THE VELOCITY COMPILER: EXTRACTING EFFICIENT MULTICORE EXECUTION FROM LEGACY SEQUENTIAL CODES

MATTHEW JOHN BRIDGES

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: PROF. DAVID I. AUGUST

NOVEMBER 2008

Abstract

Multiprocessor systems, particularly chip multiprocessors, have emerged as the predominant organization for future microprocessors. Systems with 4, 8, and 16 cores are already shipping and a future with 32 or more cores is easily conceivable. Unfortunately, multiple cores do not always directly improve application performance, particularly for a single legacy application. Consequently, parallelizing applications to execute on multiple cores is essential.

Parallel programming models and languages could be used to create multi-threaded applications. However, moving to a parallel programming model only increases the complexity and cost involved in software development. Many automatic thread extraction techniques have been explored to address these costs.

Unfortunately, the amount of parallelism that has been automatically extracted using these techniques is generally insufficient to keep many cores busy. Though there are many reasons for this, the main problem is that extensions are needed to take full advantage of these techniques. For example, many important loops are not parallelized because the compiler lacks the necessary scope to apply the optimization. Additionally, the sequential programming model forces programmers to define a single legal application outcome, rather than allowing for a range of legal outcomes, leading to conservative dependences that prevent parallelization.

This dissertation integrates the necessary parallelization techniques, extending them where necessary, to enable automatic thread extraction. In particular, this includes an expanded optimization scope, which facilitates the optimization of large loops, leading to parallelism at higher levels in the application. Additionally, this dissertation shows that many unnecessary dependences can be broken with the help of the programmer using natural, simple extensions to the sequential programming model.

Through a case study of several applications, including several C benchmarks in the SPEC CINT2000 suite, this dissertation shows how scalable parallelism can be extracted.

By changing only 38 source code lines out of 100,000, several of the applications were parallelizable by automatic thread extraction techniques, yielding a speedup of 3.64x on 32 cores.

Acknowledgments

First and foremost, I would like to thank my advisor, David August, for his assistance and support throughout my years in graduate school at Princeton. He has taught me a great deal about research, and I would not be here if not for his belief in me. Additionally, the strong team environment and investment in infrastructure that he has advocated has been a wonderful benefit. His continued expectation that I work to the best of my abilities has lead to this dissertation, and for that I am grateful.

This dissertation was also made possible by the support of the Liberty research group, including Ram Rangan, Manish Vachharajani, David Penry, Thomas Jablin, Yun Zhang, Bolei Guo, Adam Stoler, Jason Blome, George Reis, and Jonathan Chang. Spyros Triantafyllis, Easwaran Raman, and Guilherme Ottoni have contributed greatly to the VELOCITY compiler used in this research and have my thanks for both their friendship and support. Special thanks goes to Neil Vachharajani, both for the many contributions to VELOCITY and this work, as well as for the many interesting discussions we have had over the years.

I would like to thank the members of my dissertation committee, Kai Li, Daniel Lavery, Margaret Martonosi, and Doug Clark. Their collective wisdom and feedback have improved this dissertation. In particular, I thank my advisor and my readers, Kai Li and Daniel Lavery, for carefully reading through this dissertation and providing comments. Additionally, I thank my undergraduate research advisor, Lori Pollock, without whom I would not have entered graduate research.

This dissertation would not be possible without material research support from the National Science Foundation, Intel Corporation, and GSRC. Special thanks also goes to Intel for summer internship that they provided and the guidance given to me by Daniel Lavery and Gerolf Hoflehner.

Finally, I thank my friends and family. I thank my mother Betty, my father John, my step-father Doug, my mother-in-law Janet, and my father-in-law Russell, for having supported me throughout this process with their love and encouragement. Above all, my wife Marisa has my sincere thanks for putting up with everything involved in creating this dissertation and for reading early drafts. She has seen me through the good times and the bad and I would not have accomplished this with out her.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Until recently, increasing uniprocessor clock speed and microarchitectural improvements generally improved performance for all programs. Unfortunately, as Figure 1.1 illustrates, since approximately 2004, the performance of single-threaded applications is no longer increasing at the same rate as it was prior to 2004. This change came about because of several factors, though two predominant factors stand out. First, processor designers were unable to increase processor frequency as they had in the past without exceeding power and thermal design constraints. Second, even with a steady increase in transistors (Figure 1.2), processor designers have also been unable to create new microarchitectural innovations without exceeding power and heat budgets.

Instead, processor designers now use the continuing growth in transistor count to place multiple cores on a processor die. This strategy has several microarchitectural benefits, including more scalable thermal characteristics, better throughput per watt, and greatly reduced design complexity. Because of this, processor manufacturers are already shipping machines with four or eight cores per processor die and tomorrow's machines promise still more cores [36]. However, additional cores only improve the performance of multi-threaded applications. The prevalence of single-threaded applications means that these cores often provide no performance improvement, as shown in the post-2004 portion of

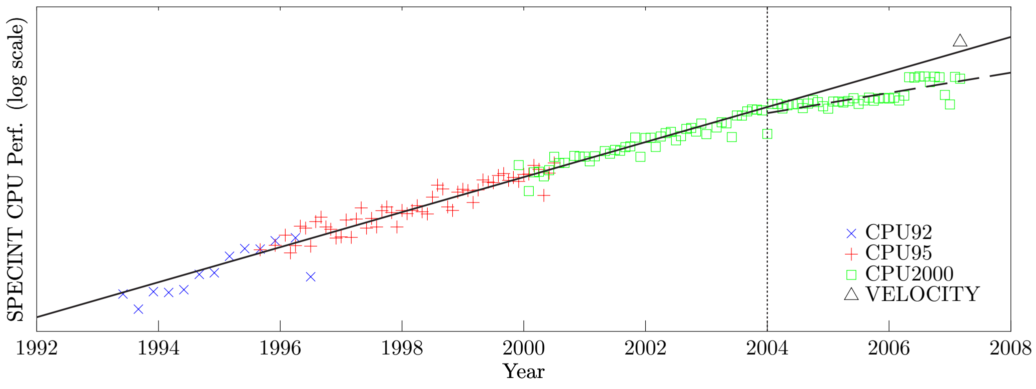


Figure 1.1: Normalized performance of the SPEC 92, 95, and 2000 integer benchmarks suites. Each version of the suite is normalized to the previous version using matching hardware. Only the highest performance result for each time period is shown. The solid line is a linear regression on all points prior to 2004, while the dashed line a linear regression on all points in or after 2004. Source data from SPEC [4].

Figure 1.1.

One potential solution to achieving performance that gets back to pre-2004 historical trends is to rewrite single-threaded applications as multi-threaded applications. Many new languages have been proposed to ease the burden of writing parallel programs [12, 26, 29]. While parallel programming in these languages is easier than those of the past, the programming effort involved in creating correct and efficient parallel programs is still far more than that of writing the equivalent single-threaded version. Developers must be trained to program and debug their applications with the additional concerns of deadlock, livelock, race conditions, etc [16, 20, 23, 47, 68]. Converting a single-threaded application is considerably more difficult than writing a new parallel application, as existing single-threaded applications have not been developed with parallelization in mind. Therefore, even as multi-core processors have become commonplace, the costs associated with parallel programming languages have greatly limited the programming domains to which they are applied.

Automatic parallelization techniques that extract threads from single-threaded programs without programmer intervention do not suffer from these limitations and should be preferred over manual parallelization. Many automatic thread extraction techniques have been

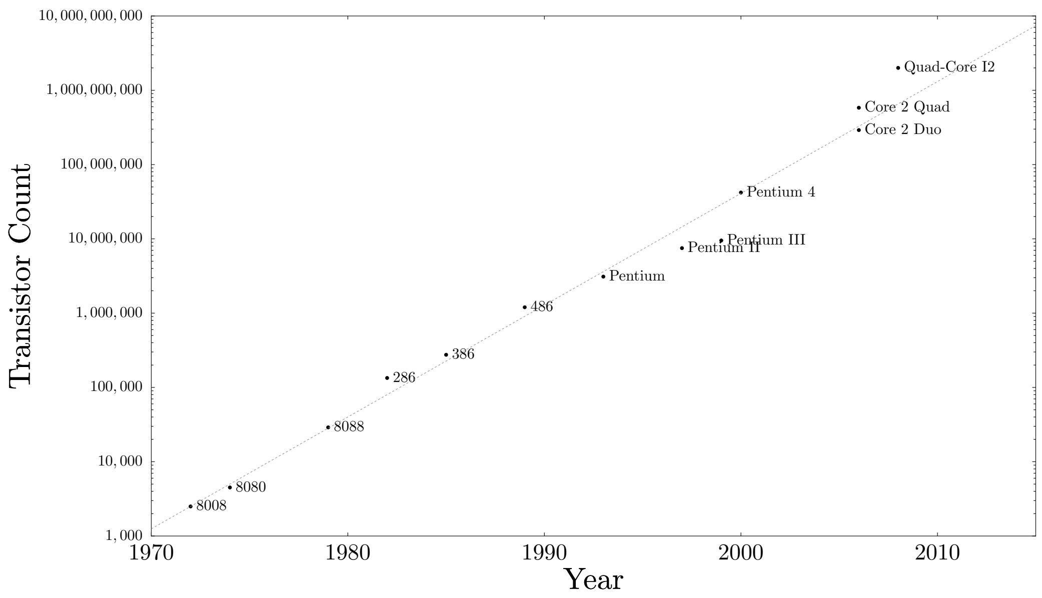


Figure 1.2: Transistor counts for successive generations of Intel processors. Source data from Intel [37].

proposed, particularly loop parallelization techniques such as DOALL [48], DSWP [55], and DOACROSS [72] (Chapter 2). In isolation, these techniques produce either scalable parallelism on a narrow range of applications (e.g. DOALL on scientific applications), or nonscalable parallelism on a broad range of applications (e.g. DOACROSS & DSWP). This dissertation focuses on the problems that have prevented the automatic extraction of scalable parallelism in general purpose applications and constructs a parallelization framework in the VELOCITY compiler to overcome these limitations.

Achieving scalable parallelism on a broad range of applications first requires an automatic parallelization technique with broad applicability. To that end, this dissertation starts with the DSWP technique. However, the base parallelization technique can extract only small amounts of parallelism and must be extended with the ability to speculate dependences and extract data-level parallelism. This dissertation integrates and extends several existing DSWP based parallelization techniques that have not previously been combined to achieve this (Chapter 3).

Beyond combining existing extensions, it is also necessary to give the parallelization technique interprocedural scope [9, 32]. Existing DOALL parallelization techniques use interprocedural array analysis, but have not needed other interprocedural analysis or the ability to optimize interprocedurally. Existing DOACROSS and DSWP techniques rely on inlining function calls to obtain loops with sufficient opportunity for parallelization. Unfortunately, the limitations of inlining, its inability to handle recursion and its explosive code growth, make it sufficient to extract parallelism only from inner loops or functions at the bottom of the call tree. To extract more scalable parallelism, this dissertation also develops a parallelization technique that can target any loop in the program, from innermost to outermost or anywhere in between (Chapter 3).

Additional support, in the form of accurate compiler analysis, aggressive compiler optimization, and efficient hardware support, ensures that the parallelization technique can extract significant parallelism. In particular, effectively applying additional optimizations during parallelization, particularly for synchronization placement [54, 90] and speculation [72, 82], requires accurate profile information about the loop being parallelized. Traditional profiling infrastructures, however, focus on profiling with respect to instructions, or basic blocks, not loops. This limits the applicability of loop parallelization techniques by conflating intra-iteration dependences with inter-iteration dependences, or worse with dependences that occur outside the loop.

Finally, even many apparently easily parallelizable programs are unable to be parallelized automatically. The dependences that inhibit parallelism are only rarely those required for correct program execution, more commonly manifesting from artificial constraints imposed by sequential execution models. In particular, the programmer is often unable to specify multiple legal execution orders or program outcomes. Because of this, the compiler is forced to maintain the single execution order and program output that a sequential program specifies, even when others are more desirable. This dissertation proposes two simple sequential annotations to help the compiler overcome this limitation through programmer intervention (Chapter 4).

Using the VELOCITY compiler, scalable parallelism can be extracted from many applications, yet simulating long running parallel loops can be hard. Existing parallelization techniques have generally been evaluated using only 2 or 4 processors on loops with iterations that take fewer than a million cycles, making them amenable to simulation. When extracting scalable parallelism, it is not uncommon for each iteration to take billions of cycles and for such parallelism to scale to 32 processors or beyond. As such, existing simulation environments are insufficient and a new evaluation infrastructure is required that can measure the parallelism extracted from outermost loops (Chapter 5).

In summary, the contributions of this dissertation are:

1. The design and implementation of a novel parallelization framework for general-purpose programs, VELOCITY, that can address the performance problem of legacy sequential codes on modern, multi-core processors.

2. The design and implementation of a loop-aware profiling infrastructure and integration of several types of loop-aware profile information into the parallelization technique.

3. Extensions to the sequential programming model that enhance VELOCITY's ability to extract parallelism *with out* forcing the programmer to consider multi-threaded concepts or execution.

4. The design and implementations of a performance evaluation methodology able to handle parallelized loops with up to 32 threads that execute for billions of cycles.

5. An evaluation of the VELOCITY compiler on several applications in the SPEC CINT2000 benchmark suite.

By bringing together these many techniques and extensions, both to compiler and programming languages, this dissertation shows that parallelism can be extracted from general-purpose applications. Using VELOCITY, seven benchmarks in the SPEC CINT2000 benchmark suite are automatically parallelized. Modifying only 38 out of 100,000 lines of code, these applications show a 3.64x speedup, on up to 32 cores, over single-threaded execution, with a 20.61x maximum speedup on a single application. Overall, a 2.12x speedup was achieved across the 12 benchmarks of the SPEC CINT2000 suite, even with no current speedup on 5 of these benchmarks. The *VELOCITY* point in Figure 1.1 indicates the expected SPEC CINT2000 score that could be obtained with VELOCITY, showing that the performance trends of the past can be obtained now and in the future via automatic parallelization.

Chapter 2

Automatic Parallelization

This chapter explores existing techniques in automatic parallelization of loops, discussing their benefits and limitations. Techniques that spawn speculative threads along multiple acyclic paths [5, 85] or to better utilize microarchitectural structures [6, 13, 15, 51, 60] are orthogonal to this work and will not be discussed further.

2.1 Extracting Parallelism

There are three types of parallelism that can be extracted from loops [62]. Each type allows or disallows certain cross-thread dependences to trade off applicability with performance.

2.1.1 Independent Multi-Threading

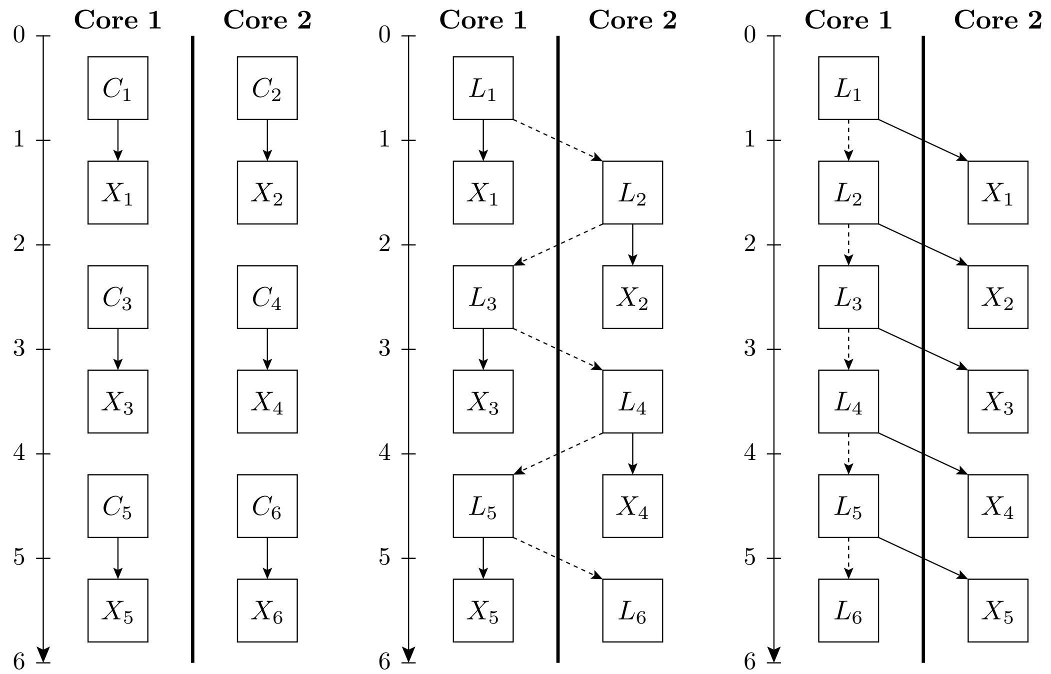
The first type of parallelism, Independent Multi-Threading (IMT), does not allow cross-thread dependences and was first introduced to parallelize array-based scientific programs The most successful IMT technique, DOALL parallelization [48], satisfies the condition of no inter-thread dependences by executing loop iterations in separate threads that it can prove have no inter-iteration dependences. A primary benefit of IMT parallelism is that it scales linearly with number of available threads. In particular, this scalability has lead to

```
   int i=0;
C: for (; i<N; i++)
X:   work(i);
```

(a) Array Traversal

```
   list *cur=head;
L: for (; cur!=NULL; cur=cur->next)
X:   work(cur);
```

(b) Linked List Traversal



(c) **IMT** Execution Example (d) **CMT** Execution Example (e) **PMT** Execution Example

Figure 2.1: Parallelization Types

the integration of DOALL into several compilers that target array-based, numerical programs [8, 32].

To see how DOALL would parallelize a loop, consider the loop in Figure 2.1a. When parallelized into two threads, one potential dynamic execution is shown in Figure 2.1c. Since this parallelization requires no communication between cores, it is insensitive to the communication latency between cores and can scale to as many cores as there are iterations. In the case of Figure 2.1c, this means that the overall speedup of the loop is 2.

IMT's requirement that there are no cross-thread dependences within a loop is both a source of scalable parallelism and limited applicability. While IMT techniques have enjoyed broad applicability for scientific applications, they have not been used to parallelize general purpose applications, which have many inter-iteration dependences. These dependences arise both from complicated control flow and the manipulation of large, heap-allocated data structures, particularly recursively defined data structures.

2.1.2 Cyclic Multi-Threading

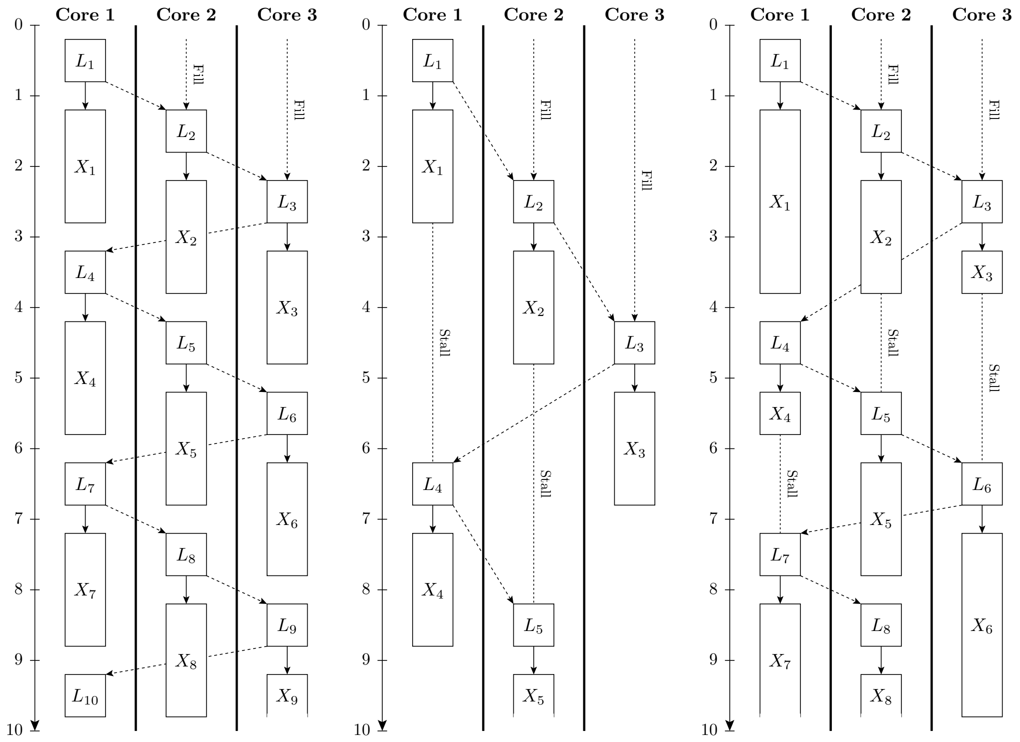
Cyclic Multi-Threading (CMT) techniques, which allow cross-thread dependences to exist, have been developed to solve the problem of applicability. To maintain correct execution, dependences between threads are synchronized so that each thread executes with the proper values. Though initially introduced to handle array-based programs with intricate access patterns, CMT techniques can also be used to extract parallelism from general purpose applications with complex dependence patterns.

The most common CMT transformation is DOACROSS [18, 57], which achieves parallelism in the same way as DOALL, by executing iterations in parallel in separate threads. To maintain correct execution, the inter-iteration dependences that cross between threads must be respected. These dependences are synchronized to ensure that later iterations receive the correct values from earlier iterations.

An example of a DOACROSS execution schedule is shown in Figure 2.1d for the loop in Figure 2.1b. Notice that DOACROSS can extract parallelism even in the face of the `list=list->next` loop-carried dependence. For the example shown, DOACROSS achieves an ideal speedup of 2 when the `work` code takes the same amount of execution time as the linked list traversal.

Unlike IMT transformations, CMT transformations do not always experience linearly increasing performance as more threads are added to the system. In particular, when the threads begin completing work faster than dependences can be communicated, the critical path of a DOACROSS parallelization becomes dependent upon the communication latency between threads. Because of this, DOACROSS parallelizations are fundamentally limited in speedup by the size of the longest cross-iteration dependence cycle and the communication latency between processor cores [45, 73].

There are often many inter-iteration, cross-thread dependences in general purpose programs, which leaves little code to execute in parallel. Synchronizing these dependences only decreases the number of threads where communication latency will become the limiting factor. Additionally, variability in iteration execution time in one iteration can potentially cause stalls in every other thread in the system, further adding to the critical path and limiting the attainable speedup.


Core 1
Core 2
Core 3
0
1
2
3
4
5
6
7
8
9
10
Fill
Stall
L_1 L_2 L_3 L_4 L_5 L_6 L_7 L_8 L_9 L_{10}
X_1 X_2 X_3 X_4 X_5 X_6 X_7 X_8 X_9

(a) Comm. Latency = 1, No Variability
(b) Comm. Latency = 2, No Variability
(c) Comm. Latency = 1, Variability

Figure 2.2: CMT Execution Examples

Figure 2.2 illustrates these problems. The parallelization shown in the CMT column of Figure 2.1 assumed a 1 cycle communication latency between cores and no variability in the time to perform work. Figure 2.2a shows a parallelization of the same code to 3 threads, assuming `work` now takes 2 cycles per iteration, completing an iteration every cycle. As the communication latency increases from 1 to 2 cycles, an iteration is completed every other cycle, as shown in Figure 2.2b. Additionally, as variability occurs in the time to perform `work`, stall cycles appear on the critical path and iterations are not completed every cycle, as shown in Figure 2.2c.

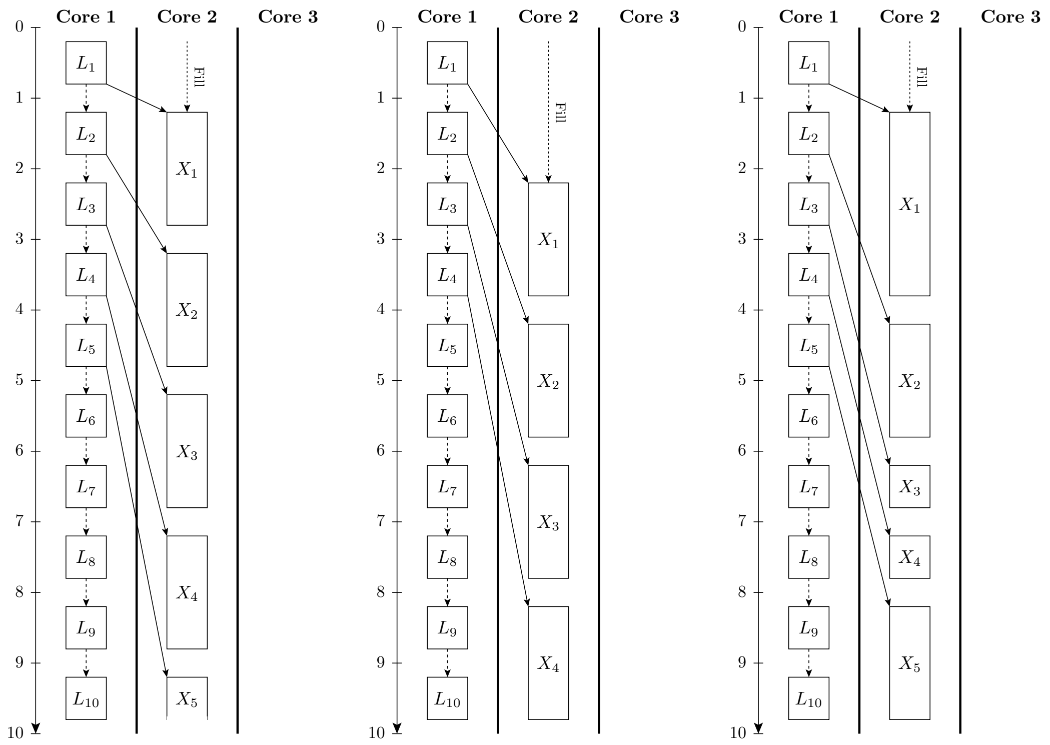
2.1.3 Pipelined Multi-Threading

In order to achieve broadly applicable parallelizations that are not limited by communication latency, a parallelization paradigm is needed that does not spread the critical path of a loop across multiple threads. Pipelined Multi-Threading (PMT) techniques have been proposed that allow cross-thread dependences while ensuring that the dependences between threads flow in only one direction. This ensures that the dependence recurrences in a loop remain local to a thread, ensuring that communication latency does not become a bottleneck. This paradigm also facilitates a decoupling between earlier threads and later threads that can be used to tolerate variable latency. The most common PMT transformation is Decoupled Software Pipelining (DSWP) [55, 63], an extension of the much earlier DOPIPE [19] technique, allowing it to handle arbitrary control flow, complicated memory accesses, and all other features of modern software.

As in CMT techniques, DSWP synchronizes dependences between threads to ensure correct execution. However, instead of executing each iteration in a different thread, DSWP executes portions of the static loop body in parallel, with dependences flowing only in a single direction, forming a pipeline.

Figure 2.1e shows an example of a DSWP parallelization executing the loop in Figure 2.1b. In the example the first thread executes the pointer chasing code, while the second thread performs the necessary work on each list element. However, the dependence recurrence formed by the pointer chasing code is executed only on the first core, keeping it thread local. The `work` portion of the loop executes in parallel on the other core, allowing the parallelization to achieve an ideal speedup of 2, assuming the list traversal and `work` have equal weight.

DSWP ensures that all operations in a dependence cycle execute in the same thread, removing communication latency from the critical path. This has the additional benefit of ensuring that dependences flow in only one direction, which allows DSWP to use communication queues [63] to decoupled execution between stage of the pipeline that can be used to tolerate variable latency.



(a) Comm. Latency = 1, No Variability

(b) Comm. Latency = 2, No Variability

(c) Comm. Latency = 1, Variability

Figure 2.3: PMT Execution Examples

Figure 2.3 illustrates these properties for the loop in Figure 2.1b. Figure 2.3a illustrates a DSWP execution similar to the one for the PMT column in Figure 2.1e, except that the `work` now takes 2 cycles. For 3 threads, DSWP is only able to extract a 2-stage, 2-thread pipeline, leaving one thread unused. However, increasing the communication latency, shown in Figure 2.3b, only increases the pipeline fill time. The steady state parallelism is unaffected, leading to an equivalent speedup as in Figure 2.3a. Because dependence recurrence along the loops critical path are not spread across cores, variability also does not dramatically affect the speedup of a DSWP partition, as shown in Figure 2.3c.

The speedup of a DSWP parallelization is limited by the size of the largest pipeline stage. Because DSWP extracts parallelism among the static operations of a loop and must keep dependence recurrences local to a thread, the largest pipeline stage is limited by the largest set of intersecting dependence cycles in the static code. In practice, this means that DSWP can often scale to only a few threads before performance no longer increases with additional threads [61].

2.2 Enhancing Parallelism

Not surprisingly, many techniques have been proposed to overcome the limitations that current parallelization techniques face. This subsection discusses the most prominent techniques for each type of parallelism.

2.2.1 Enhancing IMT Parallelism

Because IMT techniques do not allow inter-iteration, cross-thread dependences to exist, extensions to IMT techniques have focused on turning inter-iteration, cross-thread dependences into thread-local dependences. In particular, many techniques have extended the base DOALL technique to remove inter-iteration dependences, either through more accurate analysis or program transformations.

Reduction Expansion

Dependences that form specific patterns are amenable to reduction transformations [49, 56, 64]. Though often used to expose instruction level parallelism for single-thread scheduling, the same techniques hold merit when extracting thread level parallelism. These transformations take accumulators, min/max reductions, etc., and make inter-iteration dependences thread-local by creating local copies of the variable. The correct value is then

```
for (int i=0; i<N; i++)
  sum += A[i];
```

(a) Reduction Code

```
int _sum[THREADS] = {0,...,0}
for (int i=0; i<N; i++)
  _sum[TID] += A[i];

for (int i=0; i<THREADS; i++)
  sum += _sum[i];
```

(b) Expanded Reduction

Figure 2.4: Reduction Expansion Example

calculated via appropriate cleanup code at the end of the loop's invocation.

Figure 2.4 shows an example of reduction expansion applied to a simple array traversal. The unoptimized code in Figure 2.4a summarizes the elements of an array `A`. The cross-iteration dependence on the `sum` variable make a DOALL parallelization impossible. However, since the expression computing `sum` is a reduction, it can be expanded to produce the code in Figure 2.4b. In the expanded version, each thread computes a thread-local value for `sum` in the `_sum` array. After the loop executes the `_sum` is reduced into the `sum` variable. So long as the number of iterations is much greater than the number of threads, the execution time of the reduction code run after the loop finished is negligible.

