized. The compiler also creates an array called `work` whose entries contain the amount of work done by each thread. This is used in dynamic load balancing.

The compiler generates the code corresponding to Algorithm 7 in each thread to memoize the values. Each thread maintains a counter `my-work` that is a measure of the amount of work done by that thread. In the implementation described in this dissertation, the threads increment the counter once per loop iteration. A more accurate measure of the work done could be obtained by making use of hardware performance counters. If the

Algorithm 7 Spice value prediction

```
my-work = 0
for each iteration of the loop do
  my-work = my-work + 1
  if my-work > svat[list-index] then
    sva[svai[list-index]] = loop carried live-ins in current iteration
    list-index = list-index+1
  end if
end for
work[my-thread-id] = my-work
```

work done so far exceeds the threshold found at the head of the `svat` list for this thread, then the current loop live-in values are recorded in the `sva` array. The index to the `sva` array location to which the value must be written is given by the value at the head of the `svai` list. After writing to the `sva` array, the head pointers of both the lists are incremented. After exiting the loop, the thread writes the total work done in that invocation to the global `work` array.

The other component of the value predictor is centrally implemented in a separate procedure. This component is executed at the end of each loop invocation. It collects the amount of work done by each thread in that invocation and decides the iterations of the next loop invocation whose live-in values are memoized. Depending on which threads execute those iterations, it generates the entries of the `svai` list. The partitioning of the iterations among the threads in the next invocation cannot be decided a priori at the end of current invocation since it depends on the program state. Hence, the predictor makes the following assumptions:

1. In future invocations, the total amount of work done will remain the same as in the current invocation.

2. In the immediately following invocation, all the threads will execute the same amount of work.

These assumptions enable the predictor to orchestrate the memoization in a way that results in load balancing. This is best illustrated with an example. Consider the case where there are three threads and the work done by each thread in a particular invocation are 10, 1 and 1 units respectively. Based on the first assumption, the subsequent invocations will also perform a total of twelve units of work. Hence the predictor wants to collect the live-ins when the total work done in a sequential execution are four and eight. It then uses the second assumption to determine which threads have to write those values. In this example, the work done by the first thread in the next invocation is assumed to be ten and so both the rows of the `sva` are written to by the first thread after iterations four and eight and the other two threads do not write into the `sva`. Hence the `svat` list of the first thread is set to [4, 8] and its `svai` list is set to [0,1]. For the other two threads, the head element of `svat` is set to ∞ so that they never write to the `sva` array.

5.4 Related Work

Marcuello *et al.* [37] proposed the use of value prediction for speculative multi-threaded architectures. They investigated both instruction based predictors and trace based predictors, in which the predictor uses the loop iteration trace as the prediction context. Steffan *et al.* [64] propose the use of value prediction as part of a suite of techniques to improve value communication in thread level speculation. They use a hybrid predictor that chooses between a stride predictor and a context based predictor. Cintra and Torellas [10] also propose value prediction as a mechanism to improve TLS. They use a simple *silent store predictor* that predicts the value of a load to be the last value written to that location in the main memory. Liu *et al.* [32] predicts only the values of variables that look like induction variables. Oplinger [43] also incorporates value prediction in TLS design, with a particular focus on function return values. The focus of this work is a compiler based technique that predicts a few values with high accuracy to extract thread level parallelism.

Zilles [78] proposed Master/Slave speculative parallelization(MSSP), which is a speculative multi-threading technique that uses value speculation. The prediction in MSSP is made by a *distilled program*, which executes the speculative backward slice of the loop live-ins. The software value predictor used in Spice predicts based on values seen in the past and does not execute a separate thread for prediction.

Some TLS techniques [9, 10, 48, 49] have also proposed the use of iteration chunking. These chunks are iterations of fixed size, while in our case the number of chunks equals the number of processor cores and the chunk size is determined at runtime by the load balancing algorithm. The LRPD test [57] and the R-LRPD test [13] are also speculative parallelization techniques that chunk the iteration space, but they use memory dependence speculation and not value speculation.

Chapter 6

Experimental Evaluation

This chapter performs a quantitative evaluation of the transformations proposed in this dissertation. The compilation framework, simulation methodology and a description of the benchmarks are first presented. Then this chapter presents an evaluation of PS-DSWP and speculative PS-DSWP and makes a performance comparison with DSWP and TLS. Finally, an evaluation of Spice is presented.

6.1 Compilation Framework

The transformations described in this dissertation are implemented as part of the VELOCITY compiler [69]. The application is first compiled into a low-level intermediate representation to which a series of classical optimizations are applied. Then, the parallelization transformation is applied. The parallelized code is again subjected to a round of classical optimizations and then lowered into IA64 assembly code. This is followed by a first pass of scheduling, register allocation and a final scheduling pass. Performance comparisons are made with respect to a single-threaded code which is subjected to the exact sequence of transformations described above, except the parallelization step.

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

Table 6.1: Details of the multi-core machine model used in simulations.

6.2 Simulation Methodology

Since the compiler transformations described in this dissertation require special hardware support in the form of communication queues, multi-threaded transactions and ISA support to program these hardware structures, they cannot be implemented on commodity multi-processor systems. Hence simulation of a multi-processor with the necessary hardware modifications is employed to evaluate the techniques. This dissertation employs two different simulation approaches that are described below.

6.2.1 Cycle Accurate Simulation

Suitable candidates for non-speculative PS-DSWP and Spice are typically hot inner loops that account for significant fraction of a program's execution. At the granularity of outermost loops in a general-purpose program, there are usually many complex memory dependences, whose dependence distance is hard to detect by static analysis. PS-DSWP typically does not work well on such loops since those dependences have to be conservatively treated as loop-carried, preventing large parallel stages from being formed. The applicability of Spice to such loops is limited by the fact that the memory locations involved in such dependences typically cannot be register promoted.

For inner loops with relatively short per-iteration execution time, cycle-accurate simulation of a multi-core IA-64 processor is used. Table 6.1 lists some of the important parameters of the processor model. The individual cores of this processor are validated, with

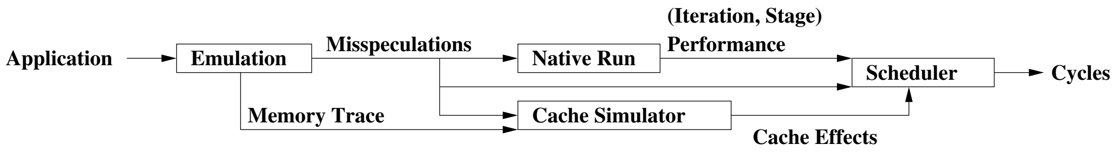


Figure 6.1: Block diagram illustrating native-execution based simulation.

IPC and constituent error components accurate to within 6% of real hardware for measured benchmarks [47]. The simulator is built using the Liberty Simulation Environment [71].

