# AUTOMATICALLY EXPLOITING CROSS-INVOCATION PARALLELISM USING RUNTIME INFORMATION

JIALU HUANG

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: PROFESSOR DAVID I. AUGUST

OCTOBER 2012

© Copyright by Jialu Huang, 2013.

All Rights Reserved

# Abstract

Automatic parallelization is a promising approach to producing scalable multi-threaded programs for multi-core architectures. Most existing techniques parallelize independent loops and insert global synchronizations at the end of each loop invocation. For programs with few loop invocations, these global synchronizations do not limit parallel execution performance. However, for programs with many loop invocations, those synchronizations can easily become the performance bottleneck since they frequently force all threads to wait, losing potential parallelization opportunities. To address this problem, some automatic parallelization techniques apply static analyses to enable cross-invocation parallelization. Instead of waiting, threads can execute iterations from follow-up invocations if they do not cause any conflict. However, static analysis must be conservative and cannot handle irregular dependence patterns manifested by particular program inputs at runtime.

In order to enable more parallelization across loop invocations, this thesis presents two novel automatic parallelization techniques: DOMORE and SPECCROSS. Unlike existing techniques relying on static analyses, these two techniques take advantage of runtime information to achieve much more aggressive parallelization. DOMORE constructs a custom runtime engine which non-speculatively observes dependences at runtime and synchronizes iterations only when necessary; while SPECCROSS applies software speculative barriers to permit some of the threads to execute past the invocation boundaries. The two techniques are complimentary in the sense that they can parallelize programs with potentially very different characteristics. SPECCROSS, with less runtime overhead, works best when programs' cross-invocation dependences seldom cause any runtime conflict. DOMORE, on the other hand, has its advantage in handling dependences which cause frequent conflicts. Evaluating implementations of DOMORE and SPECCROSS demonstrates that both techniques can achieve much better scalability compared to existing automatic parallelization techniques.

# Acknowledgments

First, I would like to thank my advisor David I. August for guiding me throughout my years in graduate school. I appreciate the opportunities and challenges he gave me, which made this dissertation possible. David's passion in research and his insightful vision on research direction inspired me to find the research topic I am really interested in. During the years I worked in his research group, I learnt from him how to conduct research and to write scientific papers. More importantly, I learnt from him never to give up and always to pursue a bigger goal. I could not have made my way today without his encourage and support. I believe I will continue to benefit from the knowledge he taught me even after I graduate.

I would like to thank Prof. David Walker and Prof. Kai Li for reading this dissertation and providing insightful comments. I thank Kai also for his giving me advice and help during my first semester in Princeton when I was really stressed and frustrated by study, research and language problems. Additionally, I want to thank Prof. Jaswinder Singh and Prof. Doug Clark for serving as my thesis committee member and their feedbacks that helped me to polish and refine my thesis.

This dissertation would not have existed without the help and support from everyone in the Liberty Research Group. I have found great friendship with everyone in the Liberty research group. I would like to thank some senior group members, Neil Vachharajani, Matt Bridges, and Guilherme Ottoni for their guide and help in my early years into my PhD study. I must also thank Thomas Jablin, who spent a lot of time helping me improve my listening and spoken English at my early days in Princeton. I also want to thank Prakash Prabhu, Souymadeep Ghosh, Jae W. Lee, Stephen Beard, Matthew Zoufaly, Nick Johnson and Yun Zhang for helping me with my research. Throughout the years, we engaged in numerous brainstormings, discussions and presentations. I will never forget the sleepless nights we spent together before paper deadlines and the coffee and junk food we shared.

I also thank the entire staff of Princeton, and of the Department of Computer Science

in particular. Their professionalism really makes this such a great place to study and to do research. I especially thank Melissa Lawson who, as the highly effective Graduate Coordinator, makes all the bureaucracy so simple and allows us to completely focus on our studies. I also thank Bob Dondero for teaching me how to be a good TA and lecturer. His patience with students and passion in teaching left a deep impression on me.

There are a lot of friends I met over the years. I thank all of them for making my life in Princeton happy and unforgettable. I thank Zhiwei Yang for picking me up at the airport when I first came to the US and driving me around for furniture and grocery shopping. Jieqi Yu and Yi Li, being my roommates for many years, helped a lot with the cooking and cleaning especially when I had crazy schedules due to paper deadlines. I also would like to thank Xiaobai Chen for being a patient listener whenever I had trouble in research or personal life. And Ke Wan, I thank him for risking his life to teach me how to drive. He is the most patient coach I've ever seen.