Privatization

```
for (int i=0; i<N; i++) {

  for (int j = 0; j < 10; j++)
    B[j] = A[i] * j;

  C[i] = max(B, 0, 10);
}
```

(a) Array Code

```
for (int i=0; i<N; i++) {
  int _B[10];
  for (int j = 0; j < 10; j++)
    _B[j] = A[i] * j;

  C[i] = max(_B, 0, 10);
}
```

(b) Privatized Array Code

Figure 2.5: Array Privatization Example

In array-based programs, the same static variable or array is often referenced in multiple iterations, but can be proven, via compiler analysis, to always be written in the current iteration. Lacking separate address spaces, the compiler can *privatize* the variable or array, conceptually creating separate variables or arrays via memory renaming [32, 41, 42, 81]. Since each thread references a thread-local variable or array as opposed to a global variable or array, there are no inter-iteration, and thus cross-thread, false memory dependences to synchronize, enabling better parallelization.

Figure 2.5 shows an example of array privatization applied to a simple array traversal. Without privatization, updates to to the `B` array in different iterations of the outer loop would be forced to execute sequentially to ensure that earlier iterations do not overwrite values already written by later iterations. However, since every read from the `B` array is preceded by a write in the same iteration, the compiler can create an iteration-local version of the array. This ensures that there is no potential for other iterations to write to the current iteration's version of `_B`, removing the false memory dependences.

Loop Transformations

```
for (int i=1; i<N; i++) {
  A[i] = A[i-1];
  B[i] = f(A, 0, i);
}
```

(a) Array Code

```
for (int i=1; i<N; i++)
  A[i] = A[i-1];
for (int i=1; i<N; i++)
  B[i] = f(A, 0, i);
```

(b) Distributed Code

Figure 2.6: Loop Transformation Example

Should a inter-iteration dependence manifest itself even in the face of other techniques, it is sometimes possible to change the loop so as to facilitate parallelization [38, 42]. One particular transformation is to split the portion of the loop that contains the dependence into a separate loop via Loop Distribution (aka. Loop Fission). In this case, Loop Distribution creates several loops with each set of intersecting, inter-iteration dependence cycles isolated into a separate loop that must execute sequentially. The remaining code is formed into loops that can execute in parallel. This technique is generally limited to loops that operate on statically sized arrays, as the compiler must be able to compute amount of state needed to hold temporary values. Other transformations rearrange the loop hierarchy (Loop Inter-change), fuse loops to create larger iterations (Loop Fusion), unroll loop iterations (Loop Unrolling), etc., in order to create more optimal loops to parallelize.

Figure 2.6 shows an example of Loop Distribution applied to a simple array traversal. The base code in Figure 2.6a has a loop-carried dependence on the update to the elements of the `A` array. However, the `A` array updates can be isolated to their own loop that executes sequentially before the original loop. The code to update the `B` array can then execute in a parallelelized loop as there are no loop-carried dependences on the calculation used to determine its value.

2.2.2 Enhancing CMT Parallelism

Even with the extensions that have been proposed for IMT techniques, it is usually impossible to extract purely DOALL parallelism from general purpose programs. However, the extensions lend themselves just as readily to CMT techniques, such as DOACROSS, as they do to IMT techniques. Using these extensions to break dependences expands the size of parallel regions and reduces the amount of synchronization. CMT-specific extensions to the DOACROSS technique focus on dealing with cross-thread dependences that cannot be removed by IMT extension techniques.

In particular, IMT techniques still fail to alleviate late-iteration synchronization points, which can greatly limit the speedup of CMT parallelizations. Figure 2.7c shows an example of this problem, with the `printf` call synchronizing two otherwise parallelizable bodies of work. The `printf` dependence is typical of dependences in general purpose code which can not be removed. However, several extensions to CMT techniques have been proposed to help deal with these dependences by limiting their influence.

Scheduling Dependences

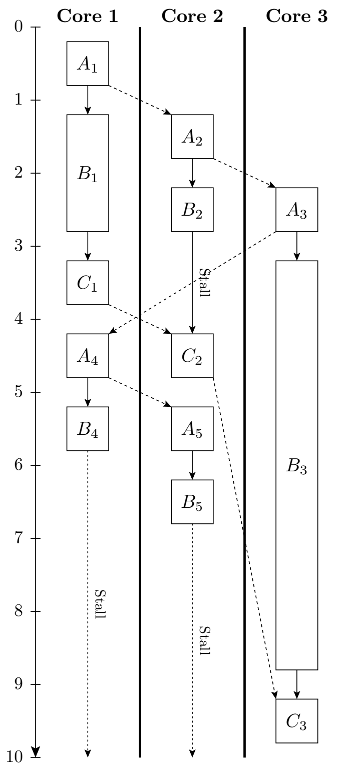
The size of regions executing in parallel directly affects the amount of speedup obtained. As a natural consequence of this, if a dependence that causes synchronization in two otherwise

```
   list *cur=head
A: for(; cur!=NULL; cur=cur->next) {
B:   work(cur);
C:   printf(cur);
   }
```

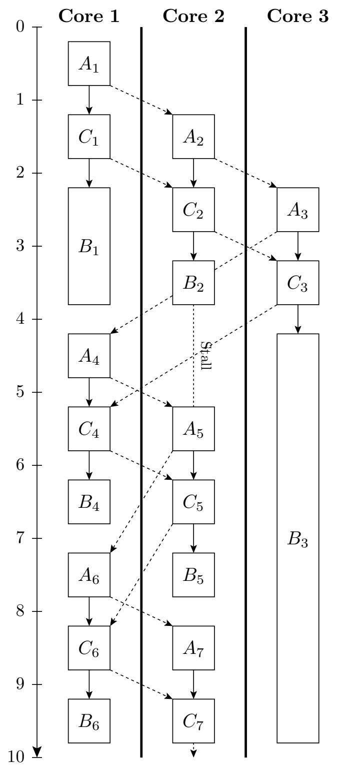
(a) List Code

```
   list *cur=head
A: for(; cur!=NULL; cur=cur->next) {
C:   printf(cur);
B:   work(cur);
   }
```

(b) Scheduled List Code



(c) Execution of Code in Figure 2.7a



(d) Execution of Code in Figure 2.7b

Figure 2.7: Scheduling Example

unsynchronized regions can be scheduled above or below the regions, then there is more potential for parallel execution. This observation is particularly useful for CMT techniques like DOACROSS [7, 18, 45, 57, 90], where scheduling synchronization to occur either earlier or later increases the size of regions that can execute in parallel.

Figure 2.7 illustrates the value of scheduling. Before scheduling, the code in Figure 2.7a is limited by the `printf` loop-carried dependence that must be synchronized. However, with scheduling, the compiler can prove that it is legal to move this dependence to the top of the iteration. This has the benefit of significantly reducing the size of the region that DOACROSS must synchronize to ensure correct execution. For Figure 2.7a scheduling moves the `printf` loop-carried dependence from the bottom of the iteration to the top, as shown in Figure 2.7b. This scheduling transformation reduces the time each thread spends waiting for communication, reducing the length of the loop's critical path.

Dynamic Work Allocation

DOACROSS statically schedules each iteration to a specific processor, which is only optimal when there is no variation in runtimes between iteration. Variance can cause a single thread can to become the bottleneck even though free processors exist. Dynamically spawning threads onto idle processors [28] allows the processors to greedily consume work as it becomes available rather than waiting for a statically determined iteration, more evenly balancing the available work, leading to more parallelism.

Figure 2.7d illustrates dynamic scheduling and instruction scheduling combining to produce better execution time than the base DOACROSS technique. Instruction scheduling moves the `print` dependence cycle to the top of the loop, allowing it to be synchronized before the variable length `work` is executed. Because of this, processors do not wait for synchronization and can dynamically execute the next ready iteration.

2.2.3 Enhancing PMT Parallelism

The techniques used for IMT parallelizations are also applicable to PMT parallelizations when they can break dependence recurrences. Unfortunately, the dependences recurrences that these techniques break tend not to break large stages into smaller stages.

Parallel Stage

Rather than break large stages into smaller stages, it is sometimes possible to improve the speedup of a DSWP parallelization by replicating the largest stage. A DSWP parallelization is limited by the size of the largest stage because DSWP extracts only parallelism among the static code of a loop. Large stages tend to be the result of inner loops whose static code cannot be split among multiple pipeline stages. However, a pipeline stage that contains no outer loop-carried dependence can be execute multiple outer loop iterations in parallel. The *Parallel-Stage DSWP* (PS-DSWP) extension to DSWP uses this insight to achieve data-level parallelism for such stages [57, 61, 75], essentially extracting IMT parallelism for that stage. For a sufficient number of threads, PS-DSWP can be used to reduce the runtime of a replicable stage to the length of the longest single dynamic iteration.

Figure 2.8b shows a PS-DSWP parallelization for the code in Figure 2.8a. In this example, the stage containing the call to work can be replicated. Conceptually, this is equivalent to unrolling the loop by the replication factor, as shown in Figure 2.8b. All operations corresponding to the non-replicable *A* stage remain in a loop-carried dependence recurrence. However, there now exist two instances of the *B* stage that can be scheduled to separate threads, increasing parallelism.

Dynamic Parallel Stage

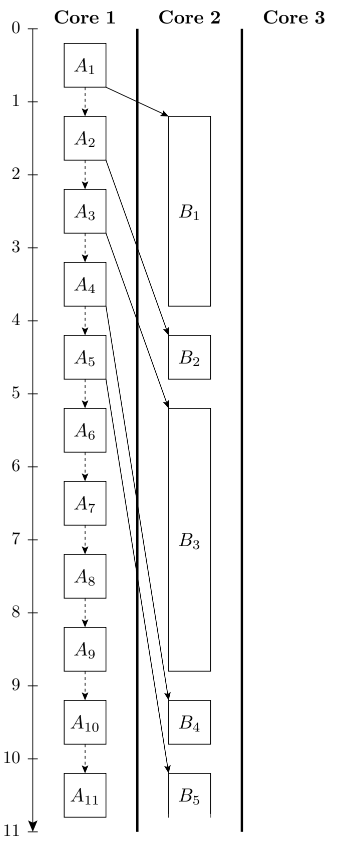
PMT techniques, like CMT techniques, benefit from dynamic allocation of work to threads. In particular, PS-DSWP can be extended so that each thread executing a replicated stage dynamically pulls work off of a master work queue. Figure 2.8e illustrates the execution of

```
   list *cur=head
A: for(; cur!=NULL; cur=cur->next) {
B:   work(cur);
   }
```

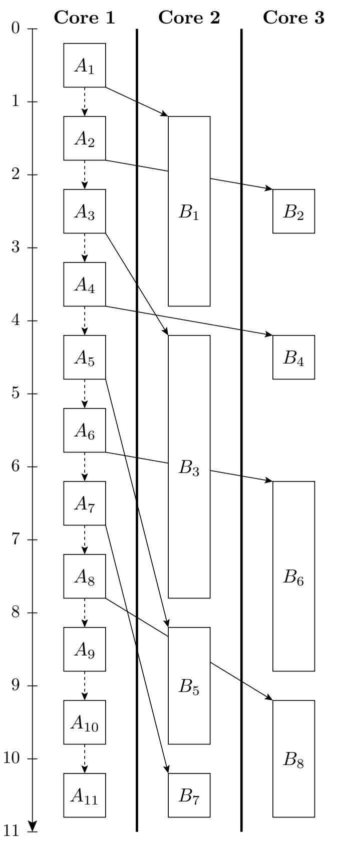
(a) List Code

```
   list *cur=head
A: for(; cur!=NULL; cur=cur->next) {
B:   work(cur);
A:   cur=cur->next; if(!cur) break;
B:   work(cur);
   }
```

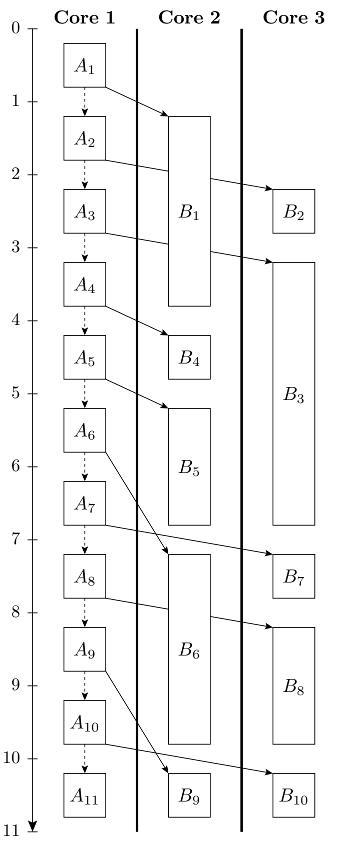
(b) Unrolled List Code



(c) Execution of Code in Figure 2.8a



(d) Execution of Code in Figure 2.8b with Static Thread Assignment



(e) Execution of Code in Figure 2.8b with Dynamic Thread Assignment

Figure 2.8: Parallel Stage Example

a PS-DSWP parallelization that dynamically assigns work to threads.

2.3 Breaking Dependences with Speculation

The success of automatic parallelization techniques is inherently limited by dependences in the region being parallelized. Since the compiler must be conservatively correct with respect to all possible dependences, it will often be limited by dependences that cannot, will not, or rarely manifest themselves dynamically. This section explores several types of speculation and their use in breaking problematic dependences to enhance parallelism.

2.3.1 Types of Speculation

Many variables in a program have predictable values at specific points in the program [58]. In particular, variables that do not change in value, variables set by the command line, and variables that have a constant update are easily predictable. *Value Speculation* [46, 59] removes dependences using a predicted value instead of the actual value to satisfy the dependence. Whenever the dependence executes and the actual value is not the predicted value, misspeculation is flagged [28, 82].

The most common type of value speculation predicts that a biased branch will not take a specific path(s) [82]. In particular, this type of speculation is often used to remove loop exits and branches that control programmer assertions and other error checking code, as these branches control, either directly or indirectly, the rest of the loop. This leads them to create long loop-carried dependence chains that limit parallelization. Other value speculations predict the value of loads [9] or that a store will be silent [40].

In general purpose programs, parallelization must be able to break dependences that arise from code that manipulates arbitrarily complex heap-allocated data structures. Even with aggressive memory analysis [14, 31, 53, 70] many loads and stores will potentially accesses the same data. At runtime, many such conflicts will not manifest themselves, either because they are not possible or are extraordinarily rare. *Memory Alias Speculation* leverages this property to remove memory data dependences between loads and stores.

Memory Alias Speculation removes memory dependences by assuming that two memory operations will not accesses the same memory location. Unlike value speculations, alias speculation only causes misspeculation when the destination of a dependence executes before the source. Alias speculation only ensures that a dependence does not have to be synchronized, but the dependence may be respected due to the timing of the threads involved in the dependence.

Finally, Memory Value Speculation combines the benefits of memory alias speculation with value speculation. Instead of checking that a load and store access the same memory location in the correct order, it checks that the value loaded is correct, usually by performing a second checking load that respects all memory dependences of the original load. Because the speculation checks values and not locations, it is only effective for the speculation of true memory dependences. However, unlike alias speculation, if a load happens to load the correct value, even from a different store, the speculation will not fail.

2.3.2 Speculative Parallelization Techniques

All speculative loop parallelization techniques use speculation to break dependences. To ensure that correct execution is always possible when speculating inter-thread dependences, speculative threads conceptually maintain speculative state separately from non-speculative state. This allows them to rollback to a known good state upon misspeculation.

Speculative IMT

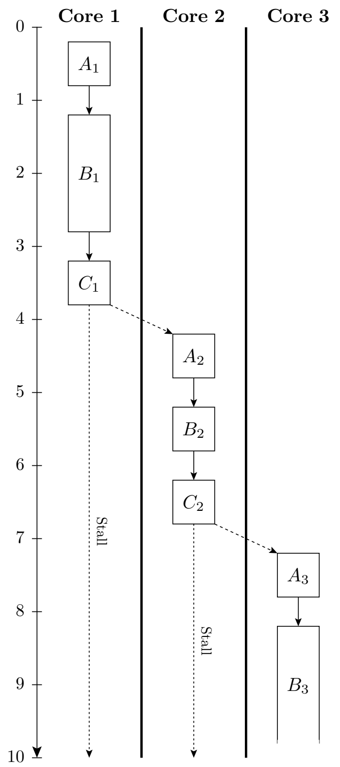
Adding speculation to the IMT parallelization techniques allows them to parallelize loops that would otherwise contain inter-iteration, cross-thread dependences. By speculating *all* inter-iteration dependences, Speculative DOALL techniques [44, 64, 91] can parallelize any loop. For most loops this involves breaking many dependences that are hard if not im-

```
   list *cur=head
A: for(; cur!=NULL; cur=cur->next) {
B:   cur->data = work(cur);
C:   if (cur->data == NULL)
       break;
   }
```

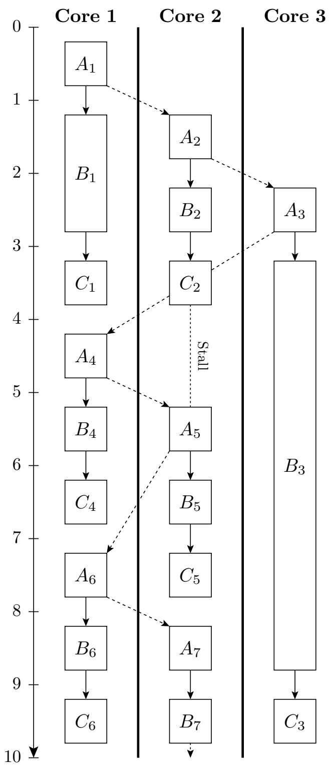
(a) List Code

```
   list *cur=head
A: for(; cur!=NULL; cur=cur->next) {
B:   cur->data = work(cur);
C:   if (cur->data == NULL)
       MISSPECULATE;
   }
```

(b) Speculated List Code



(c) Execution of Code in Figure 2.9a



(d) Execution of Code in Figure 2.9b

Figure 2.9: TLS Example

possible to predict and which manifest often. This leads to excessive misspeculation which in turn causes performance little better than their non-speculative counterparts. Where they are profitable, more accurate analyses or other transformations can often mitigate the need for speculation [44, 56].

Speculative CMT

Speculation is more useful in the domain of CMT techniques, which can use speculation to break problematic dependences in a more controlled manner than Speculative IMT techniques. There are many speculative CMT techniques that extend the basic DOACROSS concept with speculation and shall be referred to as Thread-Level Speculation (TLS) [22, 33, 71, 72, 80, 83] techniques.

As in DOACROSS, TLS techniques execute multiple iterations of a loop in parallel by executing each iteration in a separate thread. Unlike DOACROSS, only the oldest thread in sequential program order is non-speculative, all other threads execute speculatively. Threads commit in the original sequential program order to maintain correctness. Once a thread commits, the oldest speculative thread becomes non-speculative and the process repeats.

TLS techniques generally checkpoint memory at the beginning of each iteration, storing speculative writes into a separate version of memory space that is made visible to the rest of the system when the iteration becomes non-speculative. These iteration specific versions of memory closely resemble transactional memories in their semantics, but with a constrained commit and rollback operations [28, 39, 65]. That is, the oldest iteration's memory version must commit before younger versions. When a conflict occurs that requires a rollback, the younger iteration is always rolled back. To ensure that younger iterations do not commit their state until all older iterations have completed, the younger iteration waits at the end of its iteration to receive a *commit token* from the next oldest iteration.

Speculation in TLS techniques focuses on removing inter-iteration dependences. In practice, this means that memory alias speculation is used to remove potential dependences among load and stores. Control speculation is used to ensure that potential inter-iteration register and control dependences either become unreachable or can be scheduled early in the loop [90]. Detection of misspeculation can be performed in either hardware or software. Misspeculation of control speculation is detected by the program. For alias speculation, misspeculation is detected by the same hardware that maintains speculative memory versions. The separate memory versions also implicitly break inter-iteration false (anti- and output-) memory dependences, giving more opportunity for parallelism.

Figure 2.9 shows how speculating a loop exit branch can unlock parallelism. The speculation removes the dependence cycle from the loop exit to the loop header that prevents the extraction of any parallelism. When speculated, the branch can be executed in parallel, with misspeculation used to indicate that the loop is finished.

TLS techniques have acknowledged that synchronization is the limiting factor in a CMT parallelization, particularly when it occurs late in the iteration [90]. To avoid this limitation, a TLS parallelization splits a loop iteration into two parts, delineated by a single synchronization point. All inter-iteration dependences above the synchronization point are respected and communicated to the next iteration at the synchronization point. All inter-iteration dependences below the synchronization point are speculated away. In practice, keeping the unsynchronized region of the iteration large enough to all for effective parallelization while avoiding excessive misspeculation is difficult, requiring complicated misspeculation cost models [22] and runtime feedback to prune parallelizations that perform poorly [28].

Speculative PMT

Since PMT techniques are not bound by inter-thread communication, speculation in a PMT technique is generally aimed not at reducing inter-thread communication, but in removing dependences that form large pipeline stages. The speculative PMT analogue of DSWP is the Speculative DSWP (SpecDSWP) technique [82].

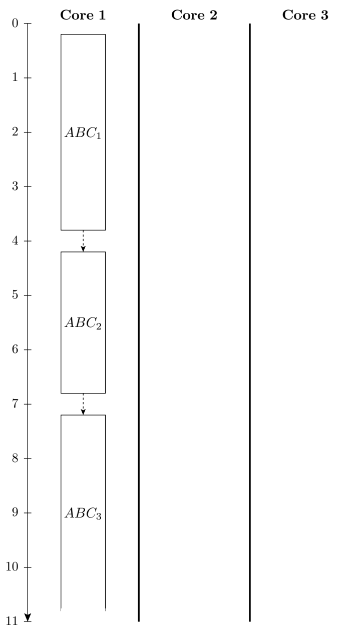
SpecDSWP uses speculation to break dependences that form large sets of intersecting dependence recurrences, leading to many smaller pipeline stages from which better parallelism can be extracted. Just as in DSWP, SpecDSWP executes stages to achieve pipelined parallelism, using queues to buffer state and tolerate variability. Since SpecDSWP allows intra-iteration speculation, no iteration executes non-speculatively. Instead each iteration is speculative and is committed when all stages have executed without misspeculation. A *Commit Thread* ensures that iterations commit in the proper order and manages the recov-

```
   list *cur=head
A: for(; cur!=NULL; cur=cur->next) {
B:   cur->data = work(cur);
C:   if (cur->data == NULL)
       break;
   }
```

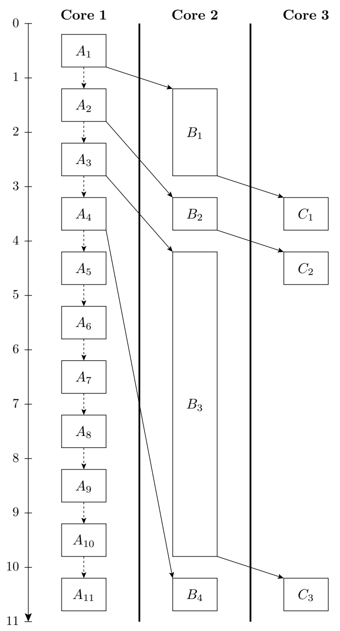
(a) List Code

```
   list *cur=head
A: for(; cur!=NULL; cur=cur->next) {
B:   cur->data = work(cur);
C:   if (cur->data == NULL)
       MISSPECULATE;
   }
```

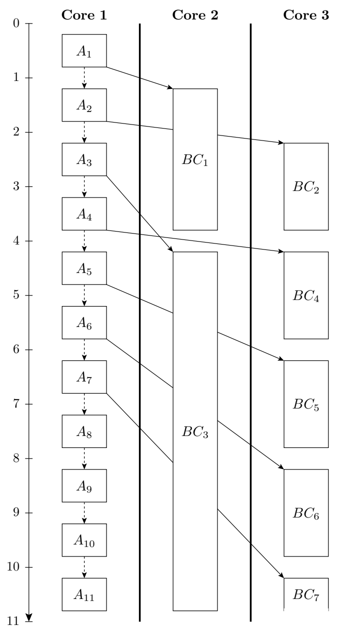
(b) Speculated List Code



(c) Execution of Code in Figure 2.10a with DSWP



(d) Execution of Code in Figure 2.10b with SpecDSWP



Core 1
Core 2
Core 3
A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10} A_{11}
BC_1 BC_3
BC_2 BC_4 BC_5 BC_6 BC_7
0 1 2 3 4 5 6 7 8 9 10 11

(e) Execution of Code in Figure 2.10b with PS-DSWP and SpecDSWP combined

Figure 2.10: SpecDSWP Example

ery process upon misspeculation detection.

As in TLS techniques, iterations in a SpecDSWP parallelization checkpoint state at the beginning of each iteration. Unlike TLS techniques, an iteration does not stay local to a single thread or processor. Instead, as it flows through the stages of the pipeline, it executes on multiple threads, potentially at the same time, on multiple processors. Existing TLS memory systems assume that the iteration executes in a single thread and cannot handle multiple threads participating in a single memory version at the same time. To handle this, SpecDSWP uses a version memory system that allows multiple threads to execute inside the same version at the same time [82].

SpecDSWP implements speculation entirely in software, relying on the hardware only for efficient execution of loads and stores in the versioned memory system. Control and silent store speculation are used to break dependences that form large sets of dependence recurrences, with misspeculation detection done in software. Though the initial implementation of SpecDSWP did not use speculation to break other memory dependences, memory value speculation can also be added to the suite of possible speculation types as it easily checked in software.

Figure 2.10 shows the result of applying SpecDSWP to the same loop as TLS was applied to in Figure 2.9, without showing the commit thread. Just as in TLS, the loop exit branch creates a large dependence recurrence that prevents SpecDSWP from extracting parallelism. After the branch is speculated, SpecDSWP can extract a 3-stage DSWP partition (Figure 2.10d). The power of SpecDSWP is greatly increased when it is combined with PS-DSWP. Figure 2.10e shows a 2-stage speculative PS-DSWP partition with the second stage replicated twice.

2.3.3 Loop-Sensitive Profiling

Many parallelization techniques use heuristics to guide them to profitable areas of the optimization space. Speculation in particular is highly dependent upon accurate profile information to determine when a control edge is rarely taken, a memory alias rarely manifests, or an operation usually computes the same value regardless of input. Many profiling techniques and infrastructures have been built to extract this information from programs. However, the profiles extracted are often loop-insensitive. This is problematic for loop parallelization techniques that want to speculate dependences relative to a specific loop header. In particular, identifying intra-iteration versus inter-iteration dependences can allow the compiler to more accurately determine how often a dependence will misspeculate [22].

2.4 Compilation Scope

Even an aggressive parallelization technique is of little use if it cannot parallelize loops that contain large portions of a program's execution time. In general-purpose programs, these loops often exist at or close to the program's outermost loop [9]. Consequently, it is important that the parallelization framework be powerful enough to identify and leverage parallelism that contains function calls and where dependences cross between many levels of the call graph.

In Figure 2.11, interprocedural analysis and optimization are needed to effectively parallelize the loop in `compute`. If the various parallelization techniques are applied only to the `compute` function (Figure 2.11b), the compiler cannot deal the `cache_insert` to `cache_lookup` loop-carried dependence inside `work`. Therefor, the call to `work` has inter-iteration dependences with itself that greatly limit the parallelism extractable by either CMT or PMT parallelization techniques.

To achieve the larger scope necessary to parallelize loops like those in Figure 2.11, most techniques remain intra-procedural. They rely on a pass of compiler inlining to expand a procedure to the point where optimizations can be successful, but inlining cannot hope to expose all the operations needed to parallelize effectively at the outermost loop levels. First, for a program with recursion, it is impossible to fully inline such that all operations in the

```
int work(list_t *data) {
  int result = cache_lookup(data);
  if (result != 0)
    return result;

  if (data == NULL)
    return -1;

  result = real_work(data);
  cache_insert(data, result);
}




int compute(list_t *list) {
  int total = 0;
  while (list != NULL) {
    list_t *data = getData(list);

    int result = work(data);
    if (result == -1)
      break;

    printData(data);
    total += result;
    list = list->next;
  }

  return total;
}
```

(a) Code Example

```
int work(list_t *data) {
  int result = cache_lookup(data);
  if (result != 0)
    return result;

  if (data == NULL)
    return -1;

  result = real_work(data);
  cache_insert(data, result);
}

int compute(list_t *list) {
  int total = 0;
  int __total[THREADS];

  while (1) {
    list_t *data = getData(list);
    list = list->next;
    printData(data);

    int result = work(data);
    if (result == -1)
      MISSPECULATE;

    __total[TID] += result;
  }

  total += sum(__total, THREADS);
  return total;
}
```

(b) Code Example with Parallelization Optimizations applied

Figure 2.11: Interprocedural Scope Example

loop are visible. Second, even when full inlining is possible, it leads to exponential code growth and significant compile-time concerns [34]. Because of this, inlining can rarely be applied aggressively enough to allow a parallelization technique to avoid synchronizing large portions of a program's runtime.

Whole program optimization [79] techniques can increase the compiler's visibility without requiring the modification of analysis or optimizations. Whole program optimization removes procedure boundaries, merging all code into a single control flow graph (CFG), so the compiler can both see and modify code regardless of its location in the program.

Through region formation, the compiler controls the amount of code to analyze and optimize at any given time, with code from multiple procedures potentially in the same region. This can achieve the effect of partial inlining [74, 87], without the associated code growth. This flexability comes at a cost, as regions tend to be small, on the order of 500 operations, in order to prevent a compile-time explosion. Additionally, it is not clear how to structure optimizations that cross region boundaries, as any parallelization technique is bound to do for any reasonable region size.

Previous work in parallelizing scientific programs has shown interprocedural analysis to be useful [11, 32, 42] when parallelizing array-based programs. These analyses aim to disambiguate the array accesses that occur in lower-level procedures. The results of this analysis are used only to enhance the parallelism extracted in the current procedure, not to facilitate an interprocedural parallelization. This has proved sufficient for array-based programs because they extract non-speculative DOALL parallelizations, so beyond proving a lack of dependence, there is no work to perform in lower-level procedures.