Even if a loop is innermost, its simulation cost could be prohibitive if it is a hot loop. To keep simulation costs realistic, invocations of the loop are randomly sampled with some specified probability and simulated in detail. Except for the loops that are sampled, the rest of the program is executed by the simulator in a *fast-forward* mode that maintains only the correct architectural state without a cycle-accurate simulation.

6.2.2 Native Execution Based Simulation

A cycle-accurate structural simulator of a modern uni-processor can simulate less than forty thousands of cycles per second [46]. Assuming a linear decrease in simulation throughput when simulating a CMP, an eight core CMP can be simulated at the rate of around five thousand cycles per second. This makes it prohibitively expensive to simulate outer loop iterations of most programs in the SPEC2000 and SPEC2006 benchmark suites. Since a primary advantage of speculative PS-DSWP is the ability to extend the scope of applicability to outer-most loops in programs, it is impractical to use the cycle-accurate simulation methodology to evaluate speculative PS-DSWP. Hence a native-execution based simulation methodology proposed by Bridges [5] is used to evaluate speculative PS-DSWP.

Figure 6.1 gives an overview of the simulation framework. Measuring the performance of a parallelized loop consists of the following steps:

1. The parallelized application is first emulated on a native multi-processor machine by replacing the MTX (multi-threaded transactions) and communication operations

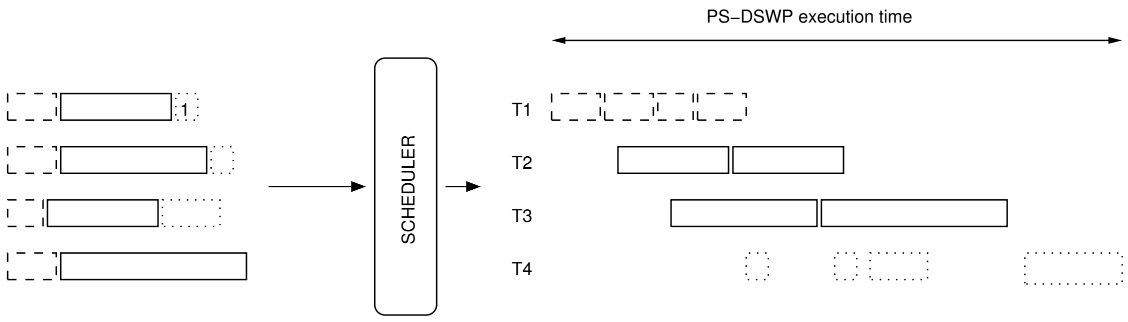


Figure 6.2: Construction of PS-DSWP schedule from per-iteration, per-stage native execution times.

with calls to a library that emulates the corresponding functionality in software. This multi-threaded binary is instrumented to collect the memory trace and a *mis-speculation* trace that contains information on which iterations of the loop mis-speculate

2. A separate binary is generated to collect native execution times of the loop iterations. This avoids the use of MTX by inserting additional synchronizations into this binary. These synchronizations ensure that a stage executes an iteration only when all the prior iterations and all the prior stages in the current iteration have been executed. The information from the mis-speculation trace is used to determine the iterations that mis-speculate and sequential execution is used to execute those iterations. This restricted execution model ensures correctness even in the absence of MTX and hence can run directly on native hardware. The IA64 performance counters are used to collect cycle counts for each stage for every iteration.
3. The per-iteration, per-stage cycle counts computed in the previous step does not reflect the additional overhead associated with MTX. To compensate for this, a MTX simulator is used to find additional costs of memory accesses. This simulator uses the memory trace and the mis-speculation trace generated in the first step and adds the MTX overhead to the per-iteration, per-stage execution times. The MTX simulator is described by Vachharajani [72].

4. The augmented per-iteration, per-stage cycle counts and the mis-speculation trace are fed into a PS-DSWP scheduler that "schedules" these individual cycle counts as well as the mis-speculation trace to compute the execution time of the parallelized loop. This is depicted in Figure 6.2. In this example, the pipeline has three stages with the second stage being a parallel stage with a replication factor of two. The first stage is shown by a dashed rectangle, the second by a solid rectangle and the third stage by a dotted rectangle. The loop is assumed to execute for four iterations. The data from the native run is visually represented on the left side of the scheduler, which constructs a valid PS-DSWP schedule as depicted on the right.

6.3 Benchmarks

The techniques presented in this dissertation are evaluated on a variety of programs from different benchmark suites. All of these applications are written in C and contain characteristics of many general-purpose applications such as complex control flow and irregular memory access patterns. The benchmarks used to evaluate PS-DSWP and speculative PS-DSWP are those that show promising improvement using the performance estimation framework described in Chapter 4. The benchmarks used to evaluate Spice are chosen by manually identifying loops with characteristics that can be exploited by Spice. Table 6.2 contains the descriptions of all the benchmarks used in the evaluation.

6.4 Non-speculative PS-DSWP

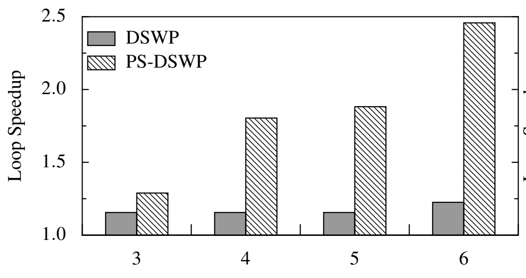
PS-DSWP is evaluated on a set of loops which cannot be parallelized by DOALL and which contribute to at least 15% of the application's execution time. The loops that are selected are those whose estimated speedup exceeds 50%. The details of the selected loops are presented in Table 6.3. The loop in `300.twolf` had to be manually annotated to indicate the absence of a memory dependence, since the current pointer analysis used in VELOCITY is

Benchmark	Suite	Description
052.alvinn	SPEC 92	Trains a neural network called ALVINN (Autonomous Land Vehicle In a Neural Network) using back-propagation
175.vpr	SPEC 2000	A placing and routing algorithm
181.mcf	SPEC 2000	Single-depot vehicle scheduling in public mass transportation
197.parser	SPEC 2000	A syntactic parser of English, based on link grammar
256.bzip2	SPEC 2000	Data compression algorithm
300.twolf	SPEC 2000	TimberWolfSC placement and global routing package
456.hmmer	SPEC 2006	Profile Hidden Markov Model
458.sjeng	SPEC 2006	A program that plays chess and several chess variants
swaptions	Parsec	Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaption
ks	–	Kernighan-Lin graph partitioning
otter	–	A theorem prover for first-order logic

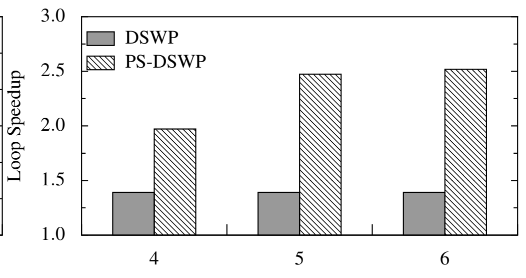
Table 6.2: A brief description of the benchmarks used in evaluation.