I want to thank my parents for their unconditional love and support. Unlike most parents in Shanghai, They let me, their only child, explore the world in the way I want. I know they would very much prefer to having me around, but instead, they encouraged me to study abroad for so many years and to pursue the dreams I have. I cannot imagine how much they sacrificed to allow me to achieve what I have today. Everything I achieved today, is truly theirs.

Last, but not the least, I want to thank my boyfriend Haakon Ringberg, for making me happy and strong. We met each other in Princeton Computer Science Department when he himself was still a graduate student. Throughout the years, he offered me priceless tips about how to survive the graduate school. His support and encourage helped me overcome difficulties and move forward. I also thank him for introducing me to his two families. I thank his American guardian parents, Bruce and Beverly Shriver for inviting me to their home and treating me with lots of delicious food. I thank his Norwegian parents, Tore Larsen and Unni Ringberg for inviting me to their home in Norway. My trips to Tromsø

were fantastic. For the first time in my life, I experienced 24-hour daylight and 24-hour night. I would very much love to invite all of them to my Chinese home and show them around the land I love.

# Contents


Abstract . . . . . . . . . . iii

Acknowledgments . . . . . . . . . . i

List of Figures . . . . . . . . . . vii

**1 Introduction** **1**

1.1 Limitations of Existing Approaches . . . . . . . . . . 2

1.2 Contributions . . . . . . . . . . 8

1.3 Dissertation Organization . . . . . . . . . . 9

**2 Background** **11**

2.1 Limitations of Analysis-based Approaches in Automatic Parallelization . . 11

2.2 Intra-Invocation Parallelization . . . . . . . . . . 13

2.3 Cross-Invocation Parallelization . . . . . . . . . . 20

**3 Non-Speculatively Exploiting Cross-Invocation Parallelism Using Runtime Information** **24**

3.1 Motivation and Overview . . . . . . . . . . 25

3.2 Runtime Synchronization . . . . . . . . . . 28

3.2.1 Detecting Dependences . . . . . . . . . . 28

3.2.2 Generating Synchronization Conditions . . . . . . . . . . 29

3.2.3 Synchronizing Iterations . . . . . . . . . . 30

3.2.4 Walkthrough Example . . . . . . . . . . 31

3.3 Compiler Implementation . . . . . . . . . . 34
3.3.1 Partitioning Scheduler and Worker . . . . . . . . . . 34
3.3.2 Generating Scheduler and Worker Functions . . . . . . . . . . 35
3.3.3 Scheduling Iterations . . . . . . . . . . 39
3.3.4 Generating the computeAddr function . . . . . . . . . . 39
3.3.5 Putting It Together . . . . . . . . . . 40
3.4 Enable DOMORE in SPECCROSS . . . . . . . . . . 42
3.5 Related Work . . . . . . . . . . 45
3.5.1 Cross-invocation Parallelization . . . . . . . . . . 45
3.5.2 Synchronization Optimizations . . . . . . . . . . 45
3.5.3 Runtime Dependence Analysis . . . . . . . . . . 46

**4 Speculatively Exploiting Cross-Invocation Parallelism 48**

4.1 Motivation and Overview . . . . . . . . . . 50
4.1.1 Limitations of analysis-based parallelization . . . . . . . . . . 50
4.1.2 Speculative cross-invocation parallelization . . . . . . . . . . 52
4.1.3 Automatic cross-invocation parallelization with software-only speculative barrier . . . . . . . . . . 54
4.2 SPECCROSS Runtime System . . . . . . . . . . 59
4.2.1 Misspeculation Detection . . . . . . . . . . 59
4.2.2 Checkpointing and Recovery . . . . . . . . . . 61
4.2.3 Runtime Interface . . . . . . . . . . 63
4.3 SPECCROSS Parallelizing Compiler . . . . . . . . . . 68
4.4 SPECCROSS Profiling . . . . . . . . . . 72
4.5 Related Work . . . . . . . . . . 72
4.5.1 Barrier Removal Techniques . . . . . . . . . . 72
4.5.2 Alternative Synchronizations . . . . . . . . . . 73
4.5.3 Transactional Memory Supported Barrier-free Parallelization . . . . 73

4.5.4 Load Balancing Techniques . . . . . . . . . . . . . . . . . . . . . . 74
4.5.5 Multi-threaded Program Checkpointing . . . . . . . . . . . . . . . . 75
4.5.6 Dependence Distance Analysis . . . . . . . . . . . . . . . . . . . . . 75