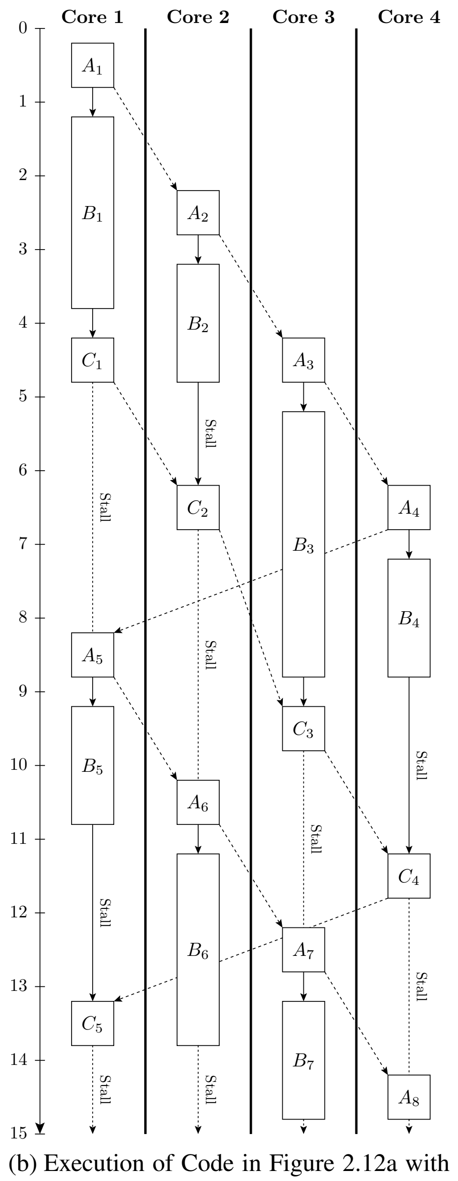
General-purpose programs, on the other hand, have dependences both in the loop being parallelized and buried deeply in lower-level functions. As such, it is important to have a interprocedural parallelization technique that is able to operate across procedures, synchronizing or speculating dependences across multiple levels of the call tree.

2.5 Parallelization For General-Purpose Programs

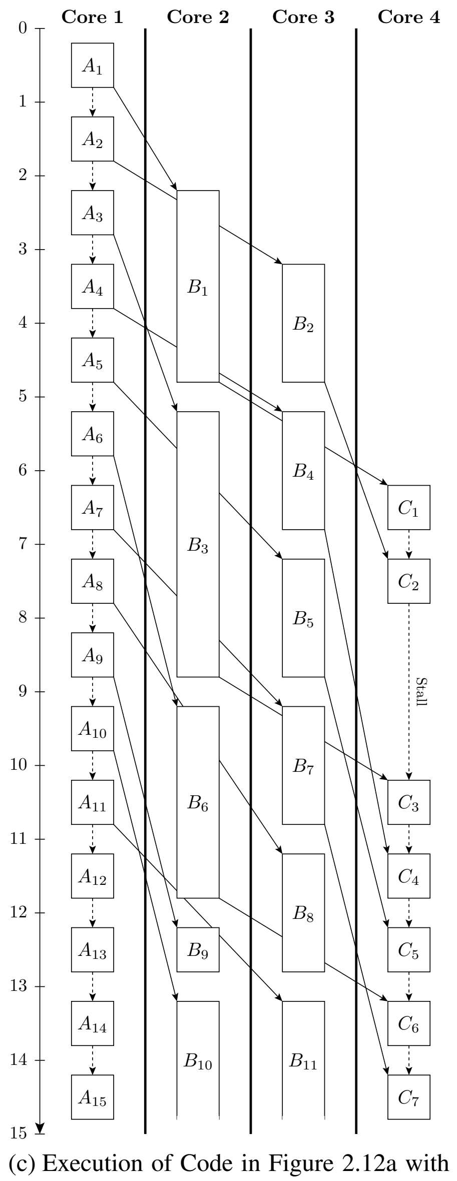
This dissertation focuses on building a parallelization framework that can extract scalable parallelism from the outermost loops of applications. Previous work has shown that scalable parallelism exists and generally has the abstract structure shown in Figure 2.12a [9, 10]. In this code sequence the `process` function takes the majority ($> 90\%$) of the program's runtime. The `get_data` function contains loop-carried dependences to itself, as does the `emit` function. Finally, the `emit` function is dependent upon the result of

```
   void *data, *result;
A: while (data = get_data()) {
B:   result = process(data);
C:   emit(data, result);
   }
```

(a) Code Example



(b) Execution of Code in Figure 2.12a with TLS



(c) Execution of Code in Figure 2.12a with PS-DSWP and SpecDSWP

Figure 2.12: Choice of Parallelism Example

the `process` function. Though the `process` function may contain loop-carried dependences, these can be broken though speculation, reduction expansion, etc.

Either a CMT (via TLS) or PMT (via SpecDSWP and PS-DSWP) technique with interprocedural scope could extract parallelism from this type of loop. In general, the SpecDSWP parallelization will offer better performance for a variety of reasons. First, as mentioned earlier, communication latency is still added to the critical path for the synchronized portion of a TLS parallelization. Additionally, TLS parallelizations generally do not allow for an explicit second synchronized region to avoid variability in one iteration stalling subsequent iterations. However, the *commit token* used by TLS implementations to ensure speculative state is committed correctly implicitly creates a second synchronized region at the end of every iteration.

SpecDSWP can better handle synchronizing dependences that occur anywhere in the iteration. The overhead of the commit thread is relevant for small numbers of threads, but as processors move to 8 or more cores, the overhead becomes smaller. Additionally, the commit thread be can context switched out to further reduce the overhead. By integrating the PS-DSWP and SpecDSWP extensions together, DSWP also gains the ability to extract dynamic parallelism, a primary benefit of a TLS parallelization. Finally, by not putting dependence recurrences across cores, DSWP is able tolerate long inter-core communication [55].

To handle the scope issues discussed in Section 2.4, the SpecDSWP technique also needs to be able to operate interprocedurally. This can be accomplished through a combination of interprocedural analysis and extensions to the way SpecDSWP represents and transforms the code, for example, by building a program-wide System Dependence Graph rather than a procedure-local Program Dependence Graph to represent depenedences.

Chapter 3

The VELOCITY Parallelization Framework

This chapter discusses the components of VELOCITY's aggressive automatic parallelization framework. This framework includes a compiler infrastructure to identify parallelism in sequential codes, hardware support to efficiently execute the parallelized code, interprocedural analysis, interprocedural optimization, and a loop-sensitive profiling infrastructure. All of these components will be discussed in the context of the Interprocedural Speculative Parallel-Stage DSWP (iSpecPS-DSWP) parallelization technique that VELOCITY uses. iSpecPS-DSWP will be formed by combining and extending the existing parallelization techniques Decoupled Software Pipelining (DSWP), Parallel-Stage DSWP (PS-DSWP), and Speculative DSWP (SpecDSWP).

3.1 Compilation and Execution Model

DSWP assumes a set of processors that each executes a single thread at a time. Each processor contains registers used by executing instructions as well as a shared memory. In the examples provided, a simple load/store architecture is assumed.

A program that has been DSWPed executes sequentially until in reaches the DSWPed loop. Upon reaching the loop it conceptually spawns the necessary threads, executes in parallel, and upon loop exit, the spawned threads conceptually terminate. In practice, the threads can be prespawned and reused by successive invocations to mitigate the overhead of spawning [55, 61].

For the sake of simplicity it is assumed that for the loop being DSWPed, all loop backedges have been coalesced into a single backedge and preheaders and postexits exist. Additionally, all irreducible inner loops have been cloned to make reducible loops [50]. Finally, all indirect branches and all indirect calls have been expanded to direct, 2-way branches and direct calls.

3.2 Decoupled Software Pipelining

Algorithm 1 DSWP

```
Require: Loop $L$, Threads $T$
1: Opportunity $O$ = DetermineAssignment($L$, $T$)
2: MultiThreadedCodeGeneration($L$, $O$)
```

The Decoupled Software Pipelining (DSWP) algorithm consists of a thread assignment analysis phase (line 1 of Algorithm 1) and a multi-threaded code generation transformation that realizes a thread assignment (line 2 of Algorithm 1).

3.2.1 Determining a Thread Assignment

Algorithm 2 DetermineAssignment

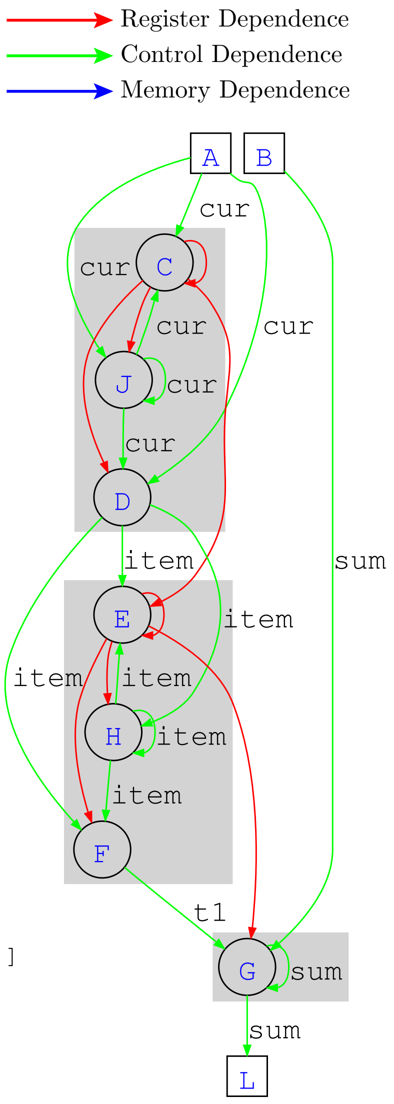
```
Require: Loop $L$, Threads $T$
1: $PDG$ = BuildPDG($L$)
2: $DAG_{SCC}$ = FormSCCs($PDG$)
3: ThreadAssignment $A$ = AssignSCCs($DAG_{SCC}$, $T$)
4: Synchronization $S$ = DetermineSynchronization($A$, $L$)
5: return ($A$, $S$)
```

```
list *cur = head;
while (cur != NULL) {
  list *item = cur->item;
  while (item != NULL) {
    sum += item->data
    item = item->next;
  }
  cur = cur->next;
}
print(sum)
```

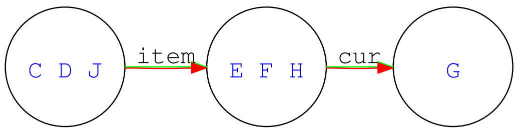
(a) Sequential C Code

```
START:
A:   mov cur=head;
B:   mov sum=0

BB1:
C:   br.eq cur, NULL, END

BB2:
D:   load item=[cur + offset(item)]

BB3:
E:   br.eq item, NULL, BB5

BB4:
F:   load t1=[item + offset(data)]
G:   add sum=sum,t1
H:   load item=[item + offset(next)]
I:   jump BB3

BB5:
J:   load cur=[cur + offset(next)]
K:   goto BB1

END:
L:   call print, sum
```

(b) Sequential Lowered Code



(c) PDG



(d) DAG_{SCC}

Figure 3.1: DSWP Example Code

Step 1: Build PDG

Since the goal of DSWP is to extract PMT parallelism, the thread assignment algorithm must extract threads that have no backwards dependences. To determine which dependences are forward or backward, DSWP first builds a Program Dependence Graph (PDG) [25], which contains all data and control dependences in the loop L. DSWP currently operates on a low-level IR representation with virtualized registers. Data dependences among these operations can come through registers or memory. Data dependences have three types:

Flow Dependence A dependence from a write to a read of either a register or a memory location. This is the only true dependence; it cannot be broken by renaming.

Anti Dependence A dependence from a read to a write of either a register or a memory location. This is a false dependence that can be broken making the write use a different name than the read.

Output Dependence A dependence from a write to a write of either a register or a memory location. This is a false dependence that can be broken by making one of the writes use a different name.

The PDG used for thread partitioning contains all memory flow, anti, and output dependences, control dependences, and register flow dependences. Both control and register dependences can be computed using standard data-flow analysis algorithms [50]. Only register flow dependences are inserted into the PDG. For any two false register dependent operations, if they are in different threads, then they will use different register files and the dependence is not violated. If they are in the same thread, the code generation algorithm will maintain the original ordering, respecting the dependence.

Memory dependences are generally harder to determine than register dependences. Unlike register dependences, memory operations can potentially reference many locations, making it hard for an analysis to obtain accurate results. For array-based programs, memory analysis focuses on disambiguating bounds of array references. For general-purpose programs that manipulate recursive data structures, more complicated analysis can disambiguate most non-conflicting memory accesses, even in type unsafe languages like C, though this remains an area of active research [14, 31, 53].

Figure 3.1c shows the PDG formed for the sequential code in Figure 3.1b. Green arrows represent register data dependences and are annotated with the register that the dependence is on. Red arrows represent control dependences. Variables live into and out of the loop being parallelized are also shown, though they will not take part in further steps unless explicitly noted.

Step 2: Form SCCs

To keep dependence recurrences thread-local, DSWP relies on the formation of the DAG_{SCC} from the PDG. The DAG_{SCC} contains the strongly connected components (SCC) [76] of the PDG (i.e. the dependence recurrences) and has no cycles in it. The shaded boxes in Figure 3.1c illustrate the three strongly connected components, while Figure 3.1d illustrates the DAG_{SCC} itself, with nodes collapsed and represented by their containing SCC.

Step 3: Assign SCCs

Given sufficient threads, each SCC in the DAG_{SCC} can be assigned its own thread and the step is finished. However, the common case is to have more SCCs than threads. A thread partitioner is used to find the allocation of SCCs to threads that maximizes parallelism by creating equal weight threads, *without creating backwards dependences*.

Unfortunately, this problem is NP-Hard even for simple processors with scalar cores [55]. Heuristic partitioners have been proposed and used in practice to solve this problem [55, 61]. These partitioners attempt to merge SCCs until there are as many SCCs as threads. Two nodes N_1 and N_2 in the DAG_{SCC} can be merged so long as there is no node $N_3 \neq N_1 \neq N_2$ such that $N_1 \xrightarrow{*} N_3$ and $N_3 \xrightarrow{*} N_2$ or that $N_2 \xrightarrow{*} N_3$ and $N_3 \xrightarrow{*} N_1$. This avoids creating cycles in the DAG_{SCC} which would create CMT instead of PMT paral-

```
   ...

L1:
A: (1) br r1<MAX1, L5

L2:
B: (1) br r2<MAX2, L5

L3:
C: (1) br r3<MAX3, L5

L4:
D: (2) add r4=r4,1
E: (1) sub r5=r5,r2

L5:
   ...
```

(a) Sequential Code with Thread Assignment in parenthesis

```
   ...

L1:
   synchronize r1, T2
A: br r1<MAX1, L5

L2:
   synchronize r2, T2
B: br r2<MAX2, L5

L3:
   synchronize r3, T2
C: br r3<MAX3, L5

L4:
E: sub r5=r5,r2

L5:
   ...
```

(b) Thread 1

```
   ...

L1:
    synchronize r1, T1
A': br r1<MAX1, L5

L2:
    synchronize r2, T1
B': br r2<MAX2, L5

L3:
    synchronize r3, T1
C': br r3<MAX3, L5

L4:
D : add r4=r4,1

L5:
   ...
```

(c) Thread 2

Figure 3.2: Communication Synchronization Example

lelism.

DSWP uses a *Load-Balance* partitioner that attempts to create equal weight threads by assigning operations to threads while minimizing the overhead of synchronization [54, 55]. In particular, because loads are often on the critical path, they are assigned a greater weight than other operations.

Step 4: Determine Synchronization

For a given thread partitioning there are a set of dependences that must be synchronized to ensure correct execution. Synchronization is implemented through cross-thread communication, as the terms will be used interchangeably. Synchronization can be broken down into two steps, determining what to synchronize and deciding where to synchronize it.

For a given dependence $S \xrightarrow{dep} D$ with S in thread T_S and D in thread T_D, if $T_S \neq T_D$, then the dependence must be communicated from T_S to T_D. For data dependences, it is sufficient to communicate the data value itself. For control dependences, DSWP assumes the branch will be replicated in T_D , meaning that only the branch's operands must be syn-

```
(1) move r1 = 0          (1) move r1 = 0          (1) move r1 = 0

L1:                      L1:                      L1:
(1) add r1 = r1, r3      (1) add r1 = r1, r3      (1) add r1 = r1, r3
(1) br r1 < MAX1, L1         synchronize r1, T2   (1) br r1 < MAX1, L1
                         (1) br r1 < MAX1, L1
                                                      synchronize r1, T1
(1) r2 = 0               (1) r2 = 0               (1) r2 = 0

L2:                      L2:                      L2:
(2) add r2 = r2, r1      (2) add r2 = r2, r1      (2) add r2 = r2, r1
(2) br r2 < MAX2, L2     (2) br r2 < MAX2, L2     (2) br r2 < MAX2, L2
```

(a) Example Code (b) Naïve Synchronization (c) Optimized Synchronization

Figure 3.3: Communication Optimization Example

crhonized as cross-thread data dependences. Alternatively, the source thread could communicate the direction of the branch along each path, which also introduces a cross-thread data dependence on the direction's data value.

However, this alone will not ensure proper control flow in T_D because indirect control dependences will not have been communicated. Figure 3.2 shows a simple 2 thread partitioning where all operations except the `add` assigned to T_2 have been assigned to T_1. If indirect control dependences are not synchronized, only the direct control dependence on branch C will have been communicated. Branches A and B will not exist and D will execute even when A and B jump to $L5$. The solution to this problem is to consider all indirect control dependences to be cross-thread control dependences. Since the branch's operands can be synchronized in the same condition of execution as the branch itself, their control dependences, both direct and indirect, do not need to be synchronized.

Once the set of dependences that must be synchronized is constructed, the algorithm must determine where to place the synchronization. The naïve approach to determining the position at which to synchronize dependences does so at the source of the dependence. Since T_S has all branches needed to compute the condition of execution for S, in the worst case it can communicate all branches related to the condition of execution for S to T_D. Thus, no backwards dependences can be created by this placement of synchronization [55].

While this placement is sufficient to ensure correctness, it can easily lead to excessive communication, particularly in the case of memory dependences [54]. A better synchronization placement algorithm can analyze the many paths between the source and destination to determine a more optimal placement. The communication optimization framework proposed by Ottoni et al [54] optimizes the placement of each type of dependence for each pair of threads, while ensuring that no new backwards dependences are created.

Register Dependence For each register r, all cross-thread register dependences are optimized as a single unit. An effective solution can be found by modeling the problem as max-flow/min-cut [17], where the nodes are control flow nodes and the edges are control flow edges. The graph is the control flow subgraph that includes all nodes all along paths from all definitions to all uses. If an edge is a legal placement for synchronization, then its weight is set to the profile weight of the control edge, otherwise it is set to ∞. In particular, edges that would result in backwards communication to facilitate new conditions of execution are illegal. The min-cut of the graph produces the points at which to synchronize r.

Memory Dependence Memory dependences are optimized similar to register dependences. However, since memory dependences only communicate a token whose value does not matter, memory dependences among unrelated groups of instructions can be optimized at the same time. Because disjoint sets of instruction can be optimized at the same time, the max-flow/min-cut problem for memory dependences has multiple sources and multiple sinks. While the multi-source, multi-sink max-flow/min-cut problem is NP-hard [27] to separate each source from its corresponding sink, heuristics exist that can achieve effective solutions [54].

Control Dependence Control dependences are not optimized by the framework. However, the arguments of the branch that is the source of the control dependence can be optimized as cross-thread register or memory dependences.

Figure 3.3 shows how optimizing synchronization placement can improve performance. The register `r1` must be communicated from T_1 to T_2, however, there are two places at which to communicate the variable. The naïve placement inserts the synchronization at the source of the dependence, inside the $L1$ loop, forcing the second thread to receive the value every iteration of the $L1$ loop. A more optimal placement for the synchronization of $r1$ occurs just before the $L2$ loop so that the value only needs to be synchronized once.

3.2.2 Realizing a Thread Assignment

Algorithm 3 MultiThreadedCodeGeneration

```
Require: Loop $L$, (ThreadAssignment $A$, Synchronization $S$)
1: CreateThreadBlocks($L$, $A$, $S$)
2: MoveInstructions($A$)
3: InsertSynchronization($S$)
4: CreateControlFlow($L$, $A$)
```

DSWP leverages the Multi-Threaded Code Generation (MTCG) algorithm proposed by Ottoni et al to generate correct multi-threaded code [55]. The code example from Figure 3.1 will be used to provide examples through this subsection, assuming a thread assignment of $\{CDJ, EFHG\}$ and an unoptimized synchronization placement.

Step 1: Create Thread Blocks

The first step of the MTCG algorithm is to create a new CFG CFG_i for each thread $T_i \in A$. The set of blocks to create in each new CFG is defined by the set of relevant basic blocks from the original code. For each basic block, B, relevant to T_i, a new basic block B' is created in CFG_i. A *Relevant Basic Block* for T_i is any block in the original CFG that contains either an instruction assigned to T_i or the synchronization point for cross-thread dependence that begins or ends in T_i. This ensures that any operation or synchronization in T_i has a place to reside. In addition to the relevant basic blocks, special $START$ and END blocks are inserted at the loop's preheader and postexits respectively.

Step 2: Moving Instructions

```
START:
A:  mov cur=head;
B:  mov sum=0

BB1:
C:  br.eq cur, NULL, END

BB2:
D:  load item=[cur + offset(item)]

BB5:
J:  load cur=[cur + offset(next)]
K:  goto BB1

END:
L:  call print, sum
```

(a) Thread 1

```
START':

BB1':
C': br.eq cur, NULL, END

BB2':

BB3':
E:  br.eq item, NULL, BB5

BB4':
F:  load t1=[item + offset(data)]
G:  add sum=sum,t1
H:  load item=[item + offset(next)]
I:  jump BB3

END':
```

(b) Thread 2

Figure 3.4: Multi-Threaded Code after moving operations

Once the basic blocks for each thread have been created, the instructions assigned to that thread can be moved into the thread. For a specific basic block BB from the original thread, all operations in BB assigned to thread T_i are inserted into the newly created basic block BB' in T_i that represents BB. The order among the operations is maintained, in order to preserve the intra-thread dependences that may exist. Figure 3.4 shows the results of moving operations to the basic blocks created for them.

Step 3: Insert Synchronization

Queue Instructions		
Instruction	Arguments	Description
`produce`	[Q] = V	Pushes value V onto the tail of queue Q if it is not full. If the queue is full, the instruction stalls until room is available.
`consume`	R = [Q]	Pops a value off of the head of queue Q if there are values in the queue and places the value in register R. If the queue is empty, the instruction stalls until one is available.

Table 3.1: DSWP ISA Extensions

The MTCG algorithm then inserts synchronization between threads to ensure correct

```
START:
A:  mov cur=head;
B:  mov sum=0
    produce [1]=sum

BB1:
    produce [2]=cur
C:  br.eq cur, NULL, END

BB2:
D:  load item=[cur + offset(item)]
    produce [3]=item

BB5:
J:  load cur=[cur + offset(next)]
K:  goto BB1

END:
    consume sum=[4]
L:  call print, sum
```

(a) Thread 1

```
START':
    consume sum=[1]
BB1':
    consume cur=[2]
C': br.eq cur, NULL, END

BB2':
    consume item=[3]

BB3':
E:  br.eq item, NULL, BB5

BB4':
F:  load t1=[item + offset(data)]
G:  add sum=sum,t1
H:  load item=[item + offset(next)]
I:  jump BB3

END':
    produce [4]=sum
```

(b) Thread 2

Figure 3.5: Multi-Threaded Code after communication insertion

execution. MTCG assumes the existence of a set of queues that can be accessed via *produce* and *consume* instructions. Table 3.1 gives the semantics of these instructions. The queues used by these instructions are orthogonal to the memory and register systems. Synchronization is achieved by *producing* a value in the source thread at the synchronization point and *consuming* it in the destination thread, also a the synchronization point. Each synchronization uses a separate queue, though later queue allocation phases can reduce the number needed. There are three types of dependences that must be synchronized:

Register Dependence The register is pushed onto the queue by the producer thread and popped off the queue by the consumer thread.

Control Dependence The registers that make up the condition of the branch are communicated as register dependences, while the branch is replicated on the consumer side.

Memory Dependence Since MTCG assumes a shared-memory model, the dependence must be synchronized via write and read barriers that enforce a global memory ordering. A produce/consume pair is inserted where the produce is a global write barrier and the consume is a global read barrier. A token value is sent by the produce to indicate that the write barrier has occurred.

In addition to communicating values during execution, the live-ins and live-outs must be communicated when moving from sequential to parallel execution and vice-versa.

For registers live into the loop, before each thread enters its copy of the DSWPed loop, the main thread sends it the register live-ins that it needs to execute. On loop exit, the thread then sends back the live-outs of its loop to the main thread. If there are multiple threads that potentially define the same live-out register, extra synchronization is inserted to insure that one of the threads always holds the correct value upon loop exit [55].

Memory live-ins do not need to be communicated specially, as the thread spawn that conceptually starts the parallel region is assumed to be a write barrier in the spawning thread and a read barrier in the spawned thread. Memory live-outs do require that a memory synchronization occur in any thread that contains store instructions, ensureing that code after the loop sees the latest computed values.

Figure 3.5 shows the results of inserting communication for the thread assignment $\{CDJ, EFGH\}$ assuming naïve synchronization points. On entry to the loop, T_1 communicates the live-in `sum` to T_2. On every iteration of the outer loop, T_1 communicates `cur` to satisfy a cross-thread control dependence, and `item` to satisfy a cross-thread register dependence. Finally, on loop exit, T_2 sends the live-out variable `sum` back to T_1.

Create Control Flow

Finally, with all the appropriate code for each thread inserted, it remains only to insert control flow into each thread's CFG. This consists of retargeting both conditional branches assigned or copied into the thread and inserting unconditional control flow at the end of basic blocks. All control flow must properly compensate for the fact that not *all* basic blocks were copied into a thread. Thus, the mapping from target in the original code to

```
START:
A:  mov cur=head;
B:  mov sum=0
    produce [1]=sum

BB1:
    produce [2]=cur
C:  br.eq cur, NULL, END

BB2:
D:  load item=[cur + offset(item)]
    produce [3]=item

BB5:
J:  load cur=[cur + offset(next)]
K:  goto BB1

END:
    consume sum=[4]
L:  call print, sum
```

(a) Thread 1

```
START':
    consume sum=[1]

BB1':
    consume cur=[2]
C': br.eq cur, NULL, END'

BB2':
    consume item=[3]

BB3':
E:  br.eq item, NULL, BB1'

BB4':
F:  load t1=[item + offset(data)]
G:  add sum=sum,t1
H:  load item=[item + offset(next)]
I:  jump BB3'

END':
    produce [4]=sum
```

(b) Thread 2

Figure 3.6: Multi-Threaded Code after redirecting branches

target in each thread is not direct.

MTCG ensures that the control dependences among basic blocks in the new thread, T_i, mimic those of the original thread, T_0. As control dependence is defined by post-dominance, it is sufficient to redirect each branch's target to the closest post-dominator in T_0's post-dominance tree for which a block in T_i exists. This post-dominator is guaranteed to exist, as in the worst case it is the END block that exits the loop. Ottoni et al [55] have shown that this transformation ensures that control dependences among basic blocks in each thread match the control dependences among the corresponding blocks in the original thread.

Figure 3.6 illustrates the results of redirection. No redirection was needed in T_1. In T_2, the C' and I branches have targets that directly map to basic blocks in T_2. However, the E branch has target $BB5$ which does not exist in T_2, so it is redirected to its closest post dominator $BB1$, which has an analogous basic block in T_2.

3.3 Parallel-Stage DSWP

As discussed in Section 2.2.3, DSWP is limited in its ability to extract DOALL-like, data-level parallelism. The Parallel-Stage extension to DSWP removes this limitation by allowing pipeline stages with no inter-iteration dependences to be replicated an arbitrary number of times [61]. This section discusses the alterations to the DSWP algorithm necessary to achieve this end.

3.3.1 Determining a Thread Assignment

Parallel-Stage DSWP (PS-DSWP) changes a DSWP thread assignment to include not just a set of stages and the synchronization amongst those stages, but also a replication count for each stage.

Algorithm 4 Partition

```
Require: Loop $L$, Threads $T$
1: $PDG$ = BuildPDG($L$)
2: $DAG_{SCC}$ = FormSCCs($PDG$)
3: ThreadAssignment $A$, Replications $R$ = AssignSCCs($DAG_{SCC}$, $T$)
4: Synchronization $S$ = DetermineSynchronization($A$, $R$, $L$)
5: return ($A$, $R$, $S$)
```

Step 1: Build PDG

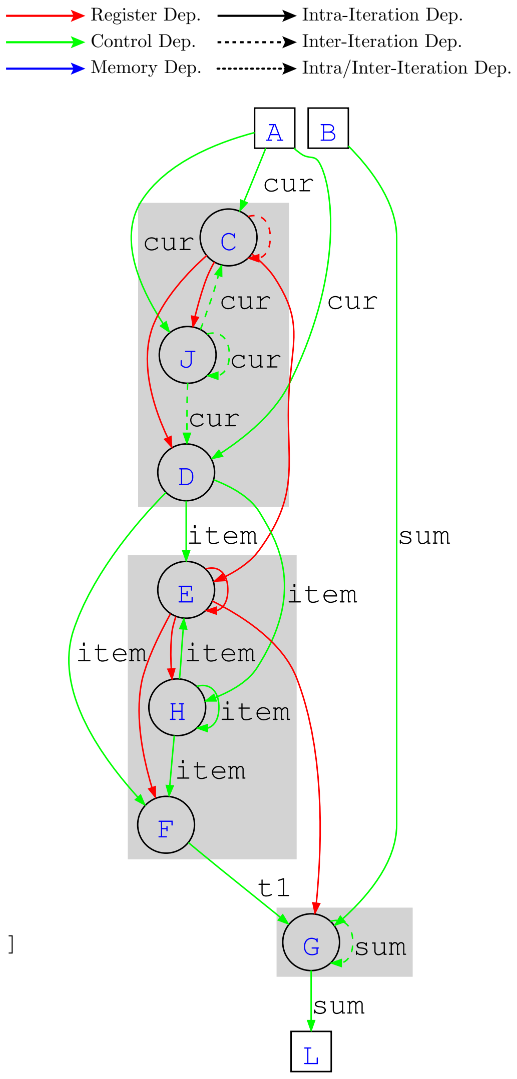
PS-DSWP changes the PDG built by the base DSWP algorithm to include information about the loop-carriedness of each dependence arc. Specifically, each arc is identified as only intra-iteration (INTRA), only inter-iteration (INTER), or either (EITHER). Note that EITHER indicates that a dependence may be both INTRA and INTER, not that it *must* be both. All dependences implicitly start as EITHER and analyzed to refine their type into either INTRA or INTER. Prior implementations of PS-DSWP distinguished only INTER from EITHER [61], but the utility of INTRA will become apparent once PS-DSWP is extended with speculation.

```
list *cur=head;
while (cur != NULL) {
  list *item=cur->item;
  while (item != NULL) {
    sum += item->data
    item = item->next;
  }
  cur = cur->next;
}
print(sum)
```

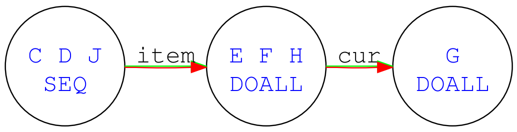
(a) Sequential C Code

```
START:
A:  mov cur=head;
B:  mov sum=0

BB1:
C:  br.eq cur, NULL, END

BB2:
D:  load item=[cur + offset(item)]

BB3:
E:  br.eq item, NULL, BB5

BB4:
F:  load t1=[item + offset(data)]
G:  add sum=sum,t1
H:  load item=[item + offset(next)]
I:  jump BB3

BB5:
J:  load cur=[cur + offset(next)]
K:  goto BB1

END:
L:  call print, sum
```

(b) Sequential Lowered Code



(c) PDG



(d) DAG_{SCC}

Figure 3.7: PS-DSWP Example Code

For each dependence, a graph reachability analysis determines whether the dependence must be INTER only. For a dependence $S \xrightarrow{dep} D$, if starting at S, D can only be reached using control flow by traversing the loop backedge, then the dependence must be INTER. Otherwise, more sophisticated analysis are used on a case-by-case basis as described below:

Register Dependence For a register dependence $S \xrightarrow{r} D$ that writes r at S and reads it at D, loop-carriedness is calculated by computing whether the dependence is carried by the loop backedge. Specifically, if the definition of r at S does not reach the loop header or there is not an upwards-exposed use of r at D at the loop header, then the dependence is INTRA. Note these conditions only prove that dependence is not loop-carried, and a failure to satisfy them does not prove the dependence is not also intra-iteration.