Benchmark	Loop	Hotness	Pipeline stages
ks	FindMaxGpAndSwap (outer)	98%	$s \rightarrow p$
otter	find_lightest_geo_child	15%	$s \rightarrow p \rightarrow s$
300.twolf	new_dbox_a (outer)	30%	$s \rightarrow p$
456.hmmer	P7_Viterbi (inner)	85%	$p \rightarrow s$
458.sjeng	std_eval	26%	$s \rightarrow p$

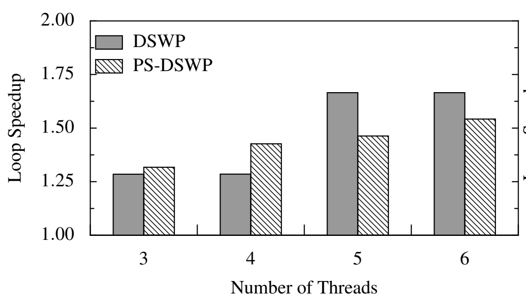
Table 6.3: Details of loops used to evaluate non-speculative PS-DSWP. The hotness column gives the execution time of the loop normalized with respect to the total execution time of the program. In the "pipeline stages" column, an s indicates a sequential stage and a p indicates a parallel stage.



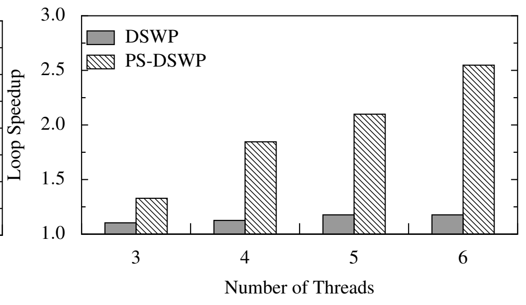
(a) ks



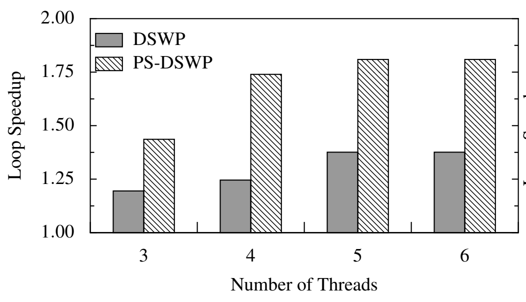
(b) otter



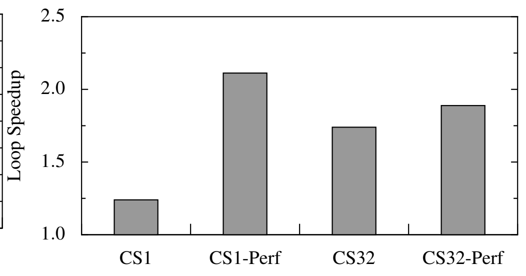
(c) 300.twolf



(d) 458.sjeng



(e) 456.hmmer



(f) Chunking on 456.hmmer with 4 threads

Figure 6.3: Speedup of non-speculative PS-DSWP and non-speculative DSWP over single-threaded execution. Geometric mean speedup of PS-DSWP with 6 threads is 2.14. Geometric mean speedup of DSWP with 6 threads is 1.36.

not powerful enough to conclude that. The pipeline stages obtained as a result of applying the PS-DSWP transformation is also given in the final column of Table 6.3.

Figures 6.3(a)-(e) show the performance of the five selected loops after applying PS-DSWP. For each loop, the graphs compare the speedups obtained by DSWP and PS-DSWP for the same number of threads. Using up to 6 threads, PS-DSWP shows a geometric-mean speedup of 2.14 and a peak speedup of 2.55 in `458.sjeng`. On the same loops, with the same number of threads, DSWP yields a speedup of only 1.36. In all the benchmarks except `300.twolf`, PS-DSWP outperforms DSWP for any number of threads. The loop in `300.twolf` executes only 3 or 4 iterations on average; hence executing the parallel stage in more than three threads results in diminishing returns. Since the body of the loop in `300.twolf` is large with many SCCs, a good balanced DSWP partition is possible. The loop in `456.hmmer` also shows poor scalability, even though the loop iterates 300 times per invocation. In `456.hmmer`, the sequential stage takes a significant amount of time to execute and becomes the bottleneck after a replication factor of 3 for the parallel stage. In the other three loops, PS-DSWP scales up to six threads, while DSWP plateaus with fewer threads.

Figure 6.3(f) shows the effect of applying iteration chunking to the loop in `456.hmmer` with four threads. The bar on the left shows the speedup obtained when iteration chunking was not applied. The second bar shows the speedup for the same code when all data accesses were assumed to always hit in the L1 cache. This speedup number is relative to a single-threaded baseline that was also simulated with a perfect cache. The difference in these speedups indicates that cache effects significantly influence the speedup of the multi-threaded code in this loop. The third bar shows that most of this performance difference goes away using the cache parameters shown in Table 6.1 and a chunk size of 32. This suggests that chunking can be effective in mitigating the effects of false sharing. The final bar shows the speedup of the chunked code with a perfect data cache. This speedup is less than what can be obtained with a perfect cache when chunking is not applied, indicating that iteration chunking results in a loss of parallelism.

6.5 Speculative PS-DSWP

Table 6.4 lists the benchmarks used to evaluate speculative PS-DSWP. Most of the execution times of these benchmarks is spent in the loops chosen for parallelization. Figure 6.4 shows the performance of PS-DSWP with four and eight threads. The geometric mean

Benchmark	Function	Hotness
052.alvinn	`main`	98%
175.vpr	`try_swap`	99%
197.parser	`batch_process`	99%
256.bzip2	`compressStream`	99%
456.hmmer	`main_loop_serial`	98%
swaptions	`HJM_Swaption_Blocking`	98%

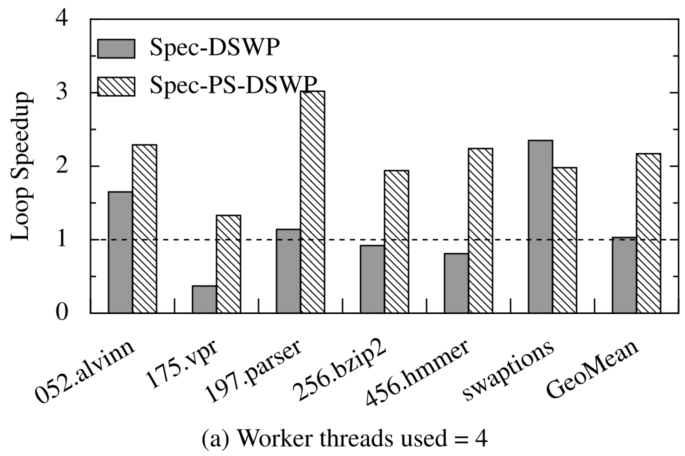
Table 6.4: Details of the loops used to evaluate speculative PS-DSWP.

speedup with four threads is 2.17 and with eight threads is 3.67. For comparison, the speedup obtained by speculative DSWP on these benchmarks is also shown in the graphs in Figure 6.4. The same speculation thresholds are used for DSWP and PS-DSWP and the same simulation methodology is used in both the cases. DSWP performs poorly in most of these benchmarks with some significant *slowdown* relative to single-threaded execution in some of the loops. The rest of this section takes a closer look at the performance characteristics of some of the benchmarks used in the evaluation.