**5 Evaluation** **76**

5.1 DOMORE Performance Evaluation . . . . . . . . . . . . . . . . . . . . . 77
5.2 SPECCROSS Performance Evaluation . . . . . . . . . . . . . . . . . . . 81
5.3 Comparison of DOMORE, SPECCROSS and Previous Work . . . . . . . . 86
5.4 Case Study: FLUIDANIMATE . . . . . . . . . . . . . . . . . . . . . . . 87
5.5 Limitations of Current Parallelizing Compiler Infrastructure . . . . . . . . 90

**6 Future Directions and Conclusion** **92**

6.1 Future Directions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
6.2 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94

# List of Figures

1.1 Scientists spend large amounts of time waiting for their program to generate results. Among the 114 interviewed researchers from 20 different departments in Princeton University, almost half of them have to wait days, weeks or even months for their simulation programs to finish. . . . . . . . . 3

1.2 Types of parallelism exploited in scientific research programs: one third of the interviewed researchers do not use any parallelism in their programs; others mainly use job parallelism or borrow already parallelized programs. 5

1.3 Example of parallelizing a program with barriers . . . . . . . . . . . . . . 7

1.4 Comparison between executions with and without barriers. A block with label $x.y$ represents the $y^{th}$ iteration in the $x^{th}$ loop invocation. . . . . . . . 7

1.5 Contribution of this thesis work . . . . . . . . . . . . . . . . . . . . . . . . 9

2.1 Sequential Code with Two Loops . . . . . . . . . . . . . . . . . . . . . . . 13

2.2 Performance sensitivity due to memory analysis on a shared-memory machine . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

2.3 Intra-invocation parallelization techniques which rely on static analysis ($X.Y$ refers to the $Y_{th}$ statement in the $X_{th}$ iteration of the loop): (a) DOALL concurrently executes iterations among threads and no inter-thread synchronization is necessary; (b) DOANY applies locks to guarantee atomic execution of function `malloc`; (c) LOCALWRITE goes through each `node` and each worker thread only updates the node belonging to itself. . . 14

2.4 Sequential Loop Example for DOACROSS and DSWP . . . . . . . . . . . 16

2.5 Parallelization Execution Plan for DOACROSS and DSWP . . . . . . . . . 16

2.6 Example Loop which cannot be parallelized by DOACROSS or DSWP . . 17

2.7 PDG after breaking the loop exit control dependence . . . . . . . . . . . . 19

2.8 TLS and SpecDSWP schedules for the loop shown in Figure 2.6 . . . . . . 19

3.1 Example program: (a) Simplified code for a nested loop in CG (b) PDG for inner loop. The dependence pattern allows DOALL parallelization. (c) PDG for outer loop. Cross-iteration dependence deriving from E to itself has manifest rate 72.4%. . . . . . . . . . . . . . . . . . . . . . . . . . . 25

3.2 Comparison of performance with and without cross-invocation parallelization : (a) DOALL is applied to the inner loop. Frequent barrier synchronization occurs between the boundary of the inner and outer loops. (b) After the partitioning phase, DOMORE has partitioned the code without inserting the runtime engine. A scheduler and three workers execute concurrently, but worker threads still synchronize after each invocation. (c) DOMORE finalizes by inserting the runtime engine to exploit cross-invocation parallelism. Assuming iteration 2 from invocation 2 (2.2) depends on iteration 5 from invocation 1 (1.5). Scheduler detects the dependence and synchronizes those two iterations. . . . . . . . . . . . . . . . . . . . . . . 27

3.3 Performance improvement of CG with and without DOMORE. . . . . . . . 28

3.4 Overview of DOMORE compile-time transformation and runtime synchronization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

3.5 Scheduler scheme running example: (a) Table showing original invocation/iteration, array element accessed in iteration, thread the iteration is scheduled to, combined iteration number, and helper data structure values (b) Execution of the example. . . . . . . . . . . . . . . . . . . . . . . . 33

3.6 Running example for DOMORE code generation: (a) Pseudo IR for CG code; (b) PDG for example code. Dashed lines represent cross-iteration and cross-invocation dependences for inner loop. Solid lines represent other dependences between inner loop instructions and outer loop instructions. (c) $DAG_{SCC}$ for example code. $DAG_{SCC}$ nodes are partitioned into scheduler and worker threads. (d) and (e) are code generated by DOMORE MTCG algorithm (3.3.2). . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
3.7 Generated code for example loop in CG. Non-highlighted code represents initial code for scheduler and worker functions generated by DOMORE's MTCG (Section 3.3.2). Code in grey is generated in later steps for iteration scheduling and synchronization. . . . . . . . . . . . . . . . . . . . . . . . . 41
3.8 Execution plan for DOMORE before and after duplicating scheduler code to worker threads. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
3.9 Optimization for DOMORE technique: duplicating scheduler code on all worker threads to enable DOMORE in SPECCROSS framework. . . . . . . 44