Control Dependence For control dependences, the graph reachability analysis used to prove INTER is also sufficient to prove INTRA. That is, if for $S \xrightarrow{cd} D$, D can only be reached using control flow from S by traversing the loop backedge, then the dependence must be INTER. Otherwise it is INTRA.

Memory Dependence Due to the issues involved in determining strong updates for heap and function argument pointers in C-like languages, it is generally not possible to determine if a store *MUST* write or a load *MUST* read a particular memory location. As such, it is all but impossible to determine the set of reaching memory definitions or upwards exposed memory uses. Thus, all memory dependences are considered EITHER unless proven INTER by the graph reachability analysis.

Reusing the example code from Figure 3.1, Figure 3.7c shows the PDG that would be built by PS-DSWP. INTRA edges are denoted by solid lines, while INTER edges a represented by dashed lines; no EITHER edges exist in this PDG, but are represented in later examples as dotted lines.

Step 2: Form SCCs

The DAG_{SCC} is formed from the PDG just as in the base DSWP algorithm. However, each SCC is marked as either ***SEQUENTIAL*** if a loop-carried edge exists among the nodes in the SCC or ***DOALL*** if there is no such edge. Additionally, an SCC that is a set of operations that create a min/max reduction or accumulator (addition or subtraction) is also marked as DOALL. The SCC G in Figure 3.7c is an accumulator, thus its loop-carried dependence does not cause it to be marked SEQUENTIAL. The code generation phase will create thread-local reductions and insert the appropriate cleanup code to calculate the global reduction. Though not discussed in this dissertation, PS-DSWP can also expand induction variables [61] when optimizing array-based loops.

Step 3: Assign SCCs

The thread assignment process is augmented to provide a replication count for each stage. Note that all DSWP thread assginments are implicitly PS-DSWP thread assignments with a replication count of 1 for each stage. Because a DOALL node replicated N times has its weight effectively reduced by a factor N, a new partitioner is needed that understands the effects of replication on performance.

The simplest such partitioner, *Single Parallel-Stage* partitioner, attempts to extract a single, large DOALL stage, potentially surrounded by a SEQUENTIAL stage on either side [61]. It does so by aggressively merging DOALL nodes together until no further legal merging possible. Higher weight DOALL nodes are merged together first. The largest DOALL node formed by this merging is retained, while the other DOALL nodes are marked SEQUENTIAL. The algorithm then aggressively merges SEQUENTIAL nodes until 2 are left, one before the DOALL stage and one after. The DOALL stage is assigned a replication factor of $T - 2$, while each of the SEQUENTIAL stages is assigned a replication factor of 1. The Single DOALL partitioner is simple, quick, and can often extract good partitions when there are large blocks of code that can execute in parallel.

However, when no DOALL stage exists, the single parallel-stage partitioner can produce worse partitionings than the partitioners used by DSWP. To avoid this problem, the *Iterative DOALL* partitioner attempts to balance the IMT parallelism of DOALL stages with the PMT parallelism of SEQUENTIAL stages. This partitioner aggressively merges DOALL SCCs just in the Single DOALL partitioner. However, after the largest DOALL SCC nodes are formed, the partitioner explores the space of possible thread assignments, potentially extracting a partitioning with several SEQUENTIAL stages, depending on whether more IMT or PMT parallelism is available.

Step 4: Determine Synchronization

This step is unchanged.

3.3.2 Realizing a Thread Assignment

Algorithm 5 MultiThreadedCodeGeneration

```
Require: Loop $L$, (ThreadAssignment $A$, Replications $R$, Synchronization $S$)
1: CreateThreadBlocks($L$, $A$, $S$)
2: MoveInstructions($A$)
3: InsertSynchronization($S$, $R$)
4: CreateControlFlow($L$, $A$, $R$)
```

The subsection describes the changes to the MTCG algorithm needed to handle a PS-DSWP thread assignment. The current PS-DSWP code generation algorithm assumes that all DOALL stages have the same replication factor R and that at least one SEQUENTIAL stage exists before and after each DOALL stage. Throughout this subsection, R_S denotes the replication factor of stage S, either 1 for SEQUENTIAL stages or R for DOALL stages, and r_T denotes the replication number of thread T. The code example from Figure 3.7 will be used to provide examples throughout this subsection, assuming a thread assignment of $\{(CDJ, 1), (EFHG, 2)\}$ and naïve synchronization.

Step 1: Create Thread Blocks

Basic block creation is essentially unchanged from the base DSWP algorithm. However, for a DOALL stage, the loop header is required to be relevant for each new CFG so that PS-DSWP can insert operations that must execute each iteration into it.

Step 2: Move Instructions

For SEQUENTIAL stages, operations are moved to the threads that they were assigned to, just as in DSWP. For DOALL stages, to avoid code bloat, each thread that is assigned to the stage will use the same code. To allow the code to distinguish which thread is executing and perform thread specific actions, each spawned thread takes as an argument a unique replication number in the range $[0, R_S)$ for a stage replicated R_S times. SEQUENTIAL stages have $R_S = 1$ and $r_T = 0$.

Any reduction expansion that occurs in a DOALL stage is expanded into thread-local copies at this point. For register variables, no change is needed to the iteration's code. For memory variables, an array of size R_S is created and each thread uses its replication number r_T to index into a separate element of the array. Buffers are placed between each element to avoid false sharing effects in the cache.

Step 3: Inserting Communication

Queue Instructions		
Instruction	Arguments	Description
`set.qbase`	V	Sets the queue base $Base$ for a thread to the given value V.
`produce`	[Q] = V	Pushes value V onto the tail of queue $Q + Base$ if it is not full. If the queue is full, the instruction stalls until room is available.
`consume`	R = [Q]	Pops a value off of the head of queue $Q + Base$ if there are values in the queue and places the value in register R. If the queue is empty, the instruction stalls until one is available.

Table 3.2: PS-DSWP ISA Extensions

For each iteration, dependences are synchronized among threads is the same way as in the base DSWP algorithm. However, to enable each thread in a DOALL stage to execute

```
START:
A:  mov cur=head;
B:  mov sum=0
    produce [1]=0
    produce [6]=0
    mov iter=r_T

BB1:
    mul qoff=iter,5
    add iter=iter,1
    mod iter=iter,2
    set.qbase qoff
    produce [3]=cur
C:  br.eq cur, NULL, END

BB2:
D:  load item=[cur + offset(item)]
    produce [4]=item

BB5:
J:  load cur=[cur + offset(next)]
K:  goto BB1

END:
    consume sum1=[5]
    consume sum2=[10]
    add sum=sum1,sum2
L:  call print, sum
```

(a) Thread 1

```
START':
    mul qoff=5,r_T
    set.qbase qoff
    consume sum=[1]

BB1':
    consume cur=[3]
C': br.eq cur, NULL, END

BB2':
    consume item=[4]

BB3':
E:  br.eq item, NULL, BB1'

BB4':
F:  load t1=[item + offset(data)]
G:  add sum=sum,t1
H:  load item=[item + offset(next)]
I:  jump BB3

END':
    produce [5]=sum
```

(b) Thread 2

Figure 3.8: Multi-Threaded Code after communication insertion

using the same code, the queues used to achieve this synchronization must be different per iteration. PS-DSWP conceptually creates a new sequence of queues per iteration to allow the threads that execute the iteration to synchronize. Each sequence of queues is referred to as a *queue set*. All synchronization instructions in each stage refer to queue numbers relative to the $Base$ of a queue set, allowing PS-DSWP to change queue sets without changing the synchronization instructions.

In an actual implementation, only a finite number of queue sets will exist, so they must be reused. However, the parallelization must ensure that at no time is the same queue set simultaneously being used for synchronization by two different iterations. Each pair of stages uses a non-overlapping portion of an iteration's queue set to ensure that each pair of stages never violate this principle. Thus, PS-DSWP must ensure that stages with two or more threads are not using the same queue set to synchronize two different iterations.

Since PS-DSWP allows only two replication factors to exist, 1 and R, a static allocation of iterations to threads can be created with R queue sets, assigning each iteration I to queue set $I \; mod \; M$. Each stage then executes the iteration on thread $((I \; mod \; M) \; mod \; R_S)$ [61]. If $R_S = M$, then queue set $(I \; mod \; M)$ is always executed on the same thread $((I \; mod \; M) \, mod \, M = (I \, mod \, M))$. If $R_S = 1$, every queue set is also executed on the same thread.

Figure 3.8 shows the implementation of a static allocation of iterations for the partitioning $((CDJ, 1), (EFHG, 2))$. Each thread computes the next iteration assigned to it in the `iter` variable and then uses that variable to compute the proper queue set. Note that in the static allocation scheme, DOALL stages do not need to update their queue set each iteration, as they will always execute uisng the same one.

A dynamic allocation can also be achieved where threads of a DOALL stage dynamically execute iterations instead of having them assigned *a priori*. In this situation, each thread in a DOALL stage must receive the iteration it will execute, and use it to calculate the value of $(I \; mod \; M)$. To ensure that the same queue set is not reused by two different threads in two different iterations, a queue set must be allocated for each potentially outstanding iteration in the pipeline. The total number of potentially outstanding iterations is bounded by the number of iterations that can be buffered between stages, calculated as $queue\_depth * R * |stages|$.

The implementation of a dynamic assignment of iterations requires a change to the DOALL stage as a method is needed for a DOALL stage to acquire the next iteration to work on. Since the synchronization mechanism in DSWP assumes a single producer and a single consumer, it is insufficient for multiple threads to consume from the same queue. For this, PS-DSWP uses a lock to ensure that only one thread at a time consumes from the queue that gives the iteration. All other synchronization in the stage is unchanged. To amortize the overhead of the lock and reduce contention, a thread can consume several iterations for each lock acquisition.

Beyond ensuring that iterations operate in the appropriate queue set, changes are needed to live-in and live-out synchronization. For SEQUENTIAL stages, this synchronization is unchanged from the base DSWP algorithm.

Live-ins to a DOALL stage defined both outside the loop and in a prior SEQUENTIAL stage must be communicated at the loop header of the SEQUENTIAL stage every iteration. Consider a definition of the register that dynamically occurred in iteration I. The DOALL thread executing I would have the appropriate synchronization and execute correctly. However, other threads executing later iterations would not receive the value and would execute incorrectly. By sending the value every iteration, all threads in the DOALL stage are assured of having the proper value.

Live-outs from a DOALL stage also require that additional information be communicated to the main thread. In particular, to allow the main thread to determine the last writer of the live-out, a last written iteration timestamp is also sent back to the main thread, which then compares the timestamps to determine the last thread to write the register. Additionally, min/max reductions communicate the thread-local min/max and iteration timestamp of the last update so that the main thread can determine the appropriate value. Finally, accumulator expanded variables similarly communicate their thread-local value, which the main thread accumulates into the proper value.

Step 4: Create Control Flow

Branches for PS-DSWP are redirected exactly as in the base DSWP algorithm. However, DOALL stages require that an extra loop exit be introduced to ensure proper termination. As only a single thread assigned to a DOALL stage is executing an iteration, if that iteration exits, the other threads assigned to the DOALL stage will not know to exit. The solution is to introduce a loop exit branch in the loop header before any other operations. The previous SEQUENTIAL stage produces the value *TRUE* to that queue each iteration to indicate that it can proceed to execute the iteration. On loop exit, the exiting thread for the DOALL stages sends the value *FALSE* to every other thread assigned to the DOALL stage, causing them to also exit and send their live-outs back to the main thread. Figure 3.9 illustrates the final code produced after branches have been redirected and the extra DOALL stage exit inserted.

```
START:
A:  mov cur=head;
B:  mov sum=0
    produce [1]=0
    produce [6]=0
    mov iter=rT

BB1:
    mul qoff=iter,5
    add iter=iter,1
    mod iter=iter,2
    set.qbase qoff
    produce [2]=TRUE

    produce [3]=cur
C:  br.eq cur, NULL, END

BB2:
D:  load item=[cur + offset(item)]
    produce [4]=item

BB5:
J:  load cur=[cur + offset(next)]
K:  goto BB1

END:
    consume sum1=[5]
    consume sum2=[10]
    add sum=sum1,sum2
L:  call print, sum
```

(a) Thread 1

```
START':
    mul qoff=5,rT
    set.qbase qoff
    consume sum=[1]

BB1':
    consume exec=[2]
    br.eq exec, FALSE, END2

    consume cur=[3]
C': br.eq cur, NULL, END'

BB2':
    consume item=[4]

BB3':
E:  br.eq item, NULL, BB1'

BB4':
F:  load t1=[item + offset(data)]
G:  add sum=sum,t1
H:  load item=[item + offset(next)]
I:  jump BB3'

END':
    add other=1,rT
    mod other=other,2
    mul qoff_exit=other,4
    set.qbase qoff_exit
    produce [2]=FALSE
    set.qbase qoff

END2:
    produce [5]=sum
```

(b) Thread 2

Figure 3.9: PS-DSWP Multi-Threaded Code after communication insertion

3.4 Speculative Parallel-Stage DSWP

Section 2.3.2 discussed how adding speculation can break dependences that inhibit parallelism. Speculative PS-DSWP (SpecPS-DSWP) adds speculation to the PS-DSWP algorithm and is heavily influenced by the Speculative DSWP work by Vachharajani et al [82], which was built on top of DSWP, not PS-DSWP. By building on top of PS-DSWP, SpecPS-DSWP can use speculation to remove inter-iteration dependences that prevent the extraction of data-level parallelism.

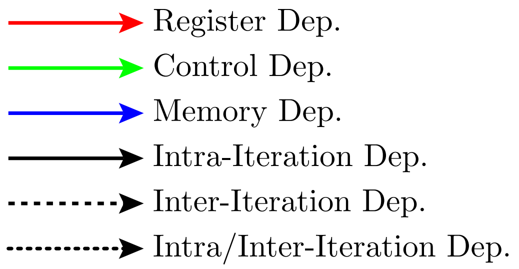
3.4.1 Determining a Thread Assignment

Algorithm 6 Partition

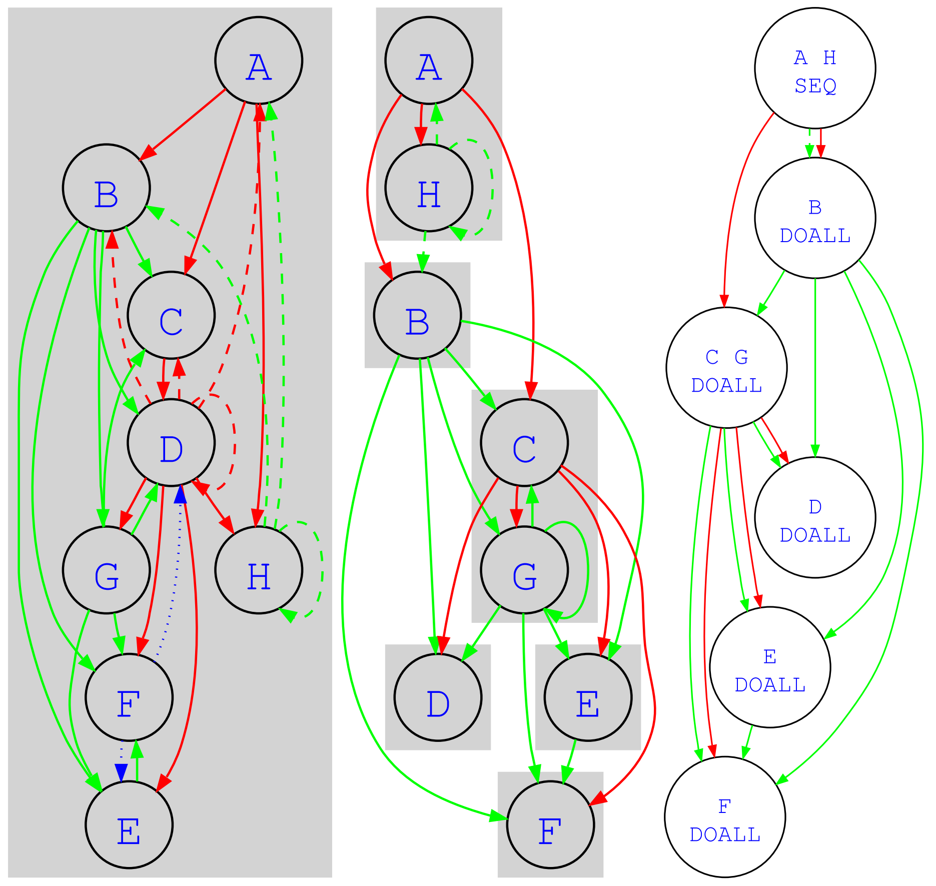
```
Require: Loop $L$, Threads $T$
 1: $PDG$ = BuildPDG($L$)
 2: Speculation $C$ = DetermineSpeculation($L$, $PDG$)
 3: $SPDG$ = BuildSpeculativePDG($L$, $C$)
 4: ThreadAssignment $A$, Replications $R$ = FormAndAssignSCCs($SPDG$, $T$)
 5: $C$ = DetermineNeededSpeculation($A$, $PDG$, $C$)
 6: $SPDG$ = RebuildSpeculativePDG($L$, $C$)
 7: $A$, $R$ = FormAndAssignSCCs($SPDG$, $T$)
 8: Versioning $V$ = DetermineMemoryVersioning($A$, $C$)
 9: Synchronization $S$ = DetermineSynchronization($A$, $L$)
10: return ($A$, $R$, $S$, $V$, $C$)
```

SpecPS-DSWP removes dependences from the PDG before partitioning in order to avoid complicating the already difficult problem of partitioning. Since speculation occurs before partitioning, it can remove dependences that occur in a forward direction, given the DSWP pipeline. This is unnecessary in SpecPS-DSWP, as the pipelined nature of a DSWP parallelization means that the only cost of forward synchronization is the overhead of the extra instructions. To avoid unnecessary misspeculation, after partitioning, only the set of speculations that remove dependences that flow backward in the pipeline are applied. The code is repartitioned after the set of needed speculations is computed, both to ensure that previously unreachable code is assigned to a thread and to give the partitioner the chance

```
   list *cur=head;
   int temp;
A: while (cur != NULL) {
B:   list *item=cur->item;
C:   while (item != NULL) {
D:     if (item->data < 0) abort();
E:     temp = -item->data;
F:     item->data = temp;
G:     item = item->next;
     }
H:   cur = cur->next;
   }
```



(a) Sequential C Code



(b) Non-Speculative PDG (c) Speculative PDG (d) Speculative DAG_{SCC}

Figure 3.10: SpecPS-DSWP Example Code

to better balance the effects of communication among threads. Steps related to forming the DAG_{SCC} and assigning SCCs to threads are unchanged from the PS-DSWP algorithm and are combined into a single step.

Step 1: Build PDG

The PDG formed by SpecPS-DSWP is the same as that formed by PS-DSWP except that false (anti- and output-) memory dependence are not inserted into the PDG. Any backward false memory dependence will be removed through appropriate memory versioning in the generated code. Figure 3.10b shows the PDG built for the example code in Figure 3.10a. Note that even though flow memory dependences exist from $F \xrightarrow{flow} D$ and $F \xrightarrow{flow} E$, the false memory dependences $F \xrightarrow{anti} D$, $F \xrightarrow{anti} E$, and $F \xrightarrow{output} F$ have not been included.

Step 2: Selecting Speculation

SpecPS-DSWP currently supports several speculation types. Three of these speculation types, biased branch, infrequent block, and silent store come from SpecDSWP, while the remaining two, memory value and loop-carried invariant, are novel to this dissertation. To avoid overspeculation, each speculation type has a threshold limit, T. For the purposes of this subsection, H represents the loop header.

Biased Branch Speculation A *Biased Branch* speculation predicts that a branch B with targets D_1 and D_2 does not go to a certain target. For a control flow edge weight function ω and operation weight function β, if $\omega(B \longrightarrow D_i)/\beta(B) <= T_{LocalBias}$, then the edge $B \longrightarrow D_i$ is speculated not taken. Unfortunately, SpecDSWP does not distinquish how often a branch is taken relative to the loop header. This is not sufficient for branches in inner loops, as they can often be heavily biased and yet still traverse the unbiased path with high probability per outer loop iteration. Therefore, in SpecPS-DSWP, a loop-sensitive control flow edge weight function δ is used as a second filter. For a loop L, if $\delta(B \longrightarrow D_i, L)/\beta(H) <= T_{LoopBias}$, then the edge

$B \longrightarrow D_i$ is allowed to be speculated. In practice, the loop threshold is set to the same percentage as the bias threshold. A threshold of 10% is used for both $T_{LocalBias}$ and $T_{LoopBias}$ in VELOCITY.

Infrequent Block Speculation An *Infrequent Block* speculation predicts that a basic block is not executed. For a given basic block BB and a loop-sensitive basic block weight function β, if $\beta(BB)/\beta(H) < T_{Frequency}$, then BB is speculated to be unreachable. A threshold of 1% is used in VELOCITY.

Silent Store Speculation A *Silent Store* speculation predicts that a value about to be stored is the same as that already in memory. Silent store speculation relies on a loop-sensitive value prediction function γ to predict how often the value being stored is the same as already exists in memory. If $\gamma(S) > T_{SilentStore}$ then the store is speculated as silent. A threshold of 1% is used in VELOCITY.

Memory Value Speculation In order to extract data-level parallelism, SpecPS-DSWP requires a way to speculate inter-iteration memory flow dependences. Since SpecDSWP avoids the use of hardware to detect misspeculation, memory value speculation is used instead of memory alias speculation to break memory flow dependences. A *Memory Value* speculation predicts that a load L will not be feed by a store S. The speculation can predict that the store feeds the load only during an iteration (inter-iteration speculation), except during an iteration (intra-iteration speculation), or ever (both intra- and inter-iteration speculation). Memory value speculation relies on a loop sensitive alias prediction function α_{INTRA} to determine the probability of aliasing intra-iteration and α_{INTER} to predict the probability of aliasing inter-iteration. If $\alpha_{INTRA}(L) <= T_{IntraAlias}$, then the intra-iteration portion is speculated not to exist. If $\alpha_{INTER}(L) <= T_{InterAlias}$, then the inter-iteration portion is speculated not to exist. A threshold of 10% for $T_{InterAlias}$ and 0% for $T_{IntraAlias}$ are used in VELOCITY. The reason for the lack of intra-iteration memory value speculation will be explained in Section 5.2.2.

Loop-Carried Invariant Speculation SpecPS-DSWP also introduce a new type of speculation, *Loop-Carried Invariant*. A loop-carried invariant speculation predicts that if a load L obtains a value from outside the current iteration, then the value will be the same as the one that existed in memory at the beginning of the last iterations. Loop-carried invariant speculation is useful for removing memory flow dependences to loads of global data structures that can change during an iteration, but are eventually reset to the value that they had at the beginning of the iteration. Loop-carried invariant speculation relies on a loop sensitive invariant prediction function λ to determine the probability of a load getting the correct value from memory T iterations ago. If $\lambda(L) <= T_{Variance}$, then L is speculated as loop-carried invariant. A threshold of 10% is use in VELOCITY.

Since memory value speculation and loop-carried invariant speculation can remove the same dependences, but require different transformations, they cannot both be applied at the same time. As the dependences removed by loop-carried invariant speculation subsume those removed by memory value speculation, when choosing speculation, loop-carried invariant speculation is determined before memory alias speculation, thus avoiding the problem.

Statically determining the prediction functions has been attempted for a loop-insensitive branch predictor [88]. In general, though, the compiler cannot easily predict the dynamic effects that these functions seek to capture. Thus, profiling is used to gather the relevant data for each function. The profiling infrastructure is described in Section 3.4.3.

For the example in Figure 3.10, biased branch speculation is applied to the loop exit branch D, and memory value speculation is applied to the $F \xrightarrow{flow} D$ and $F \xrightarrow{flow} E$ memory dependences. The speculative PDG and speculative DAG_{SCC} formed by applying these speculations to the non-speculative PDG are shown in Figure 3.10c and Figure 3.10d respectively.

Step 3: Build Speculative PDG

The speculative PDG is built using the same process as step 1, except that a Speculative CFG is used as the basis rather than the original, non-speculative CFG. The Speculative CFG is formed by removing all control edges that have been speculated not taken by Biased Branch and Infrequent Block speculation. After the Speculative PDG is formed from the Speculative CFG, dependences are removed on a per speculation basis as follows:

Silent Store Speculation If a store S is speculated as silent, then all outgoing flow memory arcs from S are removed from the Speculative PDG.

Memory Value Speculation If a memory value speculation predicts that the intra-iteration dependence will not manifest, then for an EITHER memory flow dependence, the loop-carriedness is speculated to INTER, while for a INTRA memory flow dependence, the dependence is removed from the Speculative PDG. If a memory value speculation predicts that the inter-iteration dependence will not manifest, then for an EITHER memory flow dependence, the loop-carriedness is speculated to INTRA, while for a INTER memory flow dependence, the dependence is removed from the Speculative PDG.

Loop-Carried Invariant Speculation If a load L is speculated as loop-carried invariant, then all INTER memory flow dependences with destination L are removed from the Speculative PDG. All EITHER memory flow dependences with destination L are marked INTRA.

Step 4: Form and Assign SCCs

The DAG_{SCC} is formed from the Speculative PDG and partitioned just as in PS-DSWP.

Step 5: Determine Needed Speculation

Once the Speculative PDG and Speculative CFG have been formed, the partitioning process proceeds as normal to produce a thread assignment. For a given thread assignment, speculation that only breaks forward dependences can be removed to reduce the misspeculation rate.

Choosing the amount of speculation needed to produce a good parallelization is a hard problem. Most existing techniques overspeculate to ensure that parallelism exists, using feedback directed compilation to prune unprofitable parallelizations [22]. To give the partitioner as much freedom as possible, SpecPS-DSWP also speculates more dependences than may be necessary to produce good parallelism. However, unlike speculative IMT and CMT techniques, SpecPS-DSWP can undo any speculations that does not introduce backwards dependences. The details of how to unspeculate dependences are beyond the scope of this dissertation, see Vachharajni et al [82] for more detail. The main point is that the ability to unspeculate dependences allows SpecPS-DSWP to avoid unnecessary misspeculation on an otherwise forward dependence. Additionally, in the case where a dependence can be removed by multiple types of speculation, SpecPS-DSWP can choose to keep only one speculation, again avoiding unnecessary misspeculation, if, for example, an alias speculation can be used instead of a biased branch speculation.

For Figure 3.10, if E and F are placed in the same SEQUENTIAL stage, then the memory speculation that removed the $F \xrightarrow{flow} E$ dependence can be unspeculated, as it is no longer needed. If E and F are placed in the same DOALL stage, then only the INTER portion of the dependence needs to be speculated, while the INTRA portion can be unspeculated.

Step 6: Rebuild Speculative PDG

After dependences are unspeculated, particularly biased branch and infrequent block speculations, previously unreachable code may be made reachable and will need to be integrated into the partition. Additionally, by unspeculating forward dependences, new inter-thread communication may be inserted. To handle this, after unspeculation, the Speculative PDG is reformed exactly as in Step 3, but with only the remaining needed speculation.

Step 7: Form and Assign SCCs

The same partitioner is used to repartition the code using the Speculative PDG and its DAG_{SCC} recreated by Step 6.

Step 8: Determine Memory Versions

Once a thread assignment has been established and the set of speculations is fixed, SpecPS-DSWP must determine the memory version that each memory operation and external function call will execute in. At the outermost loop level, memory versions are used to manage speculative state so that rollback can occur. Inside the loop, memory versions are used to break false memory dependences. Additionally, because the SpecPS-DSWP code generator relies on software to detect misspeculation, memory versions are used to ensure that any unanticipated, new memory flow dependences that occur in an iteration that misspeculates do not prevent misspeculation from being detected. The details of determining memory versions are beyond the scope of this dissertation and can be found in Vachharajani et al[82].

Step 9: Determine Synchronization

This step is unchanged.

3.4.2 Realizing a Thread Assignment

A SpecPS-DSWP thread assignment is realized by first reifying the speculation and then applying the PS-DSWP algorithm to the resulting code. After PS-DSWP multi-threaded

Algorithm 7 Speculative MultiThreadedCodeGeneration

```
Require: Loop $L$, (ThreadAssignment $A$, Replication $R$, Synchronization $S$, Versioning
    $V$, Speculation $C$)
1: CopyCodeForRecovery($L$)
2: ApplySpeculation($L$, $C$)
3: Threads $T$ = MultiThreadedCodeGeneration($L$, $A$, $S$)
4: ApplyMemoryVersioning($V$)
5: FinishRecoveryCode($L$, $T$)
```

code generation is done, checkpoint and recovery code are inserted into each thread and the *Commit Thread*.

Step 1: Copy Code For Recovery

Because SpecPS-DSWP can speculate intra-iteration dependences, it must have an unaltered copy of the loop body to use upon misspeculation. In this step, the commit thread is created, and a copy of the loop body is placed in it.

Step 2: Apply Speculation

```
   list *cur=head;
   int temp;
A: while (cur != NULL) {
B:   list *item=cur->item;
C:   while (item != NULL) {
D:     if (item->data < 0) abort();
E:     temp = -item->data;
F:     item->data = temp;
G:     item = item->next;
     }
H:   cur = cur->next;
   }
```

(a) Sequential C Code

```
    list *cur=head;
    int temp;
(1) while (cur != NULL) {
(1)   list *item=cur->item;
(2)   while (item != NULL) {
(2)     temp2 = item->data;
(3)     if (item->data!=temp2)
          MISSPEC;
(2)     if (temp2 < 0) MISSPEC;
(2)     temp3 = item->data;
(3)     if (item->data!=temp3)
          MISSPEC;
(2)     temp = -temp3;
(2)     item->data = temp;
(2)     item = item->next;
      }
(1)   cur = cur->next;
    }
```

(b) Single-Threaded Code with Speculation (Thread Assignment in parenthesis)

Figure 3.11: Single Threaded Speculation

Each type of speculation is applied to the code as described below:

Biased Branch Speculation To realize a biased branch speculation, the compiler inserts a new basic block on each control flow edge from the branch to each speculated target. The block signals misspeculation and then waits to be redirected to recovery code, either by consuming from any empty queue or other infinite length action. To facilitate accurate analysis by the compiler, the block then branches to one of the non-speculated targets if one exists, otherwise it is marked as an exit. This keeps the branch from introducing more region exits or control dependences that would cause conservative compiler analysis.