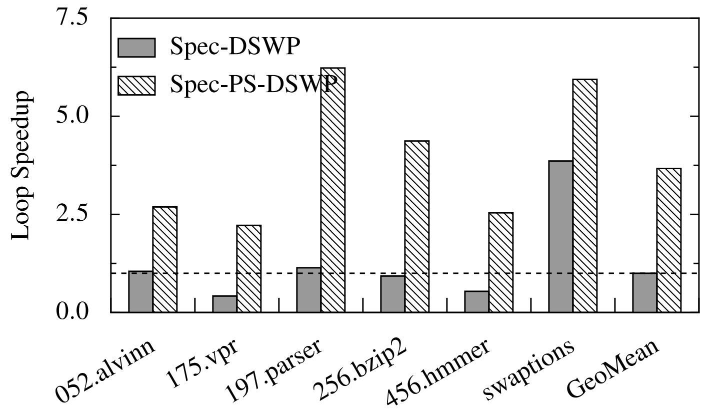
6.5.1 A closer look: 197.parser and 256.bzip2

The benchmark `197.parser` performs syntactical parsing on an input consisting of English language sentences. The program's outermost loop, which processes each sentence of the loop, is parallelized. Since the grammatical structure of a sentence does not depend on other sentences, this loop is a good candidate for PS-DSWP. The partitioner divides the loop into three stages, with the middle parallel stage contributing to a large fraction of the execution time. The PS-DSWP speedup scales well as long as there is enough work for all the threads to process.

The DSWP speedup of this loop is 1.14 with both four and eight threads. The speedup is limited by a large SCC which accounts for a little more than 80% of the total profile weight of the loop. This SCC roughly consists of most operations in the body of the inner loop that parses a single sentence. This underscores one of the main advantages of PS-DSWP: its ability to distinguish inter-iteration dependences *with respect to the loop being*



(a) Worker threads used = 4



(b) Worker threads used = 8

Figure 6.4: Speedup of speculative PS-DSWP and speculative DSWP over single-threaded execution. The horizontal dashed line corresponds to a speedup of 1.0.

parallelized from inter-iteration dependences in inner loops. These inner loop dependences may contribute to large SCCs that limit the performance of DSWP, but not that of PS-DSWP.

The performance characteristics of the loop in `256.bzip2` is similar to the one in `197.parser`. The loop being parallelized is the outer-most loop of the compresser. The bzip2 algorithm divides the data into fixed size blocks and compresses them. Since the compression of one block is independent of another, PS-DSWP is well suited for this benchmark. Again, the performance of DSWP suffers because of the presence of a large SCC due to inter-iteration dependences with respect to an inner loop.

In both these benchmarks, the speculated dependences are the memory dependences that never manifest, but whose absence cannot be established by static memory analysis. Furthermore, both of these benchmarks benefit from the privatization effect provided by multi-threaded transactions.

6.5.2 A closer look: swaptions

In `swaptions`, the non-trivial SCCs contribute to only a small portion of the execution time of the loop body. Hence, the DSWP partitioner is able to divide it into stages of roughly equal weight. However, it introduces a large number of communication operations in the process. This communication overhead significantly affects the speedup of DSWP. In the case of PS-DSWP, a three stage partition similar to `197.parser`, with most of the time spent in the middle stage. Hence, its speedup is determined by the number of threads executing the parallel stage and not the total number of threads. Thus, in the four thread case, speedup obtained by PS-DSWP is 1.98, which is less than the speedup obtained by DSWP. However, as the number of available threads increase to eight, the speedup obtained by PS-DSWP becomes close to 6, which is the number of threads that execute the parallel stage. In the case of DSWP, an eight stage partition results in more communication operations resulting in a lower speedup than PS-DSWP.

6.5.3 A closer look: 175.vpr and 456.hmmer

The main bottleneck for performance in `175.vpr` is the number of mis-speculations and the associated cost in recovering from them. This affects the performance of both PS-DSWP and DSWP. A high mis-speculation rate affects performance in several ways. First, the mis-speculated iteration is executed sequentially. There is also an additional cost associated with re-executing the mis-speculated iteration since the commit thread has to communicate the live values of the subsequent iteration to the threads executing the stages. Finally, the assumption that pipeline fill and drain costs are negligible no longer holds true since the recovery iteration can be executed only after all prior iterations complete, and so each mis-speculation necessitates draining of the pipeline.

The main contributor to the significant slowdown experienced by DSWP over single-threaded execution is the high variance in execution times that is not accounted for during thread partitioning. The execution times of the stages vary widely from one iteration to another. In the absence of mis-speculations, the decoupling between the threads helps to tolerate this variance to a certain degree. However, the frequent mis-speculations and the consequent draining of the pipeline counteract the decoupling effect.

`456.hmmer` also suffers from mis-speculation, but the mis-speculation rate is lower than `175.vpr` and hence the effects are reduced. The presence of a large SCC in an inner loop further hurts the performance of DSWP.

6.6 Comparison of Speculative PS-DSWP and TLS

This section presents a quantitative comparison between speculative PS-DSWP and TLS to better understand the differences between these two techniques. There are many different TLS proposals with varying approaches to speculation and synchronization of dependences. The results presented in this chapter are based on a TLS system described in Zhai [75]. This particular proposal is chosen for comparing PS-DSWP because this TLS system relies primarily on the compiler to do the synchronization and does not delegate that task entirely to hardware like many other TLS proposals. Faithfully implementing an existing TLS system for comparison is not feasible because of the significant effort required, non-availability of many of the implementation details and the differences in architectural and micro-architectural details between the processor used to evaluate speculative PS-DSWP and existing TLS systems. Hence, this work uses a native-execution based simulation similar to the one used to evaluate speculative PS-DSWP.

1. The first step is to identify all inter-iteration dependences and classify them into those that need to be synchronized and those that are speculated. All register dependences are synchronized. The choice between speculation and synchronization for memory dependences is based on the profile information obtained from LAMP. The information about speculated dependences are annotated into the program IR.

2. For dependences that need synchronization, the optimal points to insert the synchronization are inserted based on the analysis described in Zhai *et al.* [76]. These analyses are implemented in the VELOCITY compiler.

3. A special profiler is used to obtain a trace of the iterations during which the speculated dependences manifest. This trace records the loop invocation, loop iteration and dependence distance for each occurrence of a speculated dependence.

4. The single-threaded program is instrumented to record the cycles from the beginning of each iteration at which the synchronization points identified in Step 2 are executed. In addition the cycle at which the *home free token* [75] has to be inserted is also recorded.

5. In the final step, a TLS scheduler similar to the PS-DSWP scheduler described earlier uses the cycle counts from step 4 and the speculated-dependences from step 3 to obtain the TLS execution time of the loop.

Figure 6.5 shows the speedup obtained by TLS using the methodology described above. For comparison, the speculative PS-DSWP speedups are also given. Speculative PS-DSWP uses eight worker threads and a commit thread, while the TLS results are obtained using nine threads. In all benchmarks except `swaptions`, speculative PS-DSWP outperforms TLS. Some major factors which contribute to the performance advantage of speculative PS-DSWP and an analysis of the performance difference in `swaptions` are discussed next.

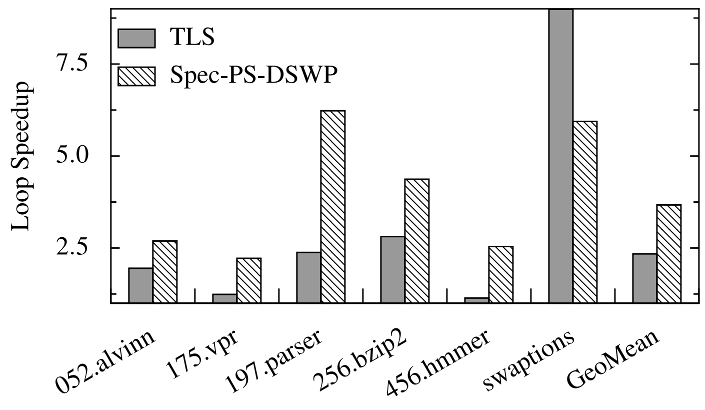


Figure 6.5: Comparison of TLS and speculative PS-DSWP. Worker threads used in speculative PS-DSWP = 8. Threads used by TLS = 9.

Synchronization stalls

Consider a strongly connected component in the PDG of a loop. Speculative PS-DSWP assigns this SCC to a sequential stage, thereby limiting the parallelism. TLS synchronizes the loop carried dependences in this SCC using *wait* and *signal* primitives. A signal has to be inserted after the source of the dependence and the wait has to be inserted before the destination. For every scalar variable or an abstract memory location, exactly one signal and wait has to be executed per iteration of the loop that is parallelized. A naïve approach to synchronization is to insert all the signals at the end of a loop iteration and all the waits at the beginning of an iteration, but this serializes the execution of the entire loop. Zhai *et al.* [76] propose a data-flow analysis and a scheduling algorithm to optimize the placement of the wait and signal primitives. There are two situations in which the placement according to this algorithm still results in significant stall cycles.

The first case is the inefficient placement of the synchronization primitives in the presence of complex control flow. The code in Figure 6.6 is based on a code fragment in the loop in `197.parser`. The variable `errors` is loop carried and is hence synchronized. The position of wait and signal shown in the figure is as determined by the synchronization optimization algorithms. As a result of this synchronization, the calls to `foo` which

```
for(...){
  wait(errors);
  if(condition1){
    errors++;
  }
  foo() //expensive computation
  if(condition2){
    errors++;
  }
  signal(errors);

}
```

Figure 6.6: Insertion of TLS synchronization in the presence of control flow.

```
for(i = 0; i < 10; i ++){
    wait(k);
    while(...){
        a[i] = foo(a[i]);
        k++;
    }
    signal(k);
}
```

Figure 6.7: Synchronization of an inter-iteration dependence inside an inner loop by TLS.

are sandwiched between the two `if` statements are also serialized even though they are completely independent of the two `if` statements. In comparison, speculative PS-DSWP assigns only the `if` statements to a sequential stage and the calls to a parallel stage since they are in two different SCCs. The speedup of `197.parser` increases from 2.38 to 2.81 if the calls are outside the synchronized region.

The problem in this specific example can be mitigated by applying an aggressive scheduling algorithm proposed by Zhai et al. [76]. The aggressive algorithm uses control and memory speculation in conjunction with data-flow analysis and scheduling. Since in this example, the error conditions are false with high frequency, the signal gets moved above the call in the common case. However, if the branches are not biased, the aggressive scheduling algorithm also produces the same synchronization shown in Figure 6.6, whereas the SCC based approach works irrespective of whether the branch is biased or not.

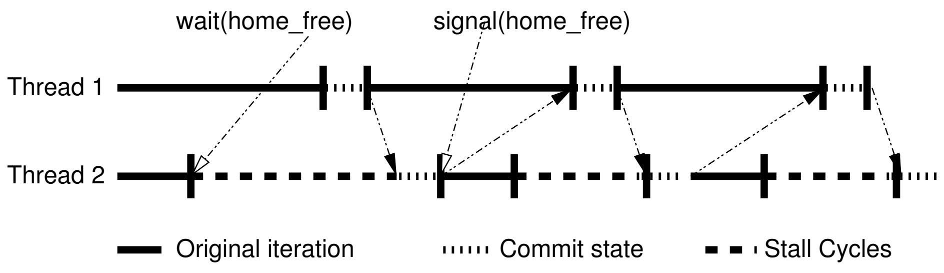


Figure 6.8: An execution schedule of TLS showing the steps involved in committing an iteration.

The second case is that of inter-iteration dependences whose source and destination operations are contained in some inner loop. Consider the loop in Figure 6.7, where the execution of `foo` in different iterations of the outer loop are assumed to be speculatively independent. The variable `k` is synchronized using signal and wait primitives. This synchronization effectively serializes the execution of the entire loop, even though the inner loop invocations can execute concurrently with the exception of the addition of the variable `k`. As in the previous example, the SCC based approach of speculative PS-DSWP assigns `foo` to a parallel stage. Unlike the previous example, even the aggressive scheduling approach does not help in this case.

All the loops that are used in this evaluation have at least one dependence which is contained within some inner loop. Their impact on performance depends on how much of parallelizable portion is serialized due to the synchronizations. For instance, in `052.alvinn`, ignoring just one such inner-loop dependence can increase the speedup obtained by TLS from 1.95 to 7.38. Note that even an optimal placement of signal and wait for this dependence would incur some synchronization cost making this speedup unrealizable, but this shows the potential cost due to the synchronization of inner-loop dependences. Another such example is seen in `256.bzip2`. By ignoring inner-loop dependences in three different functions — `spec_getc`, `spec_putc` and `bsW` — the speedup increases to 6.14.