Infrequent Block Speculation To realize an infrequent block speculation, the compiler has two options. It can replace the block with a misspeculation signal, followed by the original fall through branch. Unfortunately, this retains control dependences in non-speculated code. To avoid this, the infrequent block speculation can be mapped to a set of biased branch speculations, one for each incoming control flow edge.

Silent Store Speculation To realize a silent store speculation, the compiler inserts a load of the same memory and a branch to compare the loaded value with the one about to be stored. If they are equal, the branch jumps around the store, otherwise it signals misspeculation in the same way biased branch speculation does.

Memory Value Speculation To realize a memory value speculation from a store S to a load L, the compiler creates a copy of the load L', and updates the memory analysis to remove the dependence from S to L'. The compiler then inserts code to compare the value loaded from L' to that loaded by L. If the values differ, the code branches to a block that indicates misspeculation in the same way biased branch speculation does.

Loop-Carried Invariant Speculation To realize a loop-carried invariant speculation, the compiler employs the same methodology as Memory Value speculation. That is, the copy of the load is made and all incoming memory flow dependences are removed. A compare of the speculative load with the original load is used to determine if misspeculation occurred.

To signify to the hardware the point at which a value should be read from committed state rather than traverse the memory hierarchy, two new instructions are added. A $set.lvp$ instruction is used at the beginning of each iteration to indicate where a $load.invariant$ should stop traversing the memory version hierarchy and proceed directly to committed memory.

Figure 3.11 shows the result of applying biased branch speculation to branch D, and memory value speculation to $F \xrightarrow{flow} D$ and $F \xrightarrow{flow} E$. *MISSPEC* represents a general code sequence that will send misspeculation to the commit thread and then infinite loop until resteered by the commit thread.

Steps 3: Multi-Threaded Code Generation

Once speculation has been realized in the single-threaded code, the PS-DSWP code generation algorithm is applied to the loop.

Step 4: Apply Memory Versioning

SpecPS-DSWP relies upon a version memory system with hierarchical memory versions. A parent MTX is a virtual container for many subTX memory versions which are ordered from oldest to youngest. A subTX can contain at most one MTX or be used to execute a sequence of memory operations. At any point in time each thread is executing in a specific memory version, denoted $(MTX, subTX)$. The memory version $(0, 0)$ is special and represents nonspeculative memory. Stores are modified to store into the current memory version. Loads are modified to load from the closest memory version older than or equal to the current memory version. Thus, loads can potentially traverse all direct and indirect

Queue Instructions		
Instruction	Arguments	Description
`set.qbase`	V	Sets the queue base $Base$ for a thread to the given value V.
`produce`	[Q] = V	Pushes value V onto the tail of queue $Q + Base$ if it is not full. If the queue is full, the instruction stalls until room is available.
`consume`	R = [Q]	Pops a value off of the head of queue $Q + Base$ if there are values in the queue and places the value in register R. If the queue is empty, the instruction stalls until one is available.
`consume.poll`	P,R = [Q]	For queue $Q + Base$, if there is a value in the queue, it is popped off the queue and placed in register R and P is set to 1. If the queue is empty, the instruction writes 0 into P and the value of R is undefined. This instruction does *not* stall if the queue is empty.
`queue.flush`	Q	Flushes all values in the queue $Q + Base$, leaving it empty.
`resteer`	Q, A	Asynchronously resteer the thread that produced into queue $Q + Base$ to address A.
Version Memory Instructions		
`allocate`	(M, S)	Returns an unused MTX ID setting its parent version to memory version (M, S).
`enter`	(M, S)	Enter the specified MTX M and subTX S.
`commit_stx`	(M, S)	Commit the subTX S into the parent MTX M.
`commit_mtx`	M	Commit the MTX M into the parent (MTX, subTX) that it was created under. If the parent memory version is $(0, 0)$ then commit into nonspeculative state. S emphmust be the oldest child of M.
`rollback`	M	All the stores from the specified MTX M and any subordinate subTXs and MTXs will be discarded, and the MTX is deallocated. Threads must issue an `enter` to enter a legitimate MTX or committed state.
`set.lvp`	(M, S)	Sets the load variant point register to the specified MTX M and subTX S.
`load.variant`	R = [Addr]	Performs a load that respects the load variant point register, relative to the current memory version.

Table 3.3: SpecPS-DSWP ISA Extensions

parents, older siblings, and older siblings children to find the oldest stored value. An efficient implementation of version memory is beyond the scope of this dissertation, but as in hardware transactional memory system, versioned memory can use extra bits in the caches to hold version related information [82].

For each memory instruction and external function call, a memory version *enter* instruction is inserted ahead of it. Additionally, for any loop body with instructions in more than 1 memory version, a new hierarchical memory transaction (MTX) is *allocated*. Its parent is the last executing memory transaction and when it commits it will commit its state into that transaction. The purpose of these hierarchical transactions is to allow all of

```
list *cur=head, item;
int mtx, stx;
mtx = allocate(0, 0);
stx = 0;
produce(4, mtx);
produce(5, mtx);
produce(6, cur);
produce(7, cur);

int iter = r_T

while (cur != NULL) {
  int qset = iter % 2;
  int qoff = qset * 18;
  set_qbase(qoff)
  produce(17, TRUE);

  produce_reg_chkpt();
  enter(mtx, stx);
  iter++;

  item = cur->item;
  produce(8, item);
  produce(9, item);

  cur = cur->next;
  produce(10, cur);
  produce(11, cur);

  stx += 1;
  produce(1, OK);
}
produce(1, EXIT);
```

(a) Thread 1

```
int mtx = consume(4)
int stx = 0;

set_qbase(r_T  * 18);

list *cur = consume(6);
while (cur != NULL) {
  int exit = consume(17);
  if (exit == TRUE) {
    for (int i=0; i<R; i++) {
      set_qbase(i * 18);
      produce(17, FALSE);
    }
    break;
  }

  produce_reg_chkpt();
  enter(mtx, stx);

  mtxsave = mtx;
  stxsave = stx;
  mtx = allocate(mtx, stx);
  stx = 0;
  produce(12, mtx);

  list *item = consume(8);
  while (item != NULL) {
    enter(mtx, stx);

    int temp2 = item->data;
    produce(13, temp2);
    if (temp2 < 0) {
      produce(2, MISSPEC);
      wait();
    }

    int temp3 = item->data;
    produce(14, temp3);
    int temp = -temp3;

    enter(mtx, stx+1);
    item->data = temp;
    item = item->next;
    produce(15, item);

    stx += 2;
    produce(16, MEM_SYNC);
  }

  cur = consume(10);

  mtx = mtxsave;
  stx = stxsave;
  stx += 1;
  produce(2, OK);
}
produce(2, EXIT);
```

(b) Thread 2

```
int mtx = consume(5)
int stx = 0;

int iter = r_T
list *cur = consume(7);
while (cur != NULL) {
  int qset = iter % 2;
  int qoff = qset * 18;
  set_qbase(qoff)

  produce_reg_chkpt();
  enter(mtx, stx);
  iter++;

  mtxsave = mtx;
  stxsave = stx;
  mtx = consume(12);
  stx = 0;

  list *item = consume(9);
  while (item != NULL) {
    enter(mtx, stx);

    int temp2 = consume(13);
    if (item->data != temp2) {
      produce(3, MISSPEC);
      wait();
    }

    int temp2 = consume(14);
    if (item->data != temp2) {
      produce(3, MISSPEC);
      wait();
    }

    item = consume(15);
    int unused = consume(16);

    commit_stx(mtx, stx);
    commit_stx(mtx, stx+1);
    stx += 2;
  }

  commit_mtx(mtx);

  cur = consume(11);

  mtx = mtxsave;
  stx = stxsave;
  stx += 1;
  produce(3, OK);
}
produce(3, EXIT);
```

(c) Thread 3

Figure 3.12: SpecPS-DSWP Parallelized Code without Misspeculation Recovery

the memory versions inside the loop (subTXs) to be summarized by a single MTX, so that all version numbers assigned to operations can be statically determined. If hierarchy were not used, then the unbounded number of memory versions that the loop would use would force the compiler to communicate memory versions between threads, potentially creating backwards dependences [82].

Figure 3.12 shows the memory version code inserted for the code from Figure 3.11 after the PS-DSWP code generation algorithm has been applied. In the interests of space the code is shown in high-level C-like code.

Step 5: Finish Recovery Code

```
while (true) {
  move_to_next_memory_version();
  send_checkpoint(commit_thread);

  status = execute_loop_iteration();

  produce(commit_thread, status);
  if (status == EXIT)
    break;
  else if (status == MISSPEC)
    wait_for_resteer();
  else if (status == OK)
    continue;

RECOVERY:
  produce_resteer_ack(commit_thread);
  flush_queues();
  regs = receive_checkpoint(commit_thread);
  restore_registers(regs);
}
```

(a) Worker Thread

```
do {

  regs = receive_checkpoints(threads);

  status = poll_worker_statuses(threads);

  if (status == MISSPEC) {
    resteer_threads(threads);
    consume_resteer_acks(threads);
    rollback_memory();

    regs = execute_loop_iteration(regs);

    send_checkpoints(threads, regs);
  } else if (status == OK || status == EXIT)
      commit_memory();

  move_to_next_memory_version();

} while (status != EXIT);
```

(b) Commit Thread

Figure 3.13: Interaction of Parallel Code and Commit Thread

Once the code for each thread is finished, the only remaining step is to create code necessary for recovery from misspeculation. Each worker thread will have the same basic layout, shown in Figure 3.13a [82]. Essentially, at the beginning of an iteration, a memory checkpoint is created by moving to the next memory version and a register checkpoint is sent to the *commit thread*. The thread then executes the iteration normally. In the event of misspeculation, the thread waits for the commit thread to *resteer* it to recovery code.

The commit thread, shown in Figure 3.13b never executes speculatively. After receiving the register checkpoints at the beginning of each iteration, it waits for status updates from the threads executing the current iteration. Status messages are sent using the queue system. Since only one thread may detect misspeculation while others infinite loop, the Commit Thread is not guaranteed to receive status message from all threads. The *consume.poll* instruction is used by the commit thread to repeatedly poll the status queues for the threads until *all* of them have sent an OK/EXIT status or any *one* has sent a MISSPEC status.

If a thread signals misspeculation the commit thread uses an asynchronous *resteer* instruction to direct each thread to statically created recovery code. The system returns to a known good state by:

1. Discarding speculative writes by rolling back the version memory system via the *rollback* instruction.
2. Flushing the queue system, via a *queue.flush* operation for each queue.
3. Restoring registers from an iteration checkpoint.

If misspeculation is not detected in an iteration, the commit thread commits the speculative state to non-speculative memory via a sequence of *commit* instructions [82].

3.4.3 Loop-Aware Profiling

The ability to speculate a dependence requires the ability to predict how often a property of the dependence occurs. For example, if a load always reads the same value, then the register it defines can be speculated to always have that value. The general solution to creating predictors is to profile the application, using the results of profiling to guide prediction functions. While these profiles are useful in traditional optimizations, to be useful in loop parallelization it is necessary that they be able to distinguish how often a dependence manifests per loop iteration or whether a dependence is intra-iteration versus inter-iteration [22, 89].

VELOCITY relies on several profilers to create predictors for each type of speculation. Unlike existing approaches that rely on path-profiles to obtain loop-sensitive results for alias profiling [89], the VELOCITY compiler uses a novel methodology to maintain and collect only loop-sensitive profile information needed for speculation.

To make the profilers loop-sensitive, the profiling framework keeps track of the set of loops entered into in a stack so that it can use the *loop stack* to create loop-sensitive profiles. On entry to a loop, its context is pushed onto the stack, while on exit from the loop the top of the stack is popped off and discarded. In the current implementation, loops inside recursive functions are not profiled. Thus, each loop exists at most once in the loop context stack. Since DSWP cannot currently be applied to loops inside recursive functions, this limitation does not adversely affect VELOCITY. To efficiently test whether the profile event has occurred during a loop's current iteration, a timestamp is maintained by the profiling system and for each item in the loop stack. At every loop backedge, the current timestamp counter is incremented by one and the timestamp at the top of the loop stack is replaced with the current timestamp. Initially, the loop stack contains a pseudo-loop that represent code that does not execute in any loop and a timestamp of 0.

SpecPS-DSWP relies on several profilers detailed below to give loop-sensitive results.

Branch Profile

A branch or edge profile gives the absolute number of times each control flow edge is traversed. In an insensitive profiling environment, branches are instrumented to determine their target, producing a set of $(branch, target, count)$ tuples that guide many optimizations. In a loop-sensitive branch profile, the count for the $(branch, target, count)$ tuple is incremented for each loop in the loop stack, if it has not already been incremented during for that loop's current iteration.

When a branch executes, the loop stack is traversed from top to bottom checking to see if the branch edge has been taken for the loop's current iteration. This is accomplished by maintaining a profile event (e.g. branch edge taken) to timestamp map for each loop that indicates the last iteration the profile event was incremented. If the last timestamp is not equal to the current timestamp, then the branch edge counter is incremented. This traversal can exit on the first timestamp that is equal to the current timestamp, as it is guaranteed that all outer containing loops lower in the stack also have the same timestamp. The final result of the profiler is a set of $(loop, branch, target, count)$ tuples. These tuples and maps can be efficiently implemented using arrays by statically assigning unique, consecutive identifiers to loops and $(branch, target)$ pairs.

Silent Store Value Profile

A *Slient Store Value Profile* indicates the number of times that a store is silent [40]. That is, the number of times the value being stored equals the value already in memory. It is obtained by taking the value of each stored register value just before it is written to memory and comparing it to the value in memory at the address being stored to. If they are equal, the value profile increments an *areEqual* counter for the store, otherwise, it increments an *areDifferent* counter.

To make the store value profile loop sensitive, essentially the same actions are performed as in the branch profiling infrastructure. Instead of keeping track of a $(branch, target)$ count, the store value profile keeps track of $(areEqual, areDifferent)$ counts. Additionally, two $(loop, store) \longrightarrow Timestamp$ maps are used to ensure that $areEqual$ and $areDifferent$ are incremented once per iteration. The final result of the profiler is a set of $(loop, store, areEqual, areDifferent)$ tuples. These tuples can be efficiently represented with arrays by statically assigning unique consecutive identifiers to loops and stores.

Memory Alias Profile

A *Memory Alias Profile* indicates the number of times that a $(store, load)$ memory flow dependence manifests itself. Since a store or load may write or read multiple bytes, a particular load may increment several $(store, load)$ pairs, though this is rare in practice. The profile is obtained by keeping a map from each address to the last store to write the address. A store instruction updates the map entries for the addresses it write, while a load instruction reads the set of last stores for addresses it reads. The load then increments the set of unique $(store, load)$ tuples.

To make the alias profile loop sensitive, the loop timestamp infrastructure from the branch profiling is reused. However, besides keeping track of the last iteration's timestamp, the loop stack also keep's track of the loop's invocation timestamp. This will be used to determine if an alias dependence is intra-iteration or inter-iteration.

The address to store map is changed to an address to $(store, timestamp)$ map. A store instruction updates the map with itself and the current timestamp. On a load, the set of unique $(store, timestamp)$ tuples is read out of the accessed addresses. For each $(store, timestamp)$ pair, the loop stack is traversed from top to bottom until the store's timestamp is greater than the loop's invocation timestamp. This is the first loop which contains both the store and the load instructions and is the loop around which the dependence is either intra-iteration or inter-iteration. This is determined by comparing the store's timestamp with the loop's current iteration timestamp. If they are equal, the dependence is intra-iteration and the count for the $(loop, store, load, INTRA)$ tuple is incremented. A $(loop, store, load) \longrightarrow Timestamp$ map is used to ensure this increment occurs only once per iteration. Otherwise, the dependence is inter-iteration, represented as $(loop, store, load, INTER)$, which is also incremented if it has not already been incremented this iteration.

The final result of the profiler is a set of $(loop, store, load, INTRA|INTER, count)$ tuples. These tuples can be efficiently represented with arrays by statically assigning unique consecutive identifiers to loops, stores, and loads. In practice, the map from address to $(store, timestamp)$ can represented as a Hash Map of 4Kb pages, where each page element is a 64-bit $(store, timestamp)$ tuple with a 20-bit store identifier and 44-bit timestamp counter.

To determine the set of intra-iteration alias dependences, the compiler must consider intra-iteration alias dependences of the loop itself, as well as both the intra- and inter-iteration dependences of all loops contained, either directly or indirectly, by the loop.

Loop-Invariant Value Profile

A *Loop-Invariant Value Profile* is a purely loop sensitive profile. The purpose of the profile is determine how often a load retrieves a value from memory that, if it is inter-iteration, is the same as the previous iteration(s). If the load is feed by a store in the current iteration of the loop that first contains both instructions, it is not counted. The profiler is essentially the same as the alias profiler, except that the loop stack is modified to keep around a loop specific version of memory (an address to value map) that corresponds to the previous iteration's memory. On a load instruction feed by a store at least 1 iteration distant, the previous iteration's memory value for the address is compared to the current value in memory. If they differ, the $areDfiferent$ value is incremented, otherwise the $areEqual$ value is incremented.

Maintaining a separate copy of memory for each loop in the program is prohibitively expensive, thus only loops that account for more than 30% of the program's runtime and that are at most 4 levels deep in the loop hierarchy are profiled for loop-invariance.

The final result of the profiler is a set of $(loop, load, areEqual, areDifferent)$ tuples. These tuples can be efficiently represented with arrays by statically assigning unique consecutive identifiers to loops and loads. To determine if a loop is invariant with respect to a loop, the compiler must consider the invariance of the load in the loop itself, as well as in all loops contained, either directly or indirectly, by the loop.

3.5 Interprocedural Speculative Parallel-Stage DSWP

Section 2.4 discussed how adding interprocedural scope was beneficial to the extraction of parallelism. Interprocedural SpecPS-DSWP (iSpecPS-DSWP) is a novel technique that adds both interprocedural analysis and optimization scope to the SpecPS-DSWP algorithm, allowing it to optimize larger loops than existing parallelization techniques.

3.5.1 Determining a Thread Assignment

No additional steps are needed for iSpecPS-DSWP, however several existing steps change to become aware of the interprocedural nature.

Steps 1, 3, & 6: Build PDG

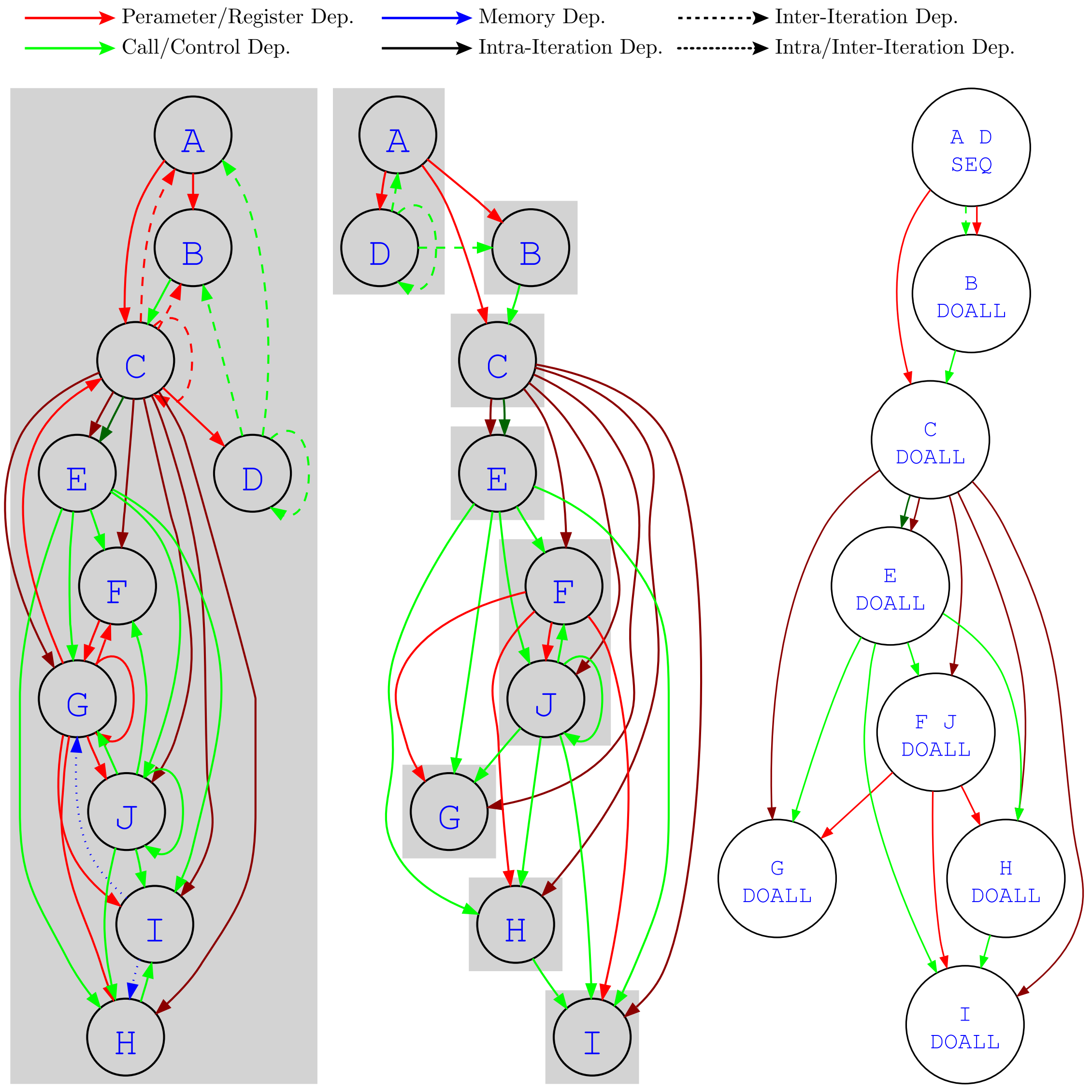
To facilitate interprocedural scope, the PDG for not just the local loop, but any reachable procedures must be formed and connected. The algorithm in this dissertation forms an Interprocedural PDG (iPDG), a derivative of the System Dependence Graph (SDG) [35, 66], an interprocedural PDG representation originally used for program slicing. As in the SDG, the iPDG contains all local dependences among operations in the loop and any procedures called directly or indirectly from operations in the loop.

To represent interprocedural dependences, the SDG uses actual parameters at the call site and formal parameters at the callee's entrance and exit. Call sites are expanded so that actual in parameters connect to formal in parameters and formal out parameters connect to actual out parameters. To handle memory data dependences, references to memory variables that are modified in the procedure or its descendants appear as separate formal and actual arguments. To handle control dependences, a single call dependence is inserted between the call and the entry node of the called procedure. The SDG forces all interprocedural dependences to occur among these formal/actual pairings so that it can be used for efficient slicing through the addition of transitive dependence edges. Transitive dependence

```
    void main() {
      list *cur=head;
A:    while (cur != NULL) {
B:      list *item=cur->item;
C:      work(item);
D:      cur = cur->next;
      }
    }
```

```
E: void work(list *item) {
F:   while (item != NULL) {
G:     if (item->data < 0) abort();
H:     int temp = -item->data;
I:     item->data = temp;
J:     item = item->next;
     }
   }
```

(a) Sequential C Code



(b) Non-Speculative PDG (c) Speculative PDG (d) Speculative DAG_{SCC}

Figure 3.14: iSpecPS-DSWP Example Code

edges connect actual outs with only the actual ins that they are transitively dependent upon. They are used by slicing algorithms to perform an accurate interprocedural slice using just and up and down pass over the SDG, rather than iterating to convergence [35].

The iPDG uses the same formal to actual relation as the SDG to represent actual parameter dependences between call site and callee. In addition to adding the call dependence from the call site to the procedure's entry, the iPDG adds the call dependence from the call site to every operation in the procedure. This is required so that later phases of the DSWP algorithm can properly replicate the branches and calls so that each operation in a thread can be executed in the correct condition of execution. Finally, DSWP does not use transitive dependence edges and so does need them in the iPDG. Because of this, the iPDG does not need to force memory dependences to indirect through pseudo formal/actual pairs, instead allowing interprocedural memory dependences to connect directly. In particular, memory dependences are calculated based on the separate results on an interprocedural context-sensitive pointer analysis [14, 53].

Determining the loop-carriedness of each edge in the iPDG occurs as follows:

Register Dependence If the register dependence is in the DSWPed loop, then its loop-carriedness is computed as in Step 1 of Section 3.3.1. Otherwise, it occurs in a called procedure, and is marked as INTRA.

Parameter Dependence A parameter dependence is always marked as INTRA, since it cannot be live around the loop backedge.

Control Dependence If the control dependence is in the DSWPed loop, then its loop-carriedness is computed as in Step 1 of Section 3.3.1. Otherwise, it occurs in a called procedure, and is marked INTRA.

Call Dependence Like a parameter dependence, a call dependence cannot involve the loop backedge, so it is marked INTRA.

Memory Dependence The loop-carriedness of a memory dependence is computed as in Step 1 of Section 3.3.1.

Figure 3.14 show the non-speculative and speculative PDG's for the same code as in Figure 3.10 except that the inner loop is now inside a function, requiring an interprocedural scope to optimize.

Step 9: Determine Synchronization

First, the new types of dependences arcs must be synchronized properly. Parameter dependences are treated just like register dependence for the purposes of synchronization. Similarly, call dependences are treated just like control dependences for the purposes of synchronization. In particular, just as all indirect control dependences become cross-thread dependences, so do all indirect call dependences. This ensures that all necessary control flow, both branches and calls, exists to allow for the correct condition of execution.

Ideally, communication could be optimized across the merged control flow graphs of the reachable procedures, also called the supergraph [52]. Unfortunately, the algorithms used to optimize communication are on the order of n^3 [54], and the supergraph usually contains tens of thousands of nodes. To avoid excessive compile times, each procedure is optimized separately. That is, only the dependences that cross threads local to a procedure are optimized together. In practice, this means that only register and control dependences are optimized, as most memory flow arcs are interprocedural.

3.5.2 Realizing the Thread Assignment

A iSpecPS-DSWP thread assignment is realized by first reifying the speculation as in SpecPS-DSWP and then applying the multi-threaded code generation algorithm locally to each procedure. After PS-DSWP multi-threaded code generation is done, the SpecPS-DSWP commit thread is created.

Figure 3.15 and 3.17 shows the results of applying iSpecPS-DSWP using essentially the same thread assignment and speculation as used in Section 3.4.

Step 1: Create Recovery Code

To allow for proper execution of an iteration that misspeculates, all code in the loop and reachable procedures is cloned. Procedure calls inside the cloned region are redirected to cloned copies. This ensures that all code that a recovery iteration can execute remains intact.

Step 2: Apply Speculation

```
   void main() {
     list *cur=head;
A:   while (cur != NULL) {
B:     list *item=cur->item;
C:     work(item);
D:     cur = cur->next;
     }
   }

E: void work(list *item) {
F:   while (item != NULL) {
G:     if (item->data < 0) abort();
H:     int temp = -item->data;
I:     item->data = temp;
J:     item = item->next;
     }
   }
```

(a) Sequential C Code

```
    void main() }
      list *cur=head;
(1)   while (cur != NULL) {
(1)     list *item=cur->item;
(2)     work(item);
(1)     cur = cur->next;
      }
    }

(2) void work(list *item) {
(2)   while (item != NULL) {
(2)     temp2 = item->data;
(3)     if (item->data!=temp2)
          MISSPEC;
(2)     if (temp2 < 0) MISSPEC;
(2)     temp3 = item->data;
(3)     if (item->data!=temp3)
          MISSPEC;
(2)     temp = -temp3;
(2)     item->data = temp;
(2)     item = item->next;
      }
    }
```

(b) Single-Threaded Code with Speculation (Thread Assignment in parenthesis)

Figure 3.15: Single Threaded Speculation

All speculation requires only local changes to the IR, so this step is unchanged from SpecPS-DSWP. Figure 3.15 shows the results of applying speculation to the unmodified code from Figure 3.14.

Step 3: Multi-Threaded Code Generation

The first step of code generation is to create thread specific procedures for each parallelized procedure. Multi-Threaded Code Generation then proceeds in turn on each procedure and the DSWPed loop. For each procedure and the DSWPed loop, the local CFG is created just as in Step 1 of Section 3.3.2. For the DSWPed loop, the CFG is created for it exactly as in Step 1 of Section 3.3.2. For the procedures, the START and END blocks correspond to procedure entry and exit respectively. Once the CFGs are created, the instructions in the procedure are moved to the cloned basic blocks.

Synchronization is then inserted for the synchronization points that occur locally in each procedure or the DSWPed loop. For register, control, and memory dependences this is handled exactly as in Step 3 of Section 3.3.2. Parameter dependences are communicated in the same way that register dependences are. Call dependences are handled similarly to control dependences, in that they are cloned into the destination of the dependence.

Finally, control flow edges are redirected, just as in Step 4 of Section 3.3.2. However, calls to procedures in the parallelized region must also be redirected. In particular, the target of a non-external call is redirected to the thread-specific version of that procedure. iSpecPS-DSWP ensures that every procedure exists in each thread, even if it empty, relying on a subsequent pass of inlining to remove empty procedures. Additionally, a thread specific procedure call may not pass all the parameters that the original did. The compiler can set the unused parameters to dummy values or, more optimally, collapse the remaining parameters by removing the unused ones as part of redirecting the call.

One complication to the DSWP MTCG alogorithm arises when the original sequential code of a procedure uses a local stack variable and its operations are assigned to multiple threads. This segments the live range of the local variable to span multiple threads. If the memory continues to be allocated on the local stack of the first thread, it would need to

```
void f(list *item) {
  int flag = false;
  g(item, &flag);
  h(item, &flag);
}
```
(a) Sequential Code

```
void f1(list *item) {
  int *flagptr = malloc(4);
  produce(1, item);
  produce(2, flagptr);
  g(item, flag);
  produce(3, SYNCH);
}
```
(b) Thread 1

```
void f2() {
  list *item = consume(1);
  int *flagptr = consume(2);
  h(item, flagptr);
  consume(3);
  free(flagptr);
}
```
(c) Thread 2

Figure 3.16: Stack Code Example

wait for later threads to communicate to it that they are done using the memory to avoid later threads using memory that has been deallocated by the first thread. This introduces an illegal backward communication into the DSWP pipeline. The alternative of having the last thread allocate the memory on its stack avoids the problem of backwards dependences, but encounters the same problem upon creating the memory, as the value of the local memory must be communicated backwards to prior threads before they can reference the segment. iSpecPS-DSWP solves this problem by changing stack allocated variables in the local variable segment that are referenced across multiple threads into heap-allocated variables. The first thread to use the segment will create a heap allocated memory location (via a thread-safe call to `malloc`) whose value can be communicated to later threads. The last thread to use the segment will destroy the heap allocated memory (via a thread-safe call to `free`) after all prior threads have finished the function and send a token to the last thread indicating that they will no longer access the memory.

Figure 3.16 illustrates a simple example for a function `f`. After partitioning, the `flag` variable is split across multiple threads. To ensure that both threads can properly access it, it is first `malloc`ed in Thread 1 and its address sent to Thread 2. When Thread 1 finishes the function, it sends a synchronization token to Thread 2 to indicate that it is no longer using the memory, allowing Thread 2 to free it.

Queue Instructions		
Instruction	Arguments	Description
`set.qbase`	V	Sets the queue base $Base$ for a thread to the given value V.
`produce`	[Q] = V	Pushes value V onto the tail of queue $Q + Base$ if it is not full. If the queue is full, the instruction stalls until room is available.
`consume`	R = [Q]	Pops a value off of the head of queue $Q + Base$ if there are values in the queue and places the value in register R. If the queue is empty, the instruction stalls until one is available.
`consume.poll`	P,R = [Q]	For queue $Q + Base$, if there are is a value in the queue, it is popped off the queue and placed in register R and P is set to 1. If the queue is empty, the instruction writes 0 into P and the value of R is undefined. This instruction does *not* stall if the queue is empty.
`queue.flush`	Q	Flushes all values in the queue $Q + Base$, leaving it empty.
`resteer`	Q, A	Asynchronous resteer the thread that produced into queue $Q + Base$ to address A.
Version Memory Instructions		
`allocate`	(M, S)	Returns an unused MTX ID setting its parent version to memory version (M, S).
`enter`	(M, S)	Enter the specified MTX M and subTX S.
`commit_stx`	(M, S)	Commit the subTX S into the parent MTX M.
`commit_mtx`	M	Commit the MTX M into the parent (MTX, subTX) that it was created under. If the parent memory version is $(0, 0)$ then commit into nonspeculative state. S emphmust be the oldest child of M.
`rollback`	M	All the stores from the specified MTX M and any subordinate subTXs and MTXs will be discarded, and the MTX is deallocated. Threads must issue `enter` to enter a legitimate MTX or committed state.
`set.lvp`	(M, S)	Sets the load variant point register to the specified MTX M and subTX S.
`load.variant`	R = [Addr]	Performs a load that respects the load variant point register, relative to the current memory version.
`get_mtx`	R	Sets R to the current MTX.
`get_stx`	R	Sets R to the current subTX

Table 3.4: iSpecPS-DSWP ISA Extensions

Step 4: Apply Memory Versioning

The instructions that memory versioning inserts for each load, store, and external function call are all local to a procedure. However, the current MTX and subTX are stored in microarchitectural registers. To accesses these registers, two additional instructions are addeed to the ISA (Table 3.4).

Step 5: Finish Recovery Code

This step proceeds as in SpecPS-DSWP. With the exception of the misspeculation recovery code, Figure 3.17 shows the final code after the transformation is finished.

```
void main1() {
  list *cur=head, *item;
  int mtx = allocate(0, 0);
  int stx = 0;
  produce(4, mtx);
  produce(5, mtx);
  produce(6, cur);
  produce(7, cur);

  int iter = $r_T$

  while (cur != NULL) {
    int qset = iter % 2;
    int qoff = qset * 18;
    set_qbase(qoff)
    produce(17, TRUE);

    produce_reg_chkpt();
    enter(mtx, stx);
    iter++;

    item = cur->item;
    produce(8, item);

    cur = cur->next;
    produce(9, cur);
    produce(10, cur);

    stx += 1;
    produce(1, OK);
  }
  produce(1, EXIT);
}
```

(a) Thread 1: `main`

```
void main2() {
  list *item, *cur;
  int mtx = consume(4);
  int stx = 0;
  cur = consume(5);

  set_qbase($r_T$ * 18);

  while (cur != NULL) {
    int exit = consume(17);
    if (exit == TRUE) {
      for (int i=0; i<R; i++) {
        set_qbase(i * 18);
        produce(17, FALSE);
      }
      break;
    }

    produce_reg_chkpt();
    enter(mtx, stx);

    item = consume(8);
    work2(item);

    cur = consume(9);

    stx += 1;
    produce(2, OK);
  }
  produce(2, EXIT);
}
```

(b) Thread 2: `main`

```
void main3() {
  list *cur;
  int mtx = consume(5);
  int stx = 0;
  cur = consume(7);

  int iter = $r_T$

  while (cur != NULL) {
    int qset = iter % 2;
    int qoff = qset * 18;
    set_qbase(qoff)

    produce_reg_chkpt();
    enter(mtx, stx);
    iter++;

    work3();

    cur = consume(10);

    stx += 1;
    produce(3, OK);
  }

  produce(3, EXIT);
}
```

(c) Thread 3: `main`

Figure 3.17: iSpecPS-DSWP Parallelized Code without Misspeculation Recovery

```
void work2(list *item) {
  int temp, temp2, temp3;
  int mtx = get_mtx();
  int stx = get_stx();

  produce(11, item);

  int mtxsave = mtx;
  int stxsave = stx;

  mtx = allocate(mtx, stx);
  stx = 0;
  produce(12, mtx);

  while (item != NULL) {
    enter(mtx, stx);

    temp2 = item->data;
    produce(13, temp2);
    if (temp2 < 0) {
      produce(2, MISSPEC);
      wait();
    }

    temp3 = item->data;
    produce(14, temp3);
    temp = -temp3;

    enter(mtx, stx+1);
    item->data = temp;
    item = item->next;
    produce(15, item);
    stx += 2;
    produce(16, MEM_SYNC);
  }

  mtx = mtxsave;
  stx = stxsave;
  enter(mtx, stx);
}
```

(d) Thread 2: `work`

```
void work3() {
  int temp2, temp3;
  int mtx = get_mtx();
  int stx = get_stx();

  list *item = consume(11);

  int mtxsave = mtx;
  int stxsave = stx;

  mtx = consume(12);
  stx = 0;

  while (item != NULL) {
    enter(mtx, stx);

    temp2 = consume(13);
    if (item->data!=temp2) {
      produce(3, MISSPEC);
      wait();
    }

    temp3 = consume(14);
    if (item->data!=temp3) {
      produce(3, MISSPEC);
      wait();
    }

    item = consume(15);
    int unused = consume(16);
    commit_stx(mtx, stx);
    commit_stx(mtx, stx + 1);
    stx += 2;
  }

  commit_mtx(mtx);
  mtx = mtxsave;
  stx = stxsave;
  enter(mtx, stx);
}
```

(e) Thread 3: `work`

Figure 3.17: iSpecPS-DSWP Parallelized Code without Misspeculation Recovery (cont.)

Chapter 4

Extending the Sequential Programming Model

Even with the framework described thus far, some applications will defy automatic parallelization. To extract parallel execution from these applications, this dissertation relies on programmer intervention. However, unlike existing languages and frameworks that force the programmer to begin thinking in terms of parallel execution, this dissertation relies on extensions to the sequential programming model, reducing the burden on the programmer. Before looking at extensions to the sequential programming model in Section 4.2, this chapter first gives an overview of programmer-specified parallelism in Section 4.1.

4.1 Programmer Specified Parallelism

Many parallel programming languages have been proposed over the years. The most popular ones extend existing sequential languages with parallel concepts, with the goal of allowing existing code to be easily parallelized. Most existing languages, including C, have been extended with threads and locks by threading libraries, notably *pthreads*. Unfortunately, by exposing only the low-level concept of threads, these languages give the programmer little if any help avoiding deadlock, livelock, or just plain incorrect execution [47]. Additionally, even if the programmer can extract a correct parallelization, he must then perform a round of performance debugging to achieve a parallelization that is faster than the equivalent single-threaded program. Unfortunately, this performance rarely carries over to a new processor architecture or even a new configuration of an existing processor architecture, necessitating a new round of performance debugging that often leads to another round of correctness debugging.

The OpenMP [2] and MPI [1] extensions to the C programming language attempt to mitigate the performance debugging aspect of manual parallelization. Aimed at processor and cluster level parallelism respectively, the extensions allow the programmer to indicate the type of parallelism desired. However, the programmer is still responsible for finding and appropriately dealing with any cross-thread dependences. Thus, these systems still suffer from the problems of creating a correct parallel program. The use of memory transactions [12] to express parallelism has recently been proposed, but often suffer from correctness [30] and performance [92] issues.

Many new languages have been developed that use higher abstractions to expose parallelism than threads and locks. If an application's algorithm can be expressed in the form of streams, languages such as StreamIt [78] ensure the correctness of the parallelism, while also extracting good performance, but streams are not the natural formulation for many problems. Several languages, such as Cilk [26], attempt to maintain C-like semantics while giving the programmer the ability to express parallelism by making the concepts of closures and continuations first-class features. Just as in language extensions for parallelism, these new languages still force the programmer to understand and effectively use a new programming model.

Beyond a purely programmer-centric approach to parallelism, several techniques in the literature advocate an integrated approach to the extraction of parallelism, combining both manual and automatic parallelization.

SUIF Explorer [43] is a tool for programmer specification of parallelism, which is then assisted by tools support to ensure correctness. The Software Behavior-Oriented Parallelization [21] system allows the programmer to specify intended parallelism. If the intended parallelization is incorrect, the worst case is single-threaded performance at the cost of extra execution resources.

Work proposed by Thies et al. [77] also extracts parallelism in conjunction with the programmer. Through profiling the program's dynamic behavior, their system extracts pipelined parallelism and produces parallelizations for `197.parser` and `256.bzip2` that are effectively the same as those in this dissertation. However, their system relies upon the programmer to mark the potentially parallel regions and often to perform several transformations to make the program amenable for parallelization, but that reduce the software-engineering quality of the code. Additionally, their system does not produce sound code, relying on the profiling phase to see the addresses of *all* potential dynamic memory dependences.

4.2 Annotations for the Sequential Programming Model

Forcing the programmer to think in terms of parallel execution and then annotate the program with this information imposes a large burden on the programmer. It would be better for the programmer to remain in a sequential execution model and for the annotations used to also fit into this sequential model. To this end, this dissertation proposes annotations for a sequential execution model that inform the compiler about non-determinism without forcing the programmer to think in terms of parallelism. By annotating non-determinism, the programmer is indicating that there is no single required order of execution or even a single correct output; rather a multitude of execution orders and outputs are correct, though syntactically or semantically different. This subsection introduces two extensions to a sequential programming model to present this information to the compiler, allowing the extraction of parallelism.

4.2.1 *Y-branch*

```
dict = start_dictionary();
while ((char = read(1)) != EOF) {
  profitable = compress(char, dict)

  @YBRANCH(probability=.000001)
  if (!profitable)
    dict = restart_dictionary(dict);
}
finish_dictionary(dict);
```

(a) *Y-branch*

```
#define CUTOFF 100000
dict = start_dictionary();
int count = 0;
while ((char = read(1)) != EOF) {
  profitable = compress(char, dict)

  if (!profitable) {
    dict = restart_dictionary(dict);
  } else if (count == CUTOFF) {
    dict = restart_dictionary(dict);
    count = 0;
  }
  count++;
}
finish_dictionary(dict);
```

(b) Manual Choice of Parallelism

Figure 4.1: Motivating Example for *Y-branch*

Of those applications with many correct outputs, a large subset of these have outputs where some are more preferable than others. When parallelized by hand, the developer must make a choice that trades off parallel performance for output quality. Instead, this flexibility should be given to the compiler, as it is often better at targeting the unique features of the machine it is compiling for. To this end, this dissertation proposes the use of a Y-branch in the source code. The semantics of the *Y-branch* are that, for all dynamic instances, the *true* path can be taken regardless of the condition of the branch [86]. The compiler is then free to generate code that pursues this path when it is profitable. In particular, this allows the compiler to balance the quality of the output with the parallelism achieved.

Figure 4.1a illustrates a case where the *Y-branch* can be used. The code is a simplified version of a compression algorithm that uses a dictionary. Heuristics are used to restart the dictionary at arbitrary intervals. Rather than insert code to split the input into multiple blocks, as is done in Figure 4.1b, the *Y-branch* communicates to the compiler that it can control when a new dictionary is started, allowing it to choose an appropriate block size. This gives the compiler the ability to break dependences on the dictionary and extract multiple threads. A probability argument informs the compiler of the relative importance of compression to performance. In the case of Figure 4.1a, a probability of $.000001$ was chosen to indicate the dictionary should not be reset until at least 100000 characters have been compressed. Determination of the proper probability is left to a profiling pass or the programmer. Simple metrics, such as the minimum number of characters, are often sufficient.

VELOCITY uses the *Y-branch* annotation [9] to extract more parallelism with iSpecPS-DSWP by allowing the compiler to make the reset function happen at the beginning of a predictable loop iteration. First, a new outer loop around the original loop containing the *Y-branch* is created. The *Y-branch* branch is then transformed to break out of the original loop after it has executed $1/probability$ times. The original loop is first called before entering the new outer loop. An iteration of the new outer loop first calls the code originally controlled by the *Y-branch* and then executes the next iteration of the original loop. Because the reset logic executes at predictable intervals, the compiler can easily parallelize the new outer loop. Figure 4.2 shows how the compiler transforms the *Y-branch* annotation to produce code that can be parallelized. In Figure 4.2a the programmer has indicated that the compiler can choose to taken the branch that controls the reset logic, and that it should not do so more than once every 100,000 iterations. Prior to parallelization, the compiler translates the loop containing the *Y-branch* to form a new outer loop, shown in Figure 4.2b, that can be parallelized automatically.

```
char c;
dict = start_dictionary();
while ((c = read(1)) != EOF) {
  profitable = compress(c, dict)

  @YBRANCH(probability=.000001)
  if (!profitable)
    dict = restart_dictionary(dict);
}
finish_dictionary(dict);
```

(a) Sequential *Y-branch* Code

```
char c;
dict = start_dictionary();
...
while (c != EOF) {
  int numchars = 0;
  dict = restart_dictionary(dict);

  while ((c = read(1)) != EOF) {
    profitable = compress(c, dict)
    if (numchars++ == 100000) break;
    if (!profitable)
      dict = restart_dictionary(dict);
  }
}
```

(b) Transformed *Y-branch* Code

Figure 4.2: *Y-branch* Example

4.2.2 *Commutative*

Many functions have the property that multiple calls to the same function are interchangeable even though they maintain internal state. Figure 4.3a illustrates a code snippet for the `rand` function, which contains an internal dependences recurrence on the `seed` variable. Multiple calls to this function will be forced to execute serially due to this dependence. The *Commutative* annotation informs the compiler that the calls to `rand` can occur in any order.

In general, the *Commutative* annotation allows the developer to leverage the notion of a commutative mathematical operator, which can facilitate parallelism by allowing function calls to execute in any order. This annotation is similar to commutativity analysis [67];

```
int seed;

@Commutative
int rand() {
  seed = seed * 2 + 1;
  return seed;
}
```

(a) Rand Example

```
void * free_list;

@Commutative(memalloc, ROLLBACK, free, ptr)
void * malloc(size_t s) {
  void *ptr = allocate(free_list, s);
  return ptr;
}

@Commutative(memalloc, COMMIT, free, ptr)
void free(void *ptr) {
  size_t size = allocated_size(ptr)
  free_list = deallocate(free_list, ptr);
}
```

(b) Malloc Example

Figure 4.3: Motivating Examples for *Commutative*

both have the goal of facilitating parallelization by reordering operations. However, calls to *Commutative* functions are generally not commutative in the way that commutativity analysis requires. Commutativity analysis looks for sets of operations whose order can be changed, but that result in the same data in the same location. *Commutative* functions can be executed in an order that leads to different values in locations versus the sequential version. The differences that may result are accepted by the programmer. Finally, the programmer annotates *Commutative* based on the definition of a function and not the many call sites it may have, making it easy to apply. *Commutative* is also similar to the Cilk *inlet* directive, as it ensures correct execution of code that executes in a non-deterministic fashion. However, *inlet* is meant to serially update state upon return from a spawned function, while *Commutative* is meant to facilitate parallelism by removing serialization. The annotation is also similar to the *async* and *race* directives used by the IPOT system [84]. In IPOT, the programmer annotates variables as races to indicate that a read can see a potentially stale value of the variable without causing misspeculation. However, a variable can only be marked race when the different data values that result are confined to internal state and are not externally visible. IPOT can also mark code regions as async to allow them to execute in an order that ignores data dependences inside the region, leading to non-determinisitic orderings. Though this annotation can achieve similar effects to *Commutative*, it's purpose is to facilitate parallel execution, forcing the programmer to consider the execution of multi-threads when applying it. Additionally, it is unclear how to use it in a speculative parallelization.

The semantics of the *Commutative* annotation are that, outside of the function, the outputs of the function call are only dependent upon the inputs to it. Additionally, there must exist a well-defined sequential ordering of calls to the *Commutative* function, though that order need not be determined *a priori*.

The *Commutative* annotation can also take an argument which indicates that groups of functions share internal state. When parallelizing, any function in the group must execute atomically with respect to every function in the group. For example, `malloc` and `free` access the `free_list` shown in Figure 4.3b, so both are part of the *memalloc Commutative* set.

The compiler uses the *Commutative* annotation to extract more parallelism by ignoring internal dependence recurrences. Specifically, when building the iPDG, the *Commutative* function is *not* added to the iPDG, just its call sites. Any memory flow dependence between two call sites to the *Commutative* is removed from the iPDG. When generating code, the *Commutative* function itself conceptually executes atomically when called and, inside the function, dependences that are local to the function are respected. By ensuring that the function executes atomically, a well-defined sequential sequence of calls to the *Commutative* function is guaranteed to exist in the absence of misspeculation.

The use of *Commutative* in a speculative execution environment requires additional care. In particular, there must always be a well-defined sequential sequence of calls to the *Commutative* function, particularly in the face of versioned memory and rollback or commit of versioned state. A well-defined ordering is maintained by ensuring that *Commutative* functions executed in non-transactional memory and that the effects of a call to a *Commutative* can be undone by a rollback or are put off until commit, specified by the ROLLBACK and COMMIT values in the *Commutative* annotation.

To handle ROLLBACK, the *Commutative* annotation takes a function as one of its arguments. On each call to the function, the specified arguments and variables of the function are appended to a list, and on rollback the function is called with these arguments. For example, the rollback function for `malloc` is `free` and the `ptr` variable is appended to a list for each call to `malloc`.

For COMMIT, the function is not executed. Instead its arguments are appended to a list. Note that this is only a viable option for functions without return values that do not modify state through pointers passed in as arguments. On commit of versioned memory to non-speculative state, the function is executed for each element of the list. For example, `free` is a COMMIT *Commutative* function. Were it a ROLLBACK *Commutative* function, a new `malloc` function would need to be written that could malloc a specific pointer rather than a size.

Chapter 5

Evaluation of VELOCITY

This chapter presents an evaluation methodology for the automatic parallelization framework presented in Chapter 3 and Chapter 4 as implemented in the VELOCITY compiler. Using several applications from the SPEC CINT2000 benchmark suite as case studies, the chapter will also show how this framework enables the automatic extraction of parallelism.

5.1 Compilation Environment

An automatic parallelization framework based on the techniques discussed in Chapter 3 was built in the VELOCITY [79] compiler. VELOCITY relies upon the IMPACT [69] compiler, a research compiler developed at the University of Illinois at Urbana-Champaign, to read in C code and emit assembly code for a virtualized Itanium® 2 architecture. The IMPACT compiler can perform many advanced analyses and optimizations designed to produce good performing code for EPIC-like architectures [69]. Among these, VELOCITY only uses IMPACT's pointer analysis [14, 53] to identify both call targets of function pointers and the points-to sets of load, stores, and external functions.

VELOCITY contains many traditional optimizations, including global versions of copy propagation, reverse copy propagation, constant propagation, constant folding, dead code elimination, unreachable code elimination, algebraic simplification, redundant load and redundant store elimination, strength reduction, and partial-redundancy elimination, as well as a local version of value numbering. When used, these optimizations are applied exhaustively in a loop until there are no further optimization opportunities. Specialization optimizations, including inlining, loop unrolling, and superblock formation, have also been implemented, with the heuristic for them strongly influenced by IMPACT's heuristics. Finally, VELOCITY contains superblock-based scheduling and global register allocation routines, whose heuristics are also strongly influenced by IMPACT's. VELOCITY outputs assembly code targeting the Itanium® 2 architecture, however it does not currently produce code to take advantage of several of Itanium® 2's more advanced features, including predication, single-threaded software pipelining, single-threaded control and data speculation, and prefetching.

After reading in lowered C code from the IMPACT compiler, all code in the VELOCITY compiler undergoes branch profiling on a training input set, traditional optimizations, inlining, and another round of traditional optimizations. After this, there are two paths that the code follows through the compiler. The standard non-parallelizing optimization pass lowers the code to a machine specific Itanium® 2 IR, performs register allocation and scheduling, and then emits Itanium® 2 assembly. This highly optimized single-threaded code will form the baseline for the experimental results. The standard parallelizing pass first runs multiple profiling passes, including loop-sensitive versions of branch, silent store, alias, and loop-invariant value profiling. After this, the iSpecPS-DSWP automatic parallelization technique is applied. Just as in the single-threaded path, the code is then lowered, register allocated, and scheduled, before emitting as Itanium® 2 assembly.

5.2 Measuring Multi-Threaded Performance

Obtaining the performance of the applications that have been parallelized is difficult. First, many of the loops that will be parallelized encompass more than 95% of the execution time of the program. Since the applications run for many tens of billions, if not hundreds of billions of cycles, and each iteration can take upwards of a billion cycles, the use of traditional simulation methodologies to target many-core processors is impractical at best. A combination of native execution and simulation was used to efficiently measure parallel performance, as shown in Figure 5.1.

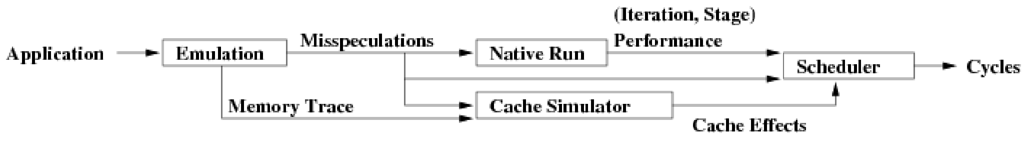


Figure 5.1: Parallel Performance Measurement Dependences

5.2.1 Emulation

First, each parallel program is run in a multi-threaded emulation environment, where the ISA extensions are replaced with function calls into an emulation library. Loads and stores are also replaced since their semantics are dependent upon the memory version system. The resulting binary is then run natively on a multi-processor Itanium® 2 system. This emulation serves several purposes. First, it can easily determine whether the benchmark runs correctly, often in a few hours. Second, it can produce the set of misspeculations that manifested during the application's execution. Lastly, it can produce a memory trace for each thread that will be used for a cache simulation.

5.2.2 Performance

While the emulation run is useful for correctness and other information, its utility as a performance run is hindered by two factors. First, replacing every load and store with a function call greatly reduces any ILP that may exist. Second, emulation must wrap external function calls and emulate the effects in a library. Determining the proper number of cycles for these functions is problematic at best. Thus, another run of the program is performed, however, this time an ordering is enforced on the multi-threaded execution that allows loads and stores to execute correctly without emulation.

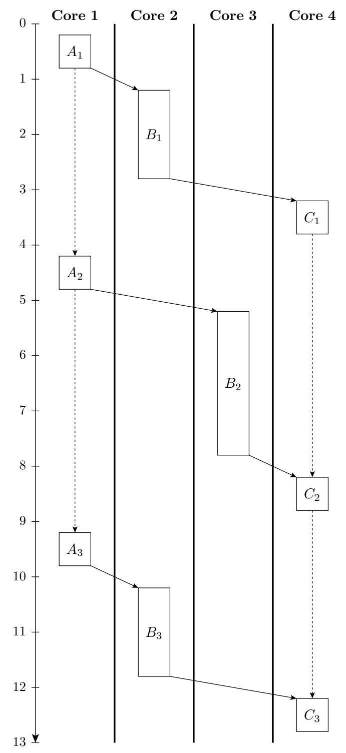


Figure 5.2: Conceptual Performance Run Execution Schedule

Figure 5.2 illustrates this execution order. Essentially, each iteration executes completely before the next iteration is allowed to run. That is, only when the last stage of an iteration completes is the first stage of the next iteration allowed to execute. So long as there is no intra-iteration memory value speculation, this execution plan is a legal multi-threaded execution that respects all memory flow dependences and false memory dependences *without* the use of versioned memory. Thus, loads and stores are not replaced and memory version instructions become nops. However, for correct execution, *produce*, *consume*, *queue.set*, and *queue.flush* are replaced with calls into the emulation library.

The multi-threaded application was run on an unloaded HP workstation zx2000 with a 900MHz Intel Itanium 2 processor and 2Gb of memory, running CentOS 4.5. Runtime numbers were gathered using HP's *pfmon* tool version 3.0, which is based on version 3.1 of the *libpfm* performance monitoring library [24]. The runtime of an application was determined by running it under the *pfmon* tool with the CPU_CYCLES, $BE\_EXE\_BUBBLE\_GRALL$, and $BE\_EXE\_BUBBLE\_GRGR$ events monitored. CPU_CYCLES represents the total number of cycles needed to execute the application. $BE\_EXE\_BUBBLE\_GRALL - BE\_EXE\_BUBBLE\_GRGR$ represents the number cycles spent stalling on an outstanding integer load. This number will be subtracted from CPU_CYCLES and a cache simulation's performance measurement added back in.

To allow for an accurate performance measurement, performance counters are turned off before entering into an emulated instruction and turned back on just after returning. Itanium® 2 contains single instruction support for turning performance counters on (*sum*) and off (*rum*). While these instructions introduce bubbles into the pipeline, the cycles lost because of this can be accurately measured via the $BE\_EXE\_BUBBLE\_ARCR$ event and subtracted from the CPU_CYCLES measurement. Additionally, to avoid profiting from the use of separate copies of each cache, branch predictor, and other microarchitectural structures per core, the threads are constrained to execute on a single processor via the `sched_setaffinity` system call.

Because loads and stores are not replaced and memory versioning is not done, memory state can *not* be undone in the event of misspeculation. To avoid the need to rollback state, the performance run takes in the misspeculations that occurred during the emulation run and proactively signals misspeculation and branches to recovery code for any iteration that misspeculates.

While this execution completely avoids parallel execution, it produces an execution time for each $(iteration, stage)$ pair which can be used to produce a multi-threaded performance number by a scheduler.

Private L1 Cache	16KB, 4-way, 64B line size, writeback, 1/1/1-cycle access/commit/merge latencies
Private L2 Cache	256KB, 8-way, 128B line size, writeback, 4/2/1-cycle access/commit/merge latencies
Shared L3 Cache	768KB, 12-way, 128B line size, writeback, 9/7/1-cycle access/commit/merge latencies
Shared L4 Cache	64MB, 256-way, 128B line size, writeback, 50/25/1-cycle access/commit/merge latencies
Main Memory	100 cycle access latency

Table 5.1: Cache Simulator Details

5.2.3 Cache Simulation

Since the stage performance run executed on a single processor, it did *not* benefit from effectively larger branch predictors or caches, however, it also did not pay any penalty due to cache coherence effects, particularly false sharing, or the use of versioned memory. To model the cache effects, the memory trace from the emulation run is feed into a multi-core cache simulator, the details of which are given in Table 5.1. The cache simulation has a 4-level cache, with private L1 and L2 caches, and shared L3 and L4 caches. The details of the version memory cache system can be found [82]. The output of the cache simulator is a stall cycle count that each $(iteration, stage)$ pair spent in the cache system. The simulated stall cycle count is added to the $(iteration, stage)$ cycle count from Section 5.2.2 and the measured load stall cycle count ($BE\_EXE\_BUBBLE\_GRALL - BE\_EXE\_BUBBLE\_GRGR$) is subtracted from it.

5.2.4 Bringing it All Together

To determine the performance of the application given the $(iteration, stage)$ performance counts, the set of misspeculating iterations, and the stage dependences of the iSpecPS-DSWP pipeline, a scheduler is used to simulate the performance of multi-threaded execution. The scheduler simulates an 32-core Itanium® 2 machine, but does not directly model the microarchitectural effects of branch predictors or other microarchitectural state. Note that cache effects are represented by the stall times incorporated into the $(iteration, stage)$ performance count from the cache simulator. The scheduler executes the multi-threaded program assuming infinite length queues and respecting the dependences among pipeline stages and misspeculating iterations. At the end, it outputs the cycle count of the last stage of the last iteration. This is the performance number of the multi-threaded code.

5.3 Measuring Single-Threaded Performance

To evaluate the performance of the single-threaded baseline, the applications were run on the same machine as the multi-threaded application. The integer load stalls were also subtracted just as in the multi-threaded applications, yielding a performance number ignoring the effects of data caches. To get the number of cycles spent stalling on the cache, the binary is instrumented to get a memory trace which is fed into the same cache simulator used to measure the multi-threaded binary. The resulting penalty is added to CPU_CYCLES to get the final single-threaded performance number.

5.4 Automatically Extracting Parallelism

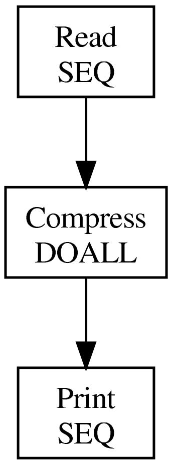
The automatic parallelization technique described in Chapter 3 can extract scalable parallelism for several applications. This section shows the results of applying VELOCITY to several benchmarks in the SPEC CINT2000 suite.

5.4.1 256.bzip2

To see how the many components of the framework extract parallelism, consider the `256.bzip2` application from the SPEC CINT2000 benchmark suite. `256.bzip2` compresses or decompresses a file using the Burrows-Wheeler transform and Huffman encoding. This dissertation focuses only on the compression portion of the benchmark. Each iteration of the `compressStream` function, shown in Figure 5.3a, compresses an independent block of data. The data from the file is run through a Run-Length Encoding filter and stored into the global `block` data structure. Because of this, the `block` array contains a varying number of bytes per data block, indicated by `last`. Assuming

```
compressStream(inStream, outStream) {
  while (True) {
    blockNo++;
    initialiseCRC();
    loadAndRLEsource(inStream);
    if (last == -1) break;

    doReversibleTransformation()
    moveToFrontCodeAndSend();
  }
}
```

(a) `compressStream` pseudo-code



(b) `compressStream` DAG_{SCC}

Figure 5.3: Simplified version of `compressStream` from `256.bzip2`

data bytes exist to compress, they are compressed by the `doReversibleTransform` and `generateMTFValues` functions before being appended to the output stream by `sendMTFValues`.