Assignment of iterations to cores

The ability to dynamically map iterations to cores gives speculative PS-DSWP a significant advantage over TLS. Such a dynamic assignment is not possible in TLS. In TLS execution, a token called *home-free* token is passed among the cores. After executing the original loop iteration, a thread waits for the home-free token from the thread executing the preceding iteration. After receiving the home-free token, a thread commits its speculative state and then signals the token to the next thread. This is illustrated in Figure 6.8. An important observation is that even if a thread has executed the original iteration, it cannot start executing the next iteration till the immediately previous iteration has committed. This restriction prevents TLS from dynamically mapping iterations to threads.

In the benchmarks that are used to compare TLS and speculative PS-DSWP, `197.parser` suffers heavily to the lack of dynamic assignment in TLS. Even if *all* synchronizations are ignored, and no mis-speculation occurs, the speedup achievable by TLS is only 2.81 – much less than the 6.23 speedup obtained by speculative PS-DSWP. This gap can be wholly attributed to dynamic assignment of iterations to threads. The presence of a few long sentences and many shorter sentences in the input set causes many stalls to be introduced in TLS.

Performance analysis of swaptions

In `swaptions`, there are only three dependences that need to be synchronized. Even though they are inside an inner loop, that inner loop's execution constitutes a small fraction of the execution time of the loop that is parallelized. Hence, the cost of synchronizing the dependences is hidden by the execution of the rest of the loop body. The loop never mis-speculates and the execution times of the different iterations are more or less the same. All of these factors result in TLS obtaining almost 9 times speedup with nine threads. In the case of speculative PS-DSWP, the loop is partitioned into a three stage pipeline with a sequential first and third stage and a parallel second stage. Both the sequential stages are

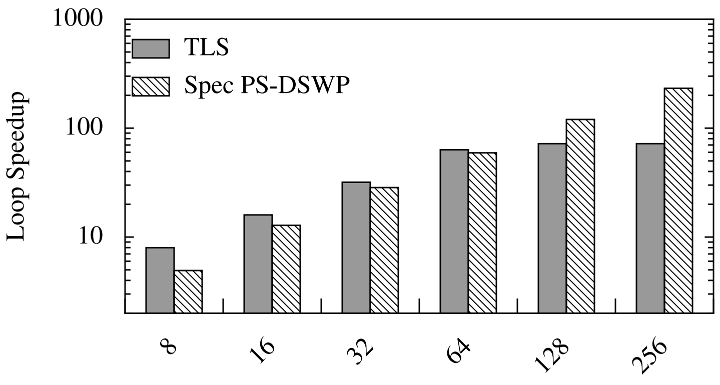


Figure 6.9: Performance of `swaptions` with TLS and speculative PS-DSWP for threads ranging from 8 to 256.

negligible in terms of execution time. Hence the speedup with nine total threads is very close to 6, which is the number of threads executing the parallel stage.

The harmful effects of cyclic communication in TLS starts to affect the performance as the number of threads exceeds a limit. Figure 6.9 shows how the performance of TLS and speculative PS-DSWP changes with an exponential increase in the number of threads. The speedup of TLS peaks at 72 after which an increase in the available number of threads does not improve the performance. This is because the length of the dependence cycle between the threads increase with the number of threads, as shown in the performance characteristics discussion of DOACROSS in Section 2.2.2 shows. The performance of speculative PS-DSWP continues to increase even when the TLS performance saturates since there is no cyclic communication.

6.7 Spice

Spice is also evaluated with the cycle-accurate simulation methodology described in Section 6.2.1. Spice is evaluated on the benchmarks in Table 6.5. These loops were chosen manually from a set of non-DOALL loops that are found to be good candidates for this technique. Among these loops, only those which account for significant fraction of the program's execution are retained.

Benchmark	Loop	Hotness
ks	FindMaxGpAndSwap (inner loop)	98%
otter	find_lightest_cl	20%
181.mcf	refresh_potential	30%
458.sjeng	std_eval	26%

Table 6.5: Details of the loops used to evaluate Spice.

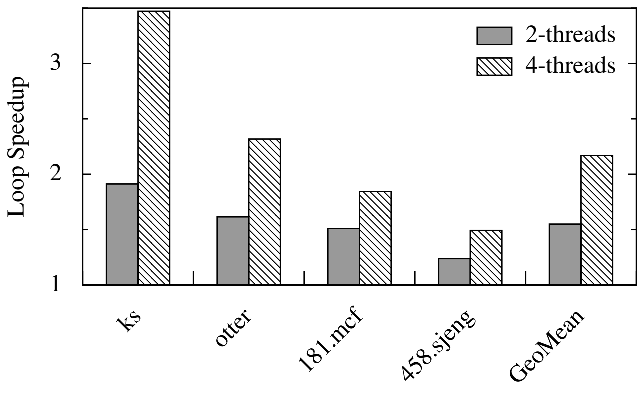


Figure 6.10: Performance improvement of Spice over single-threaded execution.

Figure 6.10 shows the speedups obtained on these four loops. The speedup numbers are shown for both the two threads and the four threads cases. The loop speedups range from 1.24 in `458.sjeng` with two threads to as high as 2.57 on the loop in `ks` with four threads. There are several factors that prevent the actual speedup from being in line with the ideal linear speedup expected from threads executing in a DOALL fashion:

Mis-speculations `458.sjeng` is the only benchmark in which performance suffers heavily due to mis-speculation. Around 25% of the invocations mis-speculate in this benchmark. In the other three loops, the mis-speculation rate is less than 1%.

Load imbalance The number of iterations per invocation varies in `otter` due to insertions to the linked list. Load balancing plays an important role in speeding up the loop's execution. Since the dynamic load balancing strategy does not exactly divide the work equally among the different cores, it results in some slowdown compared to the ideal case. In `458.sjeng`, even though the variance in the number of iterations per invocation is not very high, the actual number of instructions executed per iteration varies across iterations. A load balancing metric better than simple iteration counts would improve the speedup. Load balancing is also an issue in `181.mcf` due to the variability in the number of iterations of the inner loop per outer loop iteration.

Speculation overhead Even when there is no mis-speculation, there is an overhead in checking for mis-speculation every iteration. In `otter`, the time to execute the loop body per iteration is low and hence the overhead, as a fraction of useful execution time, is high. In `458.sjeng`, the overhead is high because there are 8 distinct live-in values that are compared with the memoized values of the next thread and ANDed together to determine whether the speculation is successful.

6.8 Summary

The results presented in this chapters validates the qualitative claims of performance advantages made in prior chapters. The techniques presented in this dissertation complement each other by efficiently parallelizing loops with different characteristics. This can be observed from the fact that there is little overlap between the set of loops that show significant performance improvement under each of the three techniques evaluated in this chapter.