Versions of the bzip2 algorithm that compress independent blocks in parallel have been implemented in parallel-programming paradigms [3]. The parallelization performed manually with mutexes and locks decomposes the original loop in `compressStream` into three separate loops, shown in Figure 5.3b. The first stage reads in blocks to compress, followed by a stage to compress, finishing with a stage to print the compressed blocks in order. Most of the parallelism extracted comes from executing multiple iterations of the second stage in parallel.

The same pipelined parallelism extracted manually can also be extracted using the DSWP [55, 63] technique. Unfortunately, the base technique cannot extract the DOALL parallelism present in the manual parallelization, as it only partitions the static body of a loop. In the case of `256.bzip2`, DSWP could extract a 4 stage pipeline, placing `doReversibleTransform` and `generateMTFValues` in their own pipeline stages. However, this limits the speedup achievable to $\sim$2x in practice, as

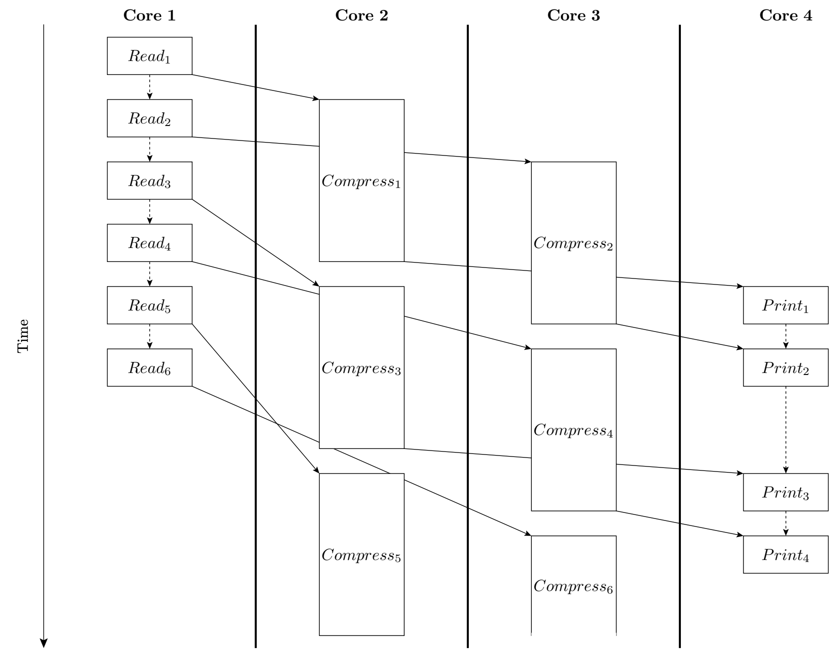


Figure 5.4: Dynamic Execution Pattern

`doReversibleTransform` takes ∼50% of the loop's runtime. Instead, since no loop-carried dependences exist in the *Compress* stage, it can be replicated several times using the PS-DSWP extension, allowing multiple iterations to execute in parallel [61].

Unfortunately, while there are no actual loop-carried dependences in the *Compress* stage, the compiler is unable to prove this. In particular, the compiler cannot prove that the accesses to `block` are limited by `last`, and must conservatively assume that writes from a previous iteration can feed forward around the back edge to the next iteration. Adding speculation to PS-DSWP allows the compiler to break these dependences [82]. To avoid excessive misspeculation, the framework uses a memory profiler to find memory dependences that rarely, or, in the case of `block`, never manifest themselves. Since breaking loop-carried dependences is the key to PS-DSWP, the memory profiler must profile memory dependences relative to loops, allowing it to determine if a dependence manifests itself intra-iteration or inter-iteration.

Memory locations reused across iterations, such as the `block` array, also present an-

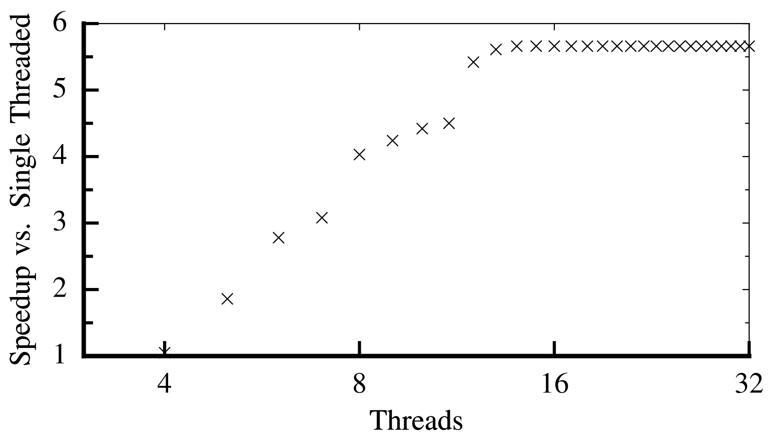


Figure 5.5: Speedup of multi-threaded (MT) execution over single-threaded (ST) execution for `256.bzip2`

other problem because the same memory location is read and written every iteration, leading to many loop-carried false memory dependences. To break these dependences, each iteration is placed in a separate ordered version memory transaction.

Lastly, the parallelization technique must be able to parallelize not just a loop, but the functions called, directly or indirectly, from the loop. In `256.bzip2`, the read and print phases are the `bsR` (bitstream Read) and `bsW` (bitstream Write) functions, respectively. These functions are found deep in the call tree, making it hard to expose the code inside for parallelization. Inlining until all calls to these functions are visible at the outermost loop level is not a practical solution, as the loop quickly becomes too large to analyze. Thus, iSpecPS-DSWP is needed to extract the expected parallelization.

The framework extracts a parallelization that executes according to the dynamic execution plan shown in Figure 5.4. As the number of *Compress* stage threads is increased, performance increases as shown in Figure 5.5. Though the parallelization relies on speculation to removes dependences, no misspeculation occurs, as no flow memory dependence is actually loop-carried in the *Compress* stage. Thus, the speedup is limited by the input file's size and the level of compression used. Since the file size is only a few megabtyes and the compression level high, only a few independent blocks exist to compress in parallel and the speedup levels off after 14 threads.

5.4.2 175.vpr

```
try_swap() {
  blk_from = rand();
  ax = blk_from.x;
  ay = blk_from.y;
  do {
    bx = rand(), by = rand();
  } while ((ax != bx) && (ay != by));
  blk_to = find_block(bx, by);

  swap(blk_from, blk_to);

  affected_nets = find_affected_nets(blk_from, blk_to);
  if (assess_swap(affected_nets, blk_from, blk_to) {
    update_nets(affected_nets);
    reset_nets(affected_nets);
  } else {
    unswap(blk_from, blk_to);
    reset_nets(affected_nets);
  }
}

try_place() {
  while (cond) {
    for (iter=0; iter < limit; iter++) {
      if (try_swap()) success++;
    }
  }
}
```

Figure 5.6: Simplified version of `try_place` from `175.vpr`

`175.vpr` is an application that performs FPGA place and route calculations. As in previous work [59], this dissertation focuses on the placement portion of the algorithm, which is distinct from the routing portion. Placement consists of repeated calls to `try_swap` in the `try_place` function. `try_swap`, as its name implies, attempts to switch a block to a random position, also switching the block at that position if one is already there. A pseudo-random number generator is used to choose a block and an (x, y) position to move the block to. If either coordinate is the same as the chosen block's `x` or `y` value, then a new random number is generated until they are distinct. The block's coordinates are then updated and the cost of updating the connecting networks is calculated. If the cost is below a certain threshold, the swap is kept, otherwise, the block's coordinates are reverted back

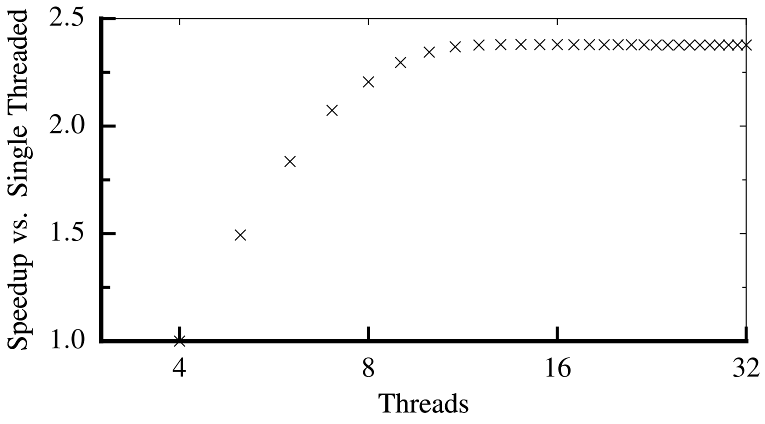


Figure 5.7: Speedup of multi-threaded (MT) execution over single-threaded (ST) execution for `175.vpr`

to their previous values.

The calls to `try_swap` can often be executed in parallel. Predicting dependences among these iterations *a priori* is hard, to the point that an alias speculation rate of 90% and control speculation rate of 30% are required to produce a good partition. However, in many cases the value a the beginning of an iteration will be the same as it was in a previous iteration. If a loop-carried invariant speculation rate of 10% is used, then a better partitioning can be achieved that has a much lower rate of misspeculation. The iteration invariant misspeculation occurs when updates to the block coordinates and network structures are not properly read by subsequent iterations, which happens, on average, every 10 iterations.

The overall performance of `175.vpr` on up to 32-threads is shown in Figure 5.7. Interestingly, misspeculation is highest in early iterations of the `try_place`, with an average of 6 iterations between misspeculations, loop where the swap is accepted more often. It reduces to almost zero in later iterations, with an average of 30 iterations between misspeculations, where few swaps are accepted. Thus, good parallel performance requires many threads in later iterations, where the speculation succeeds more than 95% of the time, to balance out the more serial performance of earlier iterations.

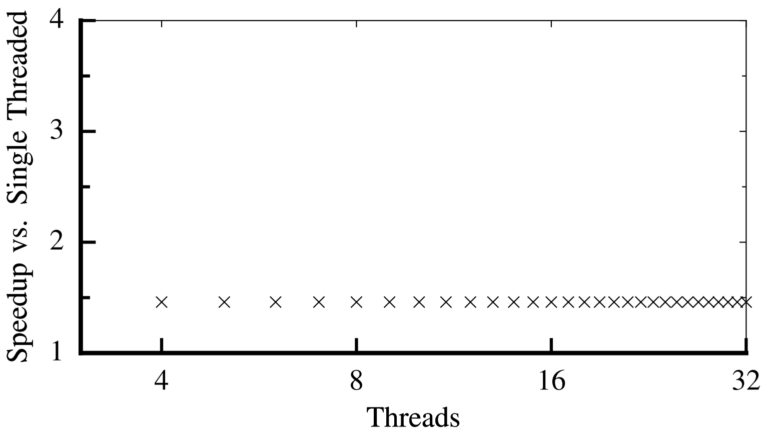
5.4.3 181.mcf

181.mcf is an application that solves the single-depot vehicle scheduling in public mass transportation, essentially a combinatorial optimization problem solved using a network simplex algorithm. The high-level loop occurs in global_opt which calls both the price_out_impl and primal_net_simplex functions.

Parallelizing the global_opt loop in 181.mcf requires a large amount of speculation that limits any speedup obtainable. However, the outer loops in the price_out_impl and primal_net_simplex functions are parallelizable.

```
void primal_net_simplex(net) {
  while (!opt) {
    if (bea = primal_bea_mpp(m, arcs, stop_arcs, ...)) {
      ...
      if (iterations % 20) { refresh_potential(); }
    }
  }
  refresh_potential(net);
}

long price_out_impl() {
  for (; i < trips; i++, arcout += 3) {
    head = arcout->head;
    arcin = first_of_sparse_list->tail->mark;
    while (arcin) {
      tail = arcin->tail;
      red_cost = compute_red_cost(arc_cost, tail);
      update_arcs(tail, head, new_arcs, red_cost)
      arcin = tail->mark;
    }
  }
}

void global_opt() {
  while (new_arcs) {
    primal_net_simplex(net);
    new_arcs = price_out_impl(net);
  }
}
```

Figure 5.8: Simplified version of global_opt from 181.mcf

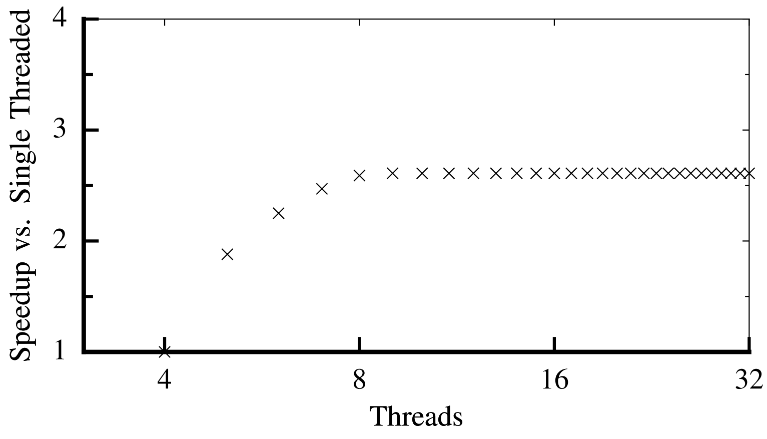


(a) `primal_net_simplex` Speedup

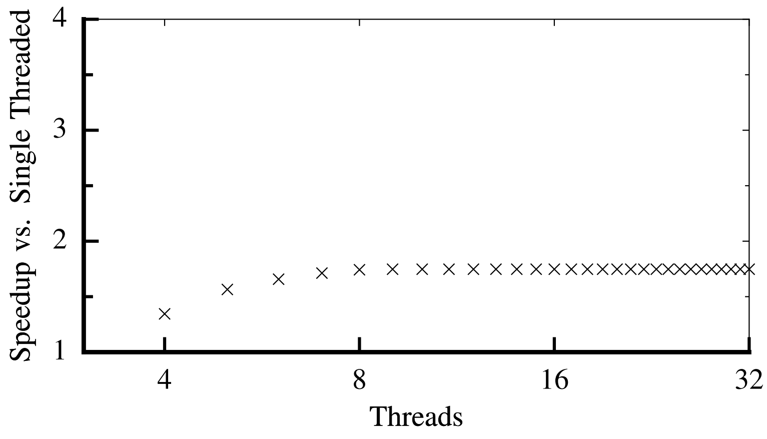
Figure 5.9: Speedup of multi-threaded (MT) execution over single-threaded (ST) execution for `181.mcf`

`primal_net_simplex` takes approximately 70% of the execution time and can be parallelized by executing `refresh_potential` in parallel with the rest of `primal_net_simplex`. This is primarily accomplished by applying silent store speculation to the `node->potential` updates in `refresh_potential`. Essentially, the parallelization speculates that refresh potential will not change the actual potential of any node in the tree, which is almost always the case. Speedup is limited not by misspeculation, but by the largest stage, which contains approximately 40% of the runtime.

The other 30% of the runtime occurs in `price_out_impl`. The outer loop in `price_out_impl` can be parallelized into three stages. The first stage is very small, 1% of the runtime, and SEQUENTIAL. It contains a few branches that control `continues` in the loop body and the `arcout->head->firstout->head->mark` update. The second stage is DOALL and contains the majority of the runtime, 75%, and the majority of the code. The third stage is SEQUENTIAL and contains the code from `insert_new_arc` and `replace_weaker_arc` that inserts new arcs into a priority queue backed by a heap. Misspeculation is not the limiting factor on speedup, as the parallelization utilizes speculation to only break potential alias dependences that never manifest.



(b) price_out_impl Speedup



(c) Benchmark Speedup

Figure 5.9: Speedup of multi-threaded (MT) execution over single-threaded (ST) execution for 181.mcf (cont.)

Figure 5.9 shows the speedup achieved when parallelizing the price_out_impl and primal_net_simplex loops. Since the price_out_impl parallelization contains a DOALL node, it is able to achieve some scalable parallelism and its speedup tops out at 2.61x. The primal_net_simplex parallelization contains no DOALL nodes, so its speedup is always 1.5x for 4 or more threads.

5.5 Extracting Parallelism with Programmer Support

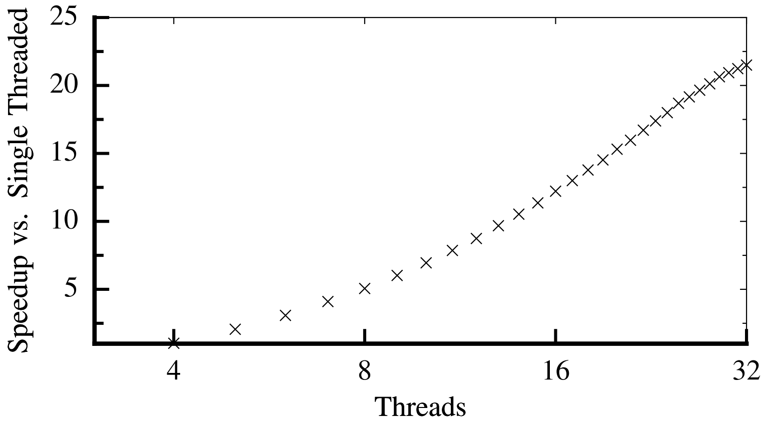
Combining the parallelization technique of Chapter 3 with the programmer annotations described in Chapter 4, VELOCITY can extract parallelism from several more benchmarks in the SPEC CINT2000 suite.

5.5.1 164.gzip

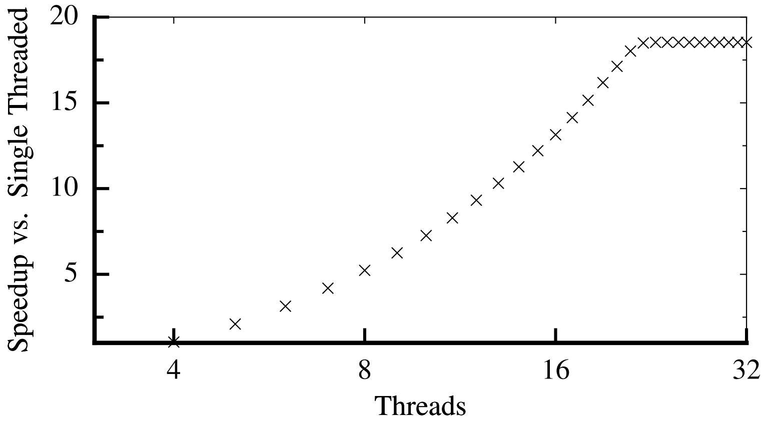
```
deflate() {
  while (lookahead != 0) {
    match = find_longest_match();
    if (match)
      reset_needed = tally_match(match);

    @YBRANCH
    if (reset_needed)
      reset();

    while (lookahead < MIN_LOOKAHEAD)
      fill_window();
  }
}

void crc(short *s, int n) {
  while (--n) {
    int indx = (crc ^ (*s++)) & 0xff;
    crc = crc_32_tab[indx] ^ (crc >> 8);
  }
}
```

Figure 5.10: Simplified version of `deflate` from `164.gzip`.

`164.gzip` is another compression and decompression application, which uses the Lempel-Ziv 1977 algorithm. As with `256.bzip2`, this dissertation focuses on the compression portion of the benchmark. Each file is compressed by either the `deflate` or `deflate_fast` function, which have almost the same code. The implementation of the algorithm uses a sliding window of 65536 characters to look for repeated substrings. The compression itself is delineated by blocks, ended by calls to `FLUSH_BLOCK`. Unlike `256.bzip2`, the choice of when to end compression of the current block and begin a new block is made based on various factors related to the compression achieved on the current block. This dependence makes it impossible to compress blocks in parallel as it very hard to predict the point at which a new block will begin.



(a) `deflate_fast` Speedup

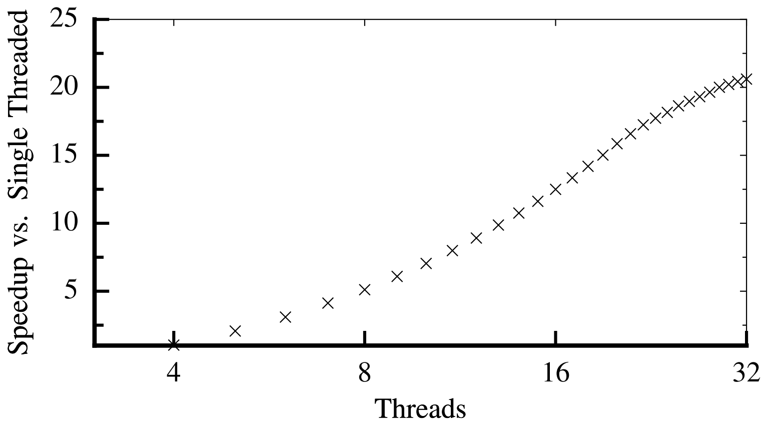


(b) `deflate` Speedup

Figure 5.11: Speedup of multi-threaded (MT) execution over single-threaded (ST) execution for `164.gzip`

Manually parallelized versions of the gzip algorithm insert code to ensure that a new block is started at fixed intervals. To allow the benchmark to be parallelized, a similar change was made to the source code to always start a new block at a fixed interval. Upon starting a new block, the sliding window is reset, avoiding cross-block matches. This code is represented by the call to `reset` highlighted in Figure 5.10.

Unfortunately, this change can cause the application to achieve a lower compression ratio than the single-threaded version. For the 32Mb input file, using the smallest block size of 64Kb, the loss in compressions was less than .5% at the highest compression level. The loss in compression decreased to less than .1% when 1Mb blocks were used. When there are multiple processors, this tradeoff between compression and performance is acceptable. However, this loss of compression should only occur if parallelization was achieved. In particular, if only a single processor is available, the same compression level should be achieved as the basic benchmark.



(c) Benchmark Speedup

Figure 5.11: Speedup of multi-threaded (MT) execution over single-threaded (ST) execution for `164.gzip` (cont.)

Figure 5.11 shows the speedup achieved on the 32Mb input file for a block size of 132Kb. Little speculation is needed, mostly alias speculation on various arrays as in `256.bzip2`, and no misspeculation occurs in practice. Additionally, since the block size is smaller than that in `256.bzip2` and the file size larger, there is is sufficient work to keep more threads busy. The limiting factor for the speedup is the calculation of the Cycle Redundancy Check (`crc`). The `crc` computation is isolated to its own thread, but takes approximately 3-4% of the runtime for both the `deflate` and `deflate_fast` loops. The code is amenable to single-threaded optimizations that would reduce its execution time, including software pipelining and load combining, that have not been implemented in VELOCITY.

5.5.2 197.parser

```
while(1) {
  sentence = read();
  if (!sentence) break;
  print(sentence);
  err = parse(sentence);
  errors += err;
  if (err < 0)
    exit();
}
print errors;
```

Figure 5.12: Simplified version of `batch_process` from `197.parser`

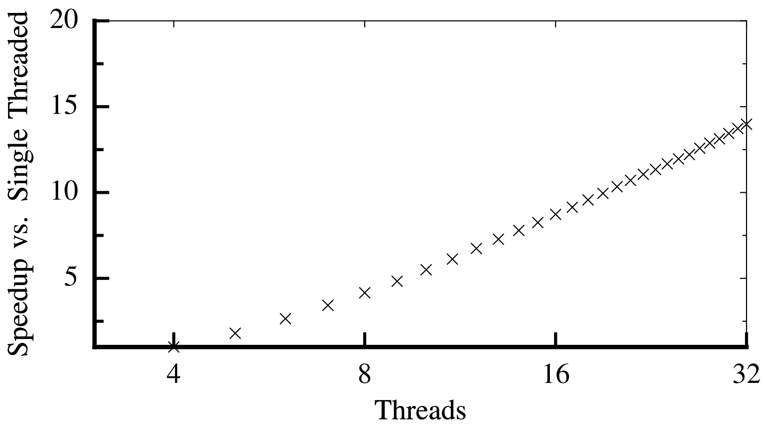


Figure 5.13: Speedup of multi-threaded (MT) execution over single-threaded (ST) execution for `197.parser`

`197.parser` is an application that parses a series of sentences, analyzing them to see if they are grammatically correct. The loop in the `batch_process` function comprises the outermost loop, of which the `parse` function call dominates the runtime of each iteration. As each sentence is grammatically independent of every other sentence, parsing can occur in parallel for each sentence.

Previous work [21] has used speculation to break dependences between iterations. A sentence may be a command for the parser rather than a sentence to parse, turning on or off echo mode, for example. However, speculation is not required for this application if these operations are placed into a SEQUENTIAL stage. The loss in parallelization is limited, as a majority of the time is taken up by the `parse` call.

To achieve a parallelization that parses each sentence in parallel, the memory allocator must also be dealt with. Upon startup, `197.parser` allocates 60MB of memory, which it then manages internally. To avoid dependences from the memory allocator interfering with parallelization, the *Commutative* annotation is used once again. This allows the compiler to break dependences across iterations arising from calls to the allocator.

The standard parallelization paradigm of SEQUENTIAL, DOALL, SEQUENTIAL, is used as the parallelization. Its speedup is shown in Figure 5.13. The limiting factor on speedup in the number of sentences to parse, and the high variance, up to 10x, that occurs when parsing a particular sentence.

5.5.3 186.crafty

```
Search(alpha, beta, depth, white) {
  initial_alpha = alpha;
  while (move = NextMove(depth, white)) {
    MakeMove(move, depth, white);
    value = -Search(-alpha-1, -alpha, !white, depth+1);
    if (value > alpha) {
      if (value >= beta) {
        UnMakeMove(depth, white);
        return value;
      }
      alpha = value;
    }
    UnMakeMove(move, depth, white);
  }
  return alpha;
}
```

Figure 5.14: Simplified version of `Search` from `186.crafty`

`186.crafty` is an application that plays chess. The high-level loop reads in a chess board and a search depth n. For each iteration of this loop the `Iterate` function is called, which executes repeated searches from a depth of 1 to n. To perform this, `Iterate` calls the `SearchRoot` function which calls the recursive `Search` function to perform an alpha-beta search. For each level of the search, several moves are computed, each of which is recursively searched and evaluated to determine the most profitable path. The

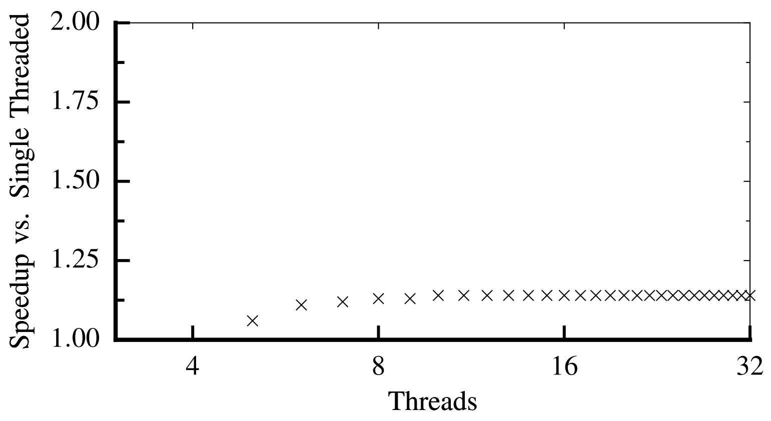


Figure 5.15: Speedup of multi-threaded (MT) execution over single-threaded (ST) execution for `186.crafty`

most profitable move is then stored and the alpha value is returned.

The most obvious parallelization is to search each of the moves at one level of the `Search` independently, similar to they way the application has been parallelized by hand. Unfortunately, `Search` is recursive, a problem that has hindered previous work in parallelization [59]. To solve this problem, the compiler conceptually peels of an iteration the recursion. To allow parallelization, one level of the recursion is peeled off.

This parallelization requires that loads of the `search` variable, which contains many fields related to the current search, be speculated as loop-carried invariant. This is always true, but is very hard for the compiler to predict as it requires understanding that the `UnMakeMove` undoes the effects of `MakeMove`. Additionally, control speculation is needed to prevent the presence of certain cutoff metrics related to timing and number of nodes searched from preventing parallelization. In particular the `next_time_check` variable branch in `Search` must be speculated not taken.

An additional problem also arises in the parallelization. Since the search reaches many of the same boards, a large amount of caching is used to prune the search space and improve performance. Unfortunately, these caches prevent parallelism, as the dependence from the store into the cache to a load from the cache is hard to predict. Alias speculation can allow the parallelization to continue, but misspeculation still limits performance. Instead, this dissertation relies on the programmer to mark each cache lookup and insertion function as *Commutative*. This includes references to the `pawn_hash_table`, `trans_ref`, `history`, and `killer_move` caches.

Finally, reads of and updates to the `alpha` variable and `largest_positional_score` are also marked as *Commutative*. Both of these variables are monotonically increasing and marking them as *Commutative* only means that the search may explore a larger space than it would have otherwise. In practice, the number of extra moves explored is negligible as both variables are not updated often.

Unfortunately, even though the parallelization framework extracts a SEQUENTIAL, DOALL, SEQUENTIAL parallelization where more than 99% of the runtime occurs in the DOALL stage, the speedup is only 1.15x. This occurs not because of misspeculation, which occurs only on loop exit, but because of the amount of time it takes to search a particular move is highly variable due to the aggressive pruning performed during the search. In particular, in most invocations of `Search`, a single iteration contains more than 90% of the runtime.

5.5.4 300.twolf

```
while (attempts < attmax) {
  a = rand();
  do {
    b = rand();
  } while (a != b);

  if (cond) {
    ucxx2();
  } else ...
}
```

Figure 5.16: Simplified version of `uloop` from `300.twolf`

`300.twolf` is an application that performs place and route simulation. The innermost loop that comprises most of the execution time is in the `uloop` function, which contains many calls to `ucxx2`. The `ucxx2` function takes up approximately 75% of the execution

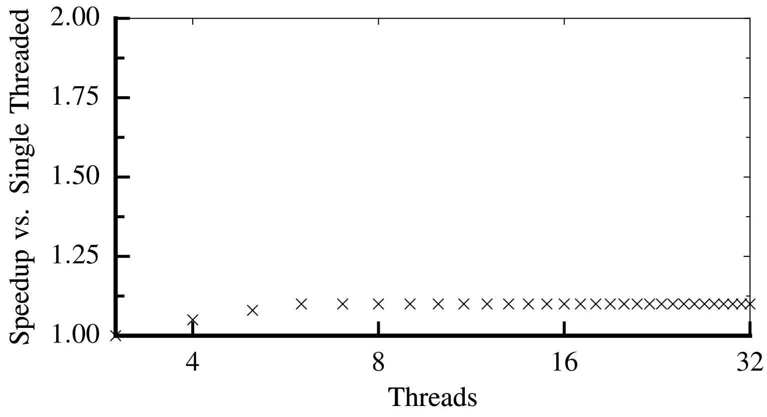


Figure 5.17: Speedup of multi-threaded (MT) execution over single-threaded (ST) execution for `300.twolf`

time. Previous work has executed portions of the `ucxx2` code in parallel via speculative pipelining [59]. This paper instead parallelizes the calls to `ucxx2` themselves by parallelizing iterations of the loop in `uloop`.