Chapter 7

Future Directions and Conclusions

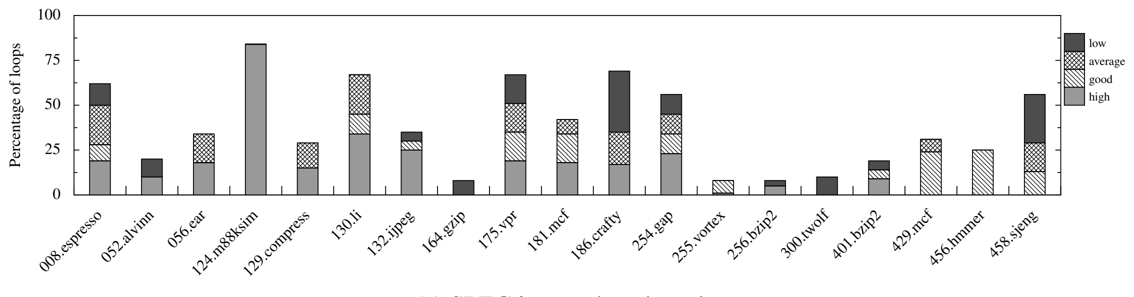
In the multi-core era, the responsibility for improving single-threaded application performance shifts to the programmer or the compiler. Leaving that responsibility with the programmer is likely to act as a hindrance to the goal of providing a rich end-user experience. Additionally, this solution does not address the large body of legacy sequential codes. By proposing two new parallelization transformations, this dissertation demonstrates that a compiler based approach could be a viable solution to the problem of improving single-threaded performance on multi-core processors.

7.1 Future Directions

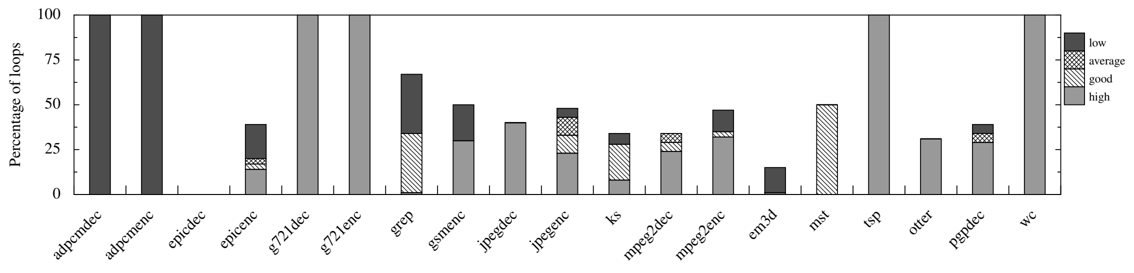
While the two transformations presented in this dissertation show significant advantages over related techniques on the selected benchmarks, there is significant scope for extending and improving them. This section discusses some future research directions related to the the two proposed transformations.

Decision Heuristics

Many heuristic based optimizations can potentially *increase* the execution time of a piece of code. PS-DSWP and Spice are no exceptions to this effect. Thus it is important to



(a) SPEC integer benchmarks



(b) Mediabench and others

Figure 7.1: Value predictability of loops in applications. Each loop is placed into 4 predictability bins (low, average, good and high) and the percentage of loops in each bin is shown.

develop heuristics to decide which loops to optimize, which transformation to apply and how much speculation to use. For a static compiler to perform these steps, it needs a good model to estimate the performance. While the performance model used by PS-DSWP thread partitioning algorithms is a good starting point, a more realistic processor model could significantly improve performance.

An alternative is to implement the transformations in a dynamic optimizer where a more realistic performance estimation can be made by monitoring the execution using hardware performance counters. This requires significant advancements in reducing the compilation time of these transformations.

Improved Exploitation of Value Predictability

The experimental evaluation of Spice in this dissertation indicates that its applicability is limited. However, analysis of value profiles shows that many applications exhibit a similar kind of predictability that Spice can exploit. The value profiler computes a hash of loop live-ins every iteration, after ignoring infrequent memory dependences. If more than half such hashes in an invocation are found in the preceding invocation, the loop live-ins of that invocation is deemed predictable by Spice. Each loop is then classified according to the percentage of its invocations that are predictable: low (1-25%), average (26-50%), good (51-75%), and high (76-100%). The number of loops in each benchmark that fall under each of these categories is shown in Figure 7.1. The results indicate that the Spice does not fully leverage the predictability of values across loop invocations.

The following extensions to Spice are required in order to deliver performance improvements from value predictability:

1. Supporting memory alias speculation in addition to value speculation is necessary to handle those infrequent inter-iteration memory dependences whose values do not show predictability across invocations. This could be accomplished by using either TLS-style hardware support for alias mis-speculation detection or an extension of the software based memory mis-speculation detection approach proposed by Vachharajani [72].

2. Most of the predictability shown in Figure 7.1 stems from inter-iteration memory dependences whose values are predictable across invocations. Unlike dependences through registers, the dependence distance of these memory dependences can be greater than 1 and hard to determine statically. The value prediction used in Spice does not directly work under these circumstances. Value prediction must be capable of handling varying dependence distances in order to realize performance gains.

Synergistic combination of PS-DSWP and Spice

Modern optimizing ILP compilers include a suite of optimizations, many of which interact in a synergistic fashion. A similar interaction between various parallelizing transformations

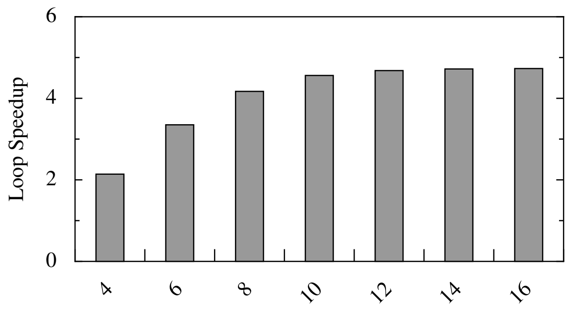


Figure 7.2: Speedup obtained by applying PS-DSWP to the inner loop in `FindMaxGpAndSwap` method from `ks`. Beyond eight threads, the sequential stage becomes the limiting factor.

may be essential to exploit parallelism across a wide spectrum of applications. The results in Section 6.5 demonstrate that the combination of speculative DSWP and PS-DSWP is better than the sum of the two individual components. Integrating Spice and PS-DSWP is an important area of future research.

One approach to combine Spice and PS-DSWP is to apply Spice to the sequential stages generated by PS-DSWP. The potential benefit of such a combination is demonstrated by the parallelization of the inner loop in the `FindMaxGpAndSwap` method from `ks`. The graph in Figure 7.2 shows the speedup obtained by applying PS-DSWP to that loop. This loop has a sequential first stage followed by a parallel stage. The performance due to PS-DSWP flattens after an eight thread parallelization. Adding more threads does not improve the performance since the sequential stage is the bottleneck. Since this sequential stage is a list traversal loop, Spice can be applied to parallelize that stage. Spice delivers a 1.45 speedup with two threads and a 2.2 speedup with four threads on the sequential stage. By reducing the sequential execution time using Spice, more threads can be used to execute the parallel stage, thereby increasing the overall speedup over single-threaded execution.