Predicting which iterations can execute in parallel *a priori* is hard, so branch and alias speculation are used to achieve the parallelization. However, the misspeculation rate greatly limits the amount of parallelism extracted [59]. This misspeculation comes from two sources, misprediction of the number of calls to the pseudo-random number generator and memory dependence violations on the block and network structures.

The misspeculation for the random-number generator occurs because of the variable number of calls to the random number generator. In previous work [59], this dependence has been broken by manually speculating the number of calls to the generator and predicting the next iteration's seed. However, it is counterintuitive for parallelism to be limited by the generation of random numbers. To avoid the misspeculation and allow the compiler to see the parallelism, that the random number generator is marked as *Commutative* by the programmer. For the pseudo-random number generator, this allows the calls to the generator to occur in any order and breaks the dependence that the generator has across iterations on the `randVarS` variable. Though output changes as a result of this, the benchmark still runs as intended.

With these changes, the parallelization is able to achieve a speedup of 1.10x, as shown in Figure 5.17. Misspeculation occurs quite often, approximately every other iteration. Because the pipeline is restarted on every misspeculation, pipeline fill time is on the critical path and lowers the speedup attainable.

5.6 Continuing Performance Trends

This chapter has shown that VELOCITY can extract parallelism from applications in the SPEC CINT2000 benchmark suite, even though these programs are generally considered not to be amenable to automatic parallelization because of the number and complexity of the dependences involved. This section summarizes the results of the dissertation and compares them to aggressive manual parallelization.

Benchmark	Loop	Exec. Time	Lines Changed	Lines Annotated	Description
164.gzip (Compress)	`deflate_fast` (deflate.c:583-655)	30%	26	2	Gzip Compression
	`deflate` (deflate.c:664-762)	70%			
175.vpr (Place)	`try_place` (place.c:506-513)	100%	0	0	Processor Node Placement
181.mcf	`price_out_impl` (implicit.c:228-273)	25%	0	0	Single-Source, Multi-Depot Optimization Problem
	`primal_net_simplex` (psimplex.c:50-138)	75%			
186.crafty	`Search` (search.c:218-368)	98%	9	9	Chess Playing Program
197.parser	`batch_process` (main.c:1522-1779)	100%	2	2	English Grammar Checker
256.bzip2 (Compress)	`compressStream` (bzip2.c:2870-2919)	100%	0	0	Bzip2 Compression
300.twolf	`uloop` (uloop.c:154-361)	100%	1	1	Processor Layout

Table 5.2: Information about the loop(s) parallelized, the execution time of the loop, and a description.

Table 5.2 gives a summary of the applications from the SPEC CINT2000 suite that were parallelized by VELOCITY, including the loop, its percent of execution, and a general description of the application. For these benchmarks, as well as the remaining C benchmarks

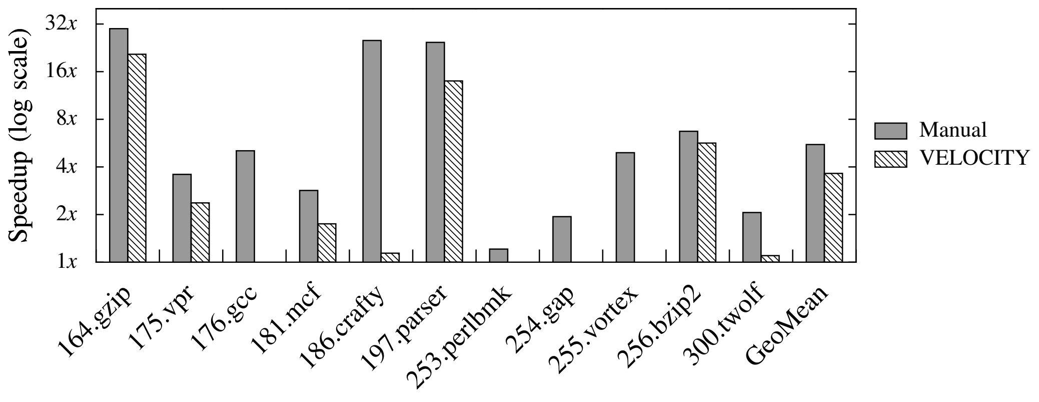


Figure 5.18: The maximum speedup achieved on up to 32 threads over single threaded execution. The Manual bar represents the speedup achieved using manual parallelization [9], while the VELOCITY bar represents the speedup achieved by VELOCITY.

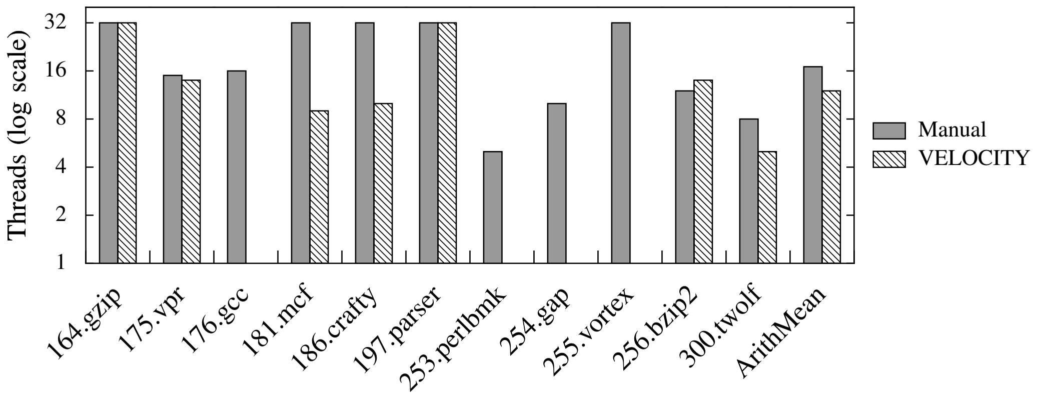


Figure 5.19: The minimum number of threads at which the maximum speedup occurred. The Manual bar represents the number of threads used by manual parallelization [9], while the VELOCITY bar represents the number of threads used by VELOCITY.

in SPEC CINT2000, Figure 5.18 shows the maximum speedup achieved for each benchmark, while Figure 5.19 shows the minimum number of threads at which the maximum speedup was achieved. The *Manual* bar represents the speedup achieved using via manual application of the parallelization techniques in this dissertation, [9], while the VELOCITY bar represents the speedup that was achievable automatically by the VELOCITY compiler. For the benchmarks that were automatically parallelized, the parallelization obtained match those that were simulated manually [9]. There are several reasons for the performance differences:

1. The manual parallelizations used the gcc compiler which contains different optimizations and heuristics than the VELOCITY compiler.

2. The manual parallelization performance evaluation did not account for the effects of multiple caches or cache coherence among the cores.

3. The manual parallelization performance evaluation was achieved by manually marking portions of the single-threaded binary that could execute in parallel and then feeding these regions into a parallel execution simulator similar to that described in Section 5.2. The VELOCITY performance evaluation uses real thread partitionings, chosen automatically, that generate actual multi-threaded code and which contain many overheads, including synchronization instructions and extra branches to ensure proper control flow.

As in the manual parallelization work, `252.eon` was not parallelized because it is a C++ benchmark. `176.gcc`, `253.perlbmk`, `254.gap`, and `255.vortex` were not parallelized by VELOCITY because of the difficulties, particularly with regards to memory usage, in building the iPDG. The smallest of these benchmarks, `255.vortex`, would require the creation of a iPDG with a hundred thousand nodes and more than 25 million edges. The vast majority of these edges are memory dependence edges, with less than a million control or register edges. The iPDG can be made more scalable by selectively expanding function calls, rather than all functions as the current iSpecPS-DSWP does. The downside of not expanding all functions is that the summarization of call sites potentially creates larger SCCs and forces the partitioner to allocate the entire function to a single thread rather than among multiple threads. Techniques that summarize operations at a granularity chosen by the compiler, particularly dependence chains, would also reduce the number of edges in the iPDG, while the retaining information needed for parallelization.

Finally, even though the `186.crafty` parallelization for the `Search` function is the same, the performance extracted is much lower because VELOCITY does not currently support parallelizing loops inside loops that are already parallelized. Thus, only one loop could be parallelized, leading to execution time that is heavily biased to only a few iterations in the loop. This limitation also results in reduced performance potential in `181.mcf`.

Even with these difficulties, targeting a 32 core system, a speedup of 3.64x was achieved by changing only 38 out of more than 100,000 lines of code. As the manual parallelizations show, the speedups obtained are not an upper bound. Indeed, there are many inner loops that have not yet been parallelized.

Benchmark	# Threads	Speedup	Historic Speedup	Ratio
164.gzip	32	20.61	5.38	3.83
175.vpr	14	2.37	2.43	0.98
181.mcf	9	1.74	2.09	0.83
186.crafty	10	1.14	2.17	0.52
197.parser	32	13.98	5.38	2.60
256.bzip2	14	5.66	2.43	2.82
300.twolf	5	1.10	1.72	0.64
GeoMean		3.24	2.80	1.16
ArithMean	16.58		3.09	

Table 5.3: The minimum # of threads at which the maximum speedup occurred. Assuming a 1.4x speedup per doubling of cores, the *Historic Speedup* column gives the speedup needed to maintain existing performance trends. The final column gives the ratio of actual speedup to expected speedup.

Table 5.3 presents the actual VELOCITY numbers for Figure 5.18 and 5.19. Additionally, the *Historic Speedup* is given, which details the expected performance increase for the number of transistors used. While there are no statistics that directly relate the doubling of cores to performance improvement, historically, the number of transistors on a chip has doubled approximately every 18 months, while performance has doubled approximately every 36 months. Assuming that all new transistors are used to place new cores on a chip, each doubling of cores must yield approximately 1.4x speedup to maintain existing performance trends. Thus, the *Historic Speedup* column represents the expected speedup for the number of threads, calculated as $1.4^{log_2(\#Threads)}$. The final column gives the ratio of the actual performance improvement to that required to maintain the 1.4x speedup. The overall performance improvement indicates that sufficient parallelism can be extracted to use the resources of current and future many-core processors.

Chapter 6

Conclusions and Future Directions

Multi-core processors have already severely challenged computer architecture. Unless compilers begin automatically extracting parallelism, multi-core processors also promise to significantly increase the burden on programmers. Relying on the programmer to correctly build large parallel programs is not a tenable solution as programming is already a complex process and making it harder is not a way forward. Additionally, this solution does not address the large body of legacy sequential codes. This dissertation shows that automatic extraction of parallelism is possible for many of the benchmarks in the SPEC CINT2000 benchmark suite and is a viable methodology to continuing existing performance trends for legacy sequential codes.

This dissertation has proposed and built a new parallelization framework, designed to extract parallelism from general-purpose programs. It has combined existing Piplined-Multi Threading techniques, including DSWP, PS-DSWP, and SpecDSWP along with a novel interprocedural scope extension to form iSpecPS-DSWP. This technique, along with existing analyisis and optimization techniques, can be used to extract large amounts of parallelism. For those applications that are not parallelized by this framework, this dissertation proposes simple additions to the standard sequential programming model that allow the framework to parallelize them. In particular, the SPEC CINT2000 benchmark suite was used as a case study to show that this framework and programming model can be applied to many applications. This allows a software developer to develop in a sequential programming model, but still obtain parallel performance. In an initial implementation, the VELOCITY compiler obtained a speedup of 3.64x modifying only a few dozen lines out of one hundred thousand.

6.1 Future Directions

Potential future research based on this dissertation is focused on making it easier for automatic tools to extract parallelism from larger regions, in both sequential *and* parallel programs, while minimizing the burden placed on the programmer.

The ability to express non-determinism in a sequential programming language is a powerful technique that can be leveraged to facilitate parallelism in many applications, not just those in the SPEC CINT2000 benchmark suite. A base set of annotations useful across a wide variety of application domains should be enumerated by studying more applications. While these non-deterministic annotations are simpler than the use of locks and other parallel constructs, they still increase programmer overhead because they cannot be statically checked for correctness. Additional research into program analysis and annotations can help programmers identify how the non-determinism introduced by annotations affects program output or other aspects of program execution.

In addition to static compilation frameworks that can extract parallelism using these annotations, dynamic compilation frameworks can potentially extract more parallelism. Dynamic compilation systems have many useful properties, particularly profile information relevant to the current run and often better debugging information. The former is useful to facilitate better speculation, which often relies on expensive and potentially inaccurate static profiling. The latter is particularly important for automatic parallelization, as the programmer may not realize that the compiler has parallelized the program and yet needs to debug a multi-threaded program. As the number of cores on a processor grows, it also becomes more feasible to set aside one (or even several) cores specifically for the purpose of dynamically optimizing a program. Balancing this process with a running application will be an interesting area of future research.

Beyond sequential programs and runtime systems, there are many parallel programs that have and will continue to been written, both because the parallel programming paradigm sometimes matches the programming domain, as in hardware simulation, or because programmers feel that that they can extract the parallelism. For the former, the conceptual parallelism of the programming domain will only rarely match that of the underlying architecture, making it necessary to re-parallelize the application to aggressively utilize the underlying resources. For the latter, retargeting the program to a new architecture often entails significant rewriting of code to extract equivalent performance on a new machine. Finally, as multicore machines approach 10s or 100s of processors, even parallel programming models will need to rely on automatic thread extraction techniques to enable good performance. Because of these factors, research should continue into the application of automatic thread extraction techniques not just to sequential programs, but also to parallel programs as well.

Bibliography

[1] Message Passing Interface (MPI). http://www.mpi-forum.org.

[2] OpenMP. http://openmp.org.

[3] Parallel BZIP2 (PBZIP2) Data Compression Software. http://compression.ca/pbzip2.

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

[5] Haitham Akkary and Michael A. Driscoll. A Dynamic Multithreading Processor. *International Symposium on Microarchitecture,* pages 226–236, Dallas, Texas, December 1998.

[6] Ronald D. Barnes, Shane Ryoo, and Wen-mei W. Hwu. "Flea-Flicker" Multipass Pipelining: An Alternative to the High-Powered Out-of-Order Offense. *International Symposium on Microarchitecture,* pages 319–330, Barcelona, Spain, November 2005.

[7] D. Baxter, R. Mirchandaney, and J. H. Saltz. Run-Time Parallelization and Scheduling of Loops. *Symposium on Parallel Algorithms and Architectures,* pages 303–312, Santa Fe, New Mexico, June 1989.

[8] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger,

Lawrence Rauchwerger, and Peng Tu. Parallel Programming with Polaris. *Computer,* 29(12):78–82, December 1996.

[9] Matthew J. Bridges, Neil Vachharajani, Yun Zhang, and David I. August. Revisiting the Sequential Programming Model for Multi-Core. *International Symposium on Microarchitecture,* Chicago, Illinois, December 2007.

[10] Matthew J. Bridges, Neil Vachharajani, Yun Zhang, and David I. August. Revisiting the Sequential Programming Model for the MultiCore Era. *IEEE Micro,* January 2008.

[11] Michael Burke and Ron Cytron. Interprocedural Dependence Analysis and Parallelization. *Symposium on Compiler Construction,* pages 162–175, Palo Alto, California, October 1986.

[12] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The Atomos Transactional Programming Language. *Programming Language Design and Implementation,* pages 1–13, Ottawa, Canada, June 2006.

[13] R. S. Chappel, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous Subordinate Microthreading. *International Symposium on Computer Architecture,* pages 186–195, Atlanta, Georgia, May 1999.

[14] Ben-Chung Cheng and Wen-mei W. Hwu. Modular Interprocedural Pointer Analysis Using Access Paths: Design, Implementation, and Evaluation. *Programming Language Design and Implementation,* pages 57–69, Vancouver, Canada, June 2000.

[15] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. P. Shen. Speculative Precomputation: Long-Range Prefetching of Delinquent Loads. *International Symposium on Computer Architecture,* Anchorage, Alaska, May 2002.

[16] James C. Corbett. Evaluating Deadlock Detection Methods for Concurrent Software. *IEEE Transactions on Software Engineering,* 22(3):161–180, 1996.

[17] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press and McGraw-Hill, 1990.

[18] R. Cytron. DOACROSS: Beyond Vectorization for Multiprocessors. *International Conference on Parallel Programming,* pages 836–884, University Park, Pennsylvania, August 1986.

[19] James Russell Beckman Davies. Parallel Loop Constructs for Multiprocessors. Master's Thesis, Department of Computer Science, Department of Computer Science, University of Illinois, Urbana-Champaign, UIUCDCS-R-81-1070, University of Illinois, Urbana, IL, May 1981.

[20] Claudio Demartini, Radu Iosif, and Riccardo Sisto. A Deadlock Detection Tool for Concurrent Java Programs. *Software: Practice and Experience,* 29(7):577–603, John Wiley & Sons, 1999.

[21] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. Software Behavior Oriented Parallelization. *Programming Language Design and Implementation,* San Diego, California, June 2007.

[22] Zhao-Hui Du, Chu-Cheow Lim, Xiao-Feng Li, Chen Yang, Qingyu Zhao, and Tin-Fook Ngai. A Cost-Driven Compilation Framework for Speculative Parallelization of Sequential Programs. *Programming Language Design and Implementation,* pages 71–81, Washington, D.C. June 2004.

[23] Perry A. Emrath and David A. Padua. Automatic Detection of Nondeterminacy in Parallel Programs. *Workshop on Parallel and Distributed Debugging,* pages 89–99, 1988.

[24] Stephane Eranian. Perfmon: LInux Performance Monitoring for IA-64. `http://www.hpl.hp.com/research/linux/perfmon`, Hewlett-Packard Laboratories, 2003.

[25] 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.

[26] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. *Programming Language Design and Implementation,* pages 212–223, Montréal, Canada, June 1998.

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

[28] María Jesús Garzarán, Milos Prvulovic, José María Llabería, Víctor Viñals, Lawrence Rauchwerger, and Josep Torrellas. Tradeoffs in Buffering Speculative Memory State for Thread-Level Speculation in Multiprocessors. *ACM Transactions on Architecture Code Optimization,* 2(3):247–279, 2005.

[29] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. *International Conference on Architectural Support for Programming Languages and Operating Systems,* pages 291–303, San Jose, California, October 2002.

[30] Dan Grossman, Jeremy Manson, and William Pugh. What Do High-Level Memory Models Mean for Transactions? *Workshop on Memory System Performance and Correctness,* pages 62–69, San Jose, California, 2006.

[31] Bolei Guo, Matthew J. Bridges, Spyridon Triantafyllis, Guilherme Ottoni, Easwaran Raman, and David I. August. Practical and Accurate Low-Level Pointer Analysis.

International Symposium on Code Generation and Optimization, San Jose, California, March 2005.

[32] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. Interprocedural Parallelization Analysis in SUIF. *ACM Transactions on Programming Languages and Systems,* 27(4):662–731, 2005.

[33] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. The Stanford Hydra CMP. *IEEE Micro,* 20(2):71–84, 2000.

[34] Richard E. Hank, Wen-mei W. Hwu, and B. Ramakrishna Rau. Region-Based Compilation: An Introduction and Motivation. *International Symposium on Microarchitecture,* pages 158-168, Ann Arbor, Michigan, December 1995.

[35] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems,* 12(1), 1990.

[36] Intel Corporation. Intel New Processor Generations. Intel Corporation. `http://www.intel.com/pressroom/archive/releases/Intel_New_Processor_Generations.pdf`.

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

[38] Ken Kennedy and Kathryn S. McKinley. Maximizing Loop Parallelism and Improving Data Locality Via Loop Fusion and Distribution. *Workshop on Languages and Compilers for Parallel Computing,* pages 301–320, Ithaca, New York, August 1994.

[39] Hammond, Lance, Carlstrom, Brian D., Wong, Vicky, Chen, Mike, Kozyrakis, Christos, and Olukotun, Kunle. Transactional Coherence and Consistency: Simplifying Parallel Hardware and Software. *IEEE Micro,* 24(6), Nov-Dec 2004.

[40] Kevin M. Lepak and Mikko H. Lipasti. Silent Stores for Free. *International Symposium on Microarchitecture,* pages 22–31, Monterey, California, December 2000.

[41] Zhiyuan Li. Array Privatization for Parallel Execution of Loops. *International Conference on Supercomputing,* pages 313–322, Minneapolis, Minnesota, November 1992.

[42] Zhiyuan Li and Pen-Chung Yew. Efficient Interprocedural Analysis for Program Parallelization and Restructuring. *Conference on Parallel Programming: Experience with Applications, Languages and Systems,* pages 85–99, New Haven, Connecticut, July 1988.

[43] Shih-Wei Liao, Amer Diwan, Robert P. Bosch Jr., Anwar M. Ghuloum, and Monica S. Lam. SUIF Explorer: An Interactive and Interprocedural Parallelizer. *Symposium on Principles and Practice of Parallel Programming,* pages 37-48, Atlanta, Georgia, May 1999.

[44] Hongtao Zhong, Steven A. Lieberman, and Scott A. Mahlke. Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-Thread Applications. *International Symposium on High-Performance Computer Architecture,* Phoenix, Arizona, February 2007.

[45] J. T. Lim, A. R. Hurson, K. Kavi, and B. Lee. A Loop Allocation Policy for DOACROSS Loops. *Symposium on Parallel and Distributed Processing,* pages 240–249, New Orleans, Louisiana, October 1996.

[46] M. H. Lipasti and J. P. Shen. Exceeding the Dataflow Limit Via Value Prediction.

International Symposium on Microarchitecture, pages 226–237, Paris, France, December 1996.

[47] Glenn R. Luecke, Yan Zou, James Coyle, Jim Hoekstra, and Marina Kraeva. Deadlock Detection in MPI Programs. *Concurrency and Computation: Practice and Experience,* 14(11):911–932, John Wiley & Sons, 2002.

[48] S. F. Lundstorm and G. H. Barnes. A Controllable MIMD Architecture. *International Conference on Parallel Programming,* pages 19–27, 1980.

[49] Scott A. Mahlke, W. Y. Chen, J. C. Gyllenhaal, Wen-mei W. Hwu, P. P. Chang, and T. Kiyohara. Compiler Code Transformations for Superscalar-Based High-Performance Systems. *International Conference on Supercomputing,* pages 808–817, Minneapolis, Minnesota, November 1992.

[50] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan-Kaufmann Publishers, San Francisco, CA, 1997.

[51] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. *International Symposium on High-Performance Computer Architecture,* Anaheim, California, February 2003.

[52] Eugene W. Myers. A Precise Inter-Procedural Data Flow Algorithm. *Symposium on Principles of Programming Languages,* pages 219–230, Williamsburg, Virginia, January 1981.

[53] E. M. Nystrom, H.-S. Kim, and Wen-mei W. Hwu. Bottom-Up and Top-Down Context-Sensitive Summary-Based Pointer Analysis. *Static Analysis Symposium,* Verona, Italy, August 2004.

[54] Guilherme Ottoni and David I. August. Communication Optimizations for Global Multi-Threaded Instruction Scheduling. *International Conference on Architectural Support for Programming Languages and Operating Systems,* pages 222-232, Seattle, Washington, March 2008.

[55] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic Thread Extraction with Decoupled Software Pipelining. *International Symposium on Microarchitecture,* pages 105–116, Barcelona, Spain, November 2005.

[56] Guilherme Ottoni, Ram Rangan, Adam Stoler, Matthew J. Bridges, and David I. August. From Sequential Programs to Concurrent Threads. *IEEE Computer Architecture Letters,* 4, June 2005.

[57] David A. Padua. Multiprocessors: Discussion of Some Theoretical and Practical Problems. Ph.D. Thesis, Department of Computer Science, Department of Computer Science, University of Illinois, Urbana-Champaign, UIUCDCS-R-79-990, University of Illinois, Urbana, IL, November 1979.

[58] J. R. C. Patterson. Accurate Static Branch Prediction By Value Range Propagation. *Programming Language Design and Implementation,* pages 67-78, La Jolla, California, June 1995.

[59] Manohar K. Prabhu and Kunle Olukotun. Exposing Speculative Thread Parallelism in SPEC2000. *Symposium on Principles and Practice of Parallel Programming,* pages 142–152, Chicago, Illinois, July 2005.

[60] Zach Purser, Karthik Sundaramoorthy, and Eric Rotenberg. A Study of Slipstream Processors. *International Symposium on Microarchitecture,* pages 269–280, Monterey, California, December 2000.

[61] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I.

August. Parallel-Stage Decoupled Software Pipelining. *International Symposium on Code Generation and Optimization,* Boston, Massachusetts, April 2008.

[62] Ram Rangan. Pipelined Multithreading Transformations and Support Mechanisms. Ph.D. Thesis, Department of Computer Science, Princeton University, Princeton, NJ, United States, June 2007.

[63] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. Decoupled Software Pipelining with the Synchronization Array. *International Conference on Parallel Architectures and Compilation Techniques,* pages 177–188, Antibes Juan-les-Pins, France, September 2004.

[64] Lawrence Rauchwerger and David Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *ACM SIGPLAN Notices,* 30(6):218–232, 1995.

[65] Rajwar, Ravi and Goodman, James R. Transactional Execution: Toward Reliable, High-Performance Multithreading. *IEEE Micro,* 23(6):117-125, Nov-Dec 2003.

[66] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding Up Slicing. *Symposium on Foundations of Software Engineering,* pages 11–20, New Orleans, Louisiana, December 1994.

[67] Martin C. Rinard and Pedro C. Diniz. Commutativity Analysis: A New Analysis Framework for Parallelizing Compilers. *Programming Language Design and Implementation,* Philadelphia, Pennsylvania, May 1996.

[68] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems,* 15(4):391–411, 1997.

[69] John W. Sias, Sain-Zee Ueng, Geoff A. Kent, Ian M. Steiner, Erik M. Nystrom, and Wen-mei W. Hwu. Field-Testing IMPACT EPIC Research Results in Itanium 2. *International Symposium on Computer Architecture,* Munich, Germany, June 2004.

[70] Jeff Da Silva and J. Gregory Steffan. A Probabilistic Pointer Analysis for Speculative Optimizations. *International Conference on Architectural Support for Programming Languages and Operating Systems,* pages 416–425, San Jose, California, October 2006.

[71] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar Processors. *International Symposium on Computer Architecture,* pages 414–425, S. Margherita Ligure, Italy, June 1995.

[72] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The STAMPede Approach to Thread-Level Speculation. *ACM Transactions on Computer Systems,* 23(3):253–300, 2005.

[73] Hong-Men Su and Pen-Chung Yew. Efficient DOACROSS Execution on Distributed Shared-Memory Multiprocessors. *International Conference on Supercomputing,* pages 842–853, Albuquerque, New Mexico, November 1991.

[74] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. A Region-Based Compilation Technique for a Java Just-in-Time Compiler. *Programming Language Design and Implementation,* pages 312–323, San Diego, California, June 2003.

[75] Motoyasu Takabatake, Hiroki Honda, and Toshitsugu Yuba. Performance Measurements on Sandglass-Type Parallelization of DOACROSS Loops. *International Conference on High-Performance Computing and Networking,* pages 663–672, Amsterdam, The Netherlands, April 1999.

[76] Robert E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing,* 1(2):146-160, 1972.

[77] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs. *International Symposium on Microarchitecture,* Chicago, Illinois, December 2007.

[78] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. *Symposium on Compiler Construction,* Grenoble, France, April 2002.

[79] Spyros Triantafyllis, Matthew J. Bridges, Easwaran Raman, Guilherme Ottoni, and David I. August. A Framework for Unrestricted Whole-Program Optimization. *Programming Language Design and Implementation,* pages 61–71, Ottawa, Canada, June 2006.

[80] Jenn-Yuan Tsai, Jian Huang, Christoffer Amlo, David J. Lilja, and Pen-Chung Yew. The Superthreaded Processor Architecture. *IEEE Transactions on Computing,* 48(9):881–902, 1999.

[81] Peng Tu and David A. Padua. Automatic Array Privatization. *Compiler Optimizations for Scalable Parallel Systems Languages,* pages 247-284, 2001.

[82] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative Decoupled Software Pipelining. *International Conference on Parallel Architectures and Compilation Techniques,* Brasov, Romania, September 2007.

[83] Sriram Vajapeyam and Tulika Mitra. Improving Superscalar Instruction Dispatch and Issue By Exploiting Dynamic Code Sequences. *International Symposium on Computer Architecture,* pages 1–12, Denver, Colorado, June 1997.

[84] Christoph von Praun, Luis Ceze, and Calin Caşcaval. Implicit Parallelism with Ordered Transactions. *Symposium on Principles and Practice of Parallel Programming,* pages 79–89, San Jose, California, March 2007.

[85] Steven Wallace, Brad Calder, and Dean M. Tullsen. Threaded Multiple Path Execution. *International Symposium on Computer Architecture,* pages 238–249, Barcelona, Spain, June 1998.

[86] Nicholas Wang, Michael Fertig, and Sanjay Patel. Y-Branches: When You Come to a Fork in the Road, Take It. *International Conference on Parallel Architectures and Compilation Techniques,* pages 56–67, New Orleans, Louisiana, September 2003.

[87] Tom Way, Ben Breech, and Lori Pollock. Region Formation Analysis with Demand-Driven Inlining for Region-Based Optimization. *International Conference on Parallel Architectures and Compilation Techniques,* pages 24, Philadelphia, Pennsylvania, October 2000.

[88] Youfeng Wu and James R. Larus. Static Branch Frequency and Program Profile Analysis. *International Symposium on Microarchitecture,* pages 1–11, San Jose, California, November 1994.

[89] Antonia Zhai. Compiler Optimization of Value Communication for Thread-Level Speculation. Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, United States, January 2005.

[90] Antonia Zhai, J. Gregory Steffan, Christopher B. Colohan, and Todd C. Mowry. Compiler and Hardware Support for Reducing the Synchronization of Speculative Threads. *ACM Transactions on Architecture Code Optimization,* 5(1):1–33, 2008.

[91] Hongtao Zhong, Mojtaba Mehrara, Steve Lieberman, and Scott Mahlke. Uncovering Hidden Loop Level Parallelism in Sequential Applications. *International Symposium on High-Performance Computer Architecture,* Salt Lake City, Utah, February 2008.

[92] Craig Zilles and David Flint. Challenges to Providing Performance Isolation in Transactional Memories. *Workshop on Duplicating, Deconstructing, and Debunking,* 2005.