7.2 Summary and Conclusions

This dissertation presented two automatic parallelization transformations: PS-DSWP and Spice. Both of these techniques can extract parallelism from general purpose applications with complex control flow and irregular memory access patterns. PS-DSWP and Spice show promising performance advantages over existing techniques.

The implementation and applications of PS-DSWP are presented in this dissertation. PS-DSWP combines the pipelined parallelism of DSWP with the iteration-level parallelism of DOALL. It has improved applicability when compared to DOALL and improved performance when compared to DSWP. PS-DSWP results in a geometric mean loop speedup of 2.13 over single-threaded performance with five threads when applied to loops from a set of five benchmarks.

The applicability and performance of PS-DSWP improves significantly when combined with speculative DSWP [73]. The combined technique outperforms either of the techniques when applied alone. Speculative PS-DSWP results in a geometric mean loop speedup of 3.67 over single-threaded performance when applied to loops from a set of 6 benchmarks. A quantitative comparison of speculative PS-DSWP and TLS is presented demonstrating significant advantages of speculative PS-DSWP over TLS.

Spice speculatively extracts thread level parallelism by making use of a novel value prediction mechanism. The software value prediction mechanism is based on two new insights on the differences between value speculation for ILP and TLP. Spice is evaluated on a set of 4 loops from general-purpose applications with complex control flow and memory access patterns. Spice shows a geometric mean loop speedup of 2.01 over single-threaded performance on the set of loops to which it is applied.

Bridges *et al.* [6] showed that application performance can be brought back to the historical performance trendline using automatic parallelization. They proposed a framework that brings together various parallelization techniques, with a few simple source-level annotations, to improve the performance of SPEC CINT2000 benchmark suite by a factor of

5.54. This dissertation provides further support to the thesis that automatic parallelization is a viable and effective solution to the challenges posed by multi-core processors. The techniques presented in this dissertation may not by themselves be sufficient to extract scalable parallelism from a wide range of applications. However, as the integration of speculative DSWP and PS-DSWP suggests, a combination of various techniques capable of optimizing codes with different dependence patterns has the potential to unlock of parallelism across a wide range of applications.

Bibliography

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

[2] X. Berenguer and J. Diaz. The weighted sperner's set problem. In *Lecture Notes in Computer Science*, volume 88, pages 137–141. Springer-Verlag, 1980.

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

[4] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The next generation in parallelizing compilers. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, pages 10–1. Springer-Verlag, Berlin/Heidelberg, August 1994.

[5] M. J. Bridges. *The VELOCITY Compiler: Extracting Efficient Multicore Execution from Legacy Sequential Codes.* PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, November 2008.

[6] 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.

[7] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–13, New York, NY, USA, 2006. ACM Press.

[8] D.-K. Chen and P.-C. Yew. Redundant synchronization elimination for doacross loops. *IEEE Transactions on Parallel and Distributed Systems*, 10(5):459–470, 1999.

[9] 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.

[10] 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.

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

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

[13] F. H. Dang, H. Yu, and L. Rauchwerger. The R-LRPD test: Speculative parallelization of partially parallel loops. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 318, 2002.

[14] J. R. B. Davies. Parallel loop constructs for multiprocessors. Master's thesis, Department of Computer Science, University of Illinois, Urbana, IL, May 1981.

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

[16] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[17] 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.

[18] 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.

[19] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM Press.

[20] 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.

[21] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Interprocedural parallelization analysis in SUIF. *ACM Trans. Program. Lang. Syst.*, 27(4):662–731, 2005.

[22] 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, January 2000.

[23] 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.

[24] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Preliminary experiences with the fortran d compiler. 1993.

[25] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

[26] Intel Corporation. PRESS KIT – Moore's Law 40th Anniversary. `http://www.intel.com/pressroom/kits/events/moores_law_40th`.

[27] 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, June 2004.

[28] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

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

[30] 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.

[31] 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.

[32] 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.

[33] D. B. Loveman. High performance fortran. *IEEE Parallel and Distributed Technology*, pages 25–42, February 1993.

[34] 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.

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

[36] 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.

[37] 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.

[38] M. Mock, M. Das, C. Chambers, and S. J. Eggers. Dynamic points-to sets: a comparison with static analyses and potential applications in program understanding and optimization. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 66–72, New York, NY, USA, 2001. ACM.

[39] G. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), April 1965.

[40] The message passing interface (MPI) standard. http://www-unix.mcs.anl.gov/mpi/.

[41] T. Mudge. Power: A first-class architectural design constraint. *Computer*, 34(4):52–58, 2001.

[42] The OpenMP API specification for parallel programming. http://www.openmp.org.

[43] 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.

[44] 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.

[45] D. A. Padua. *Multiprocessors: Discussion of some theoretical and practical problems*. PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, United States, November 1979.

[46] D. A. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. I. August, and D. Connors. Exploiting parallelism and structure to accelerate the simulation of chip multi-processors. In *Proceedings of the Twelfth International Symposium on High-Performance Computer Architecture (HPCA)*, pages 29–40, February 2006.

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

[48] M. K. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, New York, NY, USA, 2003. ACM Press.

[49] M. K. Prabhu and K. Olukotun. Exposing speculative thread parallelism in SPEC2000. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 142–152, New York, NY, USA, 2005. ACM Press.

[50] R. Rajamony and A. L. Cox. Optimally synchronizing doacross loops on shared memory multiprocessors. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, 1997.

[51] R. Rajwar and J. R. Goodman. Transactional execution: Toward reliable, high-performance multithreading. *IEEE Micro*, 23(6):117–125, Nov-Dec 2003.

[52] 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*, April 2008.

[53] E. Raman, N. Vachharajani, R. Rangan, and D. I. August. Spice: Speculative parallel iteration chunk execution. In *Proceedings of the 2008 International Symposium on Code Generation and Optimization*, 2008.

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

[55] 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.

[56] 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.

[57] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, 1999.

[58] 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.

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

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

[61] Stanford Compiler Group. SUIF: A parallelizing and optimizing research compiler. Technical Report CSL-TR-94-620, Stanford University, Computer Systems Laboratory, May 1994.

[62] J. G. Steffan. *Hardware Support for Thread-Level Speculation*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, United States, April 2003.

[63] 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, February 2005.

[64] 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.

[65] M. Takabatake, H. Honda, and T. Yuba1. Performance measurements on sandglass-type parallelization of doacross loops. In *Lecture Notes in Computer Science*, volume 1593, pages 663–672. Springer-Verlag, 1999.

[66] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

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

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

[69] 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.

[70] 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.

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

[72] N. Vachharajani. *Intelligent Speculation for Pipelined Multithreading*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, November 2008.

[73] 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.

[74] M. Wolfe. Loop rotation. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, pages 531–553, 1990.

[75] 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.

[76] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 171–183, October 2002.

[77] H. Zhong, S. A. Lieberman, and S. A. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. 2007.

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