4.1 Example program demonstrating the limitation of DOMORE transformation 49
4.2 Example of parallelizing a program with different techniques . . . . . . . . 50
4.3 Overhead of barrier synchronizations for programs parallelized with 8 and 24 threads . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
4.4 Execution plan for TM-style speculation: each block $A.B$ stands for the $B_{th}$ iteration in the $A_{th}$ loop invocation: iteration $2.1$ overlaps with iterations $2.2, 2.3, 2.4, 2.7, 2.8$, thus its memory accesses need to be compared with theirs even though all these iterations come from the same loop invocation and are guaranteed to be independent. . . . . . . . . . . . . . . . . . 52

4.5 Overview of SPECCROSS: At compile time, the SPECCROSS compiler detects code regions composed of consecutive parallel loop invocations, parallelizes the code region and inserts SPECCROSS library functions to enable barrier speculation. At runtime, the whole program is first executed speculatively without barriers. Once misspeculation occurs, the checkpoint process is woken up. It kills the original child process and spawns new worker threads. The worker threads will re-execute the misspeculated epochs with non-speculative barriers. . . . . . . . . . . . . . . . . . . . . . . . . . . 56
4.6 Timing diagram for SPECCROSS showing epoch and task numbers. A block with label $<A, B>$ indicates that the thread updates its epoch number to $A$ and task number to $B$ when the task starts executing. . . . . . . . . 57
4.7 Pseudo-code for worker threads and checker thread . . . . . . . . . . . . . 58
4.8 Data structure for Signature Log . . . . . . . . . . . . . . . . . . . . . . . 62
4.9 Demonstration of using SPECCROSS runtime library in a parallel program . 67

5.1 Performance comparison between code parallelized with pthread barrier and DOMORE. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
5.2 Performance comparison between code parallelized with pthread barrier and SPECCROSS. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
5.3 Loop speedup with and without misspeculation for execution with 24 threads: the number of checkpoints varies from 2 to 100. A misspeculation is randomly triggered during the speculative execution. With more checkpoints, overhead in checkpointing increases; however overhead in re-execution after misspeculation reduces. . . . . . . . . . . . . . . . . . . . . . . . . . 85
5.4 Best performance achieved by this thesis work and previous work . . . . . 87
5.5 Outermost loop in FLUIDANIMATE . . . . . . . . . . . . . . . . . . . . . 88
5.6 Performance improvement of FLUIDANIMATE using different techniques. 89

# Chapter 1

# Introduction

The computing industry has relied on steadily increasing clock speeds and uniprocessor micro-architectural improvements to deliver reliable performance enhancements for a wide range of applications. Unfortunately, since 2004, the microprocessor industry fell off past trends due to increasingly unmanageable design complexity, power and thermal issues. In spite of this stall in processor performance improvements, Moores Law still remains in effect. Consistent with historic trends, the semiconductor industry continues to double the number of transistors integrated onto a single die every two years. Since conventional approaches to improving program performance with these transistors has faltered, microprocessor manufacturers leverage these additional transistors by placing multiple cores on the same die. These multi-core processors can improve system throughput and potentially speed up multi-threaded applications, but the latency of any single-thread of execution remains unchanged. Consequently, to take full advantage of multi-core processors, applications must be multi-threaded, and they must be designed to efficiently use the resources provided by the processor.

## 1.1 Limitations of Existing Approaches

One could consider merely requiring the programmers to write efficient multi-threaded code to take advantage of the many processor cores. This is not a successful strategy in the general case, however. First, writing multi-threaded software is inherently more difficult than writing single-threaded codes. To ensure correctness, programmers must reason about concurrent accesses to shared data and insert sufficient synchronization to ensure data accesses are ordered correctly. Simultaneously, programmers must prevent excessive synchronizations from rendering the multi-threaded program no better than its single-threaded counterpart. Active research in automatic tools to identify deadlock, livelock, race conditions, and performance bottlenecks [14, 17, 21, 37, 65] in multi-threaded programs is a testament to the difficulty of achieving this balance. Second, there are many legacy applications that are single-threaded. Even if the source code for these applications were available, it would take enormous programming effort to translate these programs into well-performing parallel equivalents. Finally, even if efficient multi-threaded applications could be written for a particular multi-core system, these applications may not perform well on other multi-core systems. The performance of multi-threaded applications is very sensitive to the particular system for which it was optimized. This variance is due to, among other factors, the relation between synchronization overhead and memory subsystem implementation (e.g., size of caches, number of caches, what caches are shared, coherence implementation, memory consistency model, etc.) and the relation between number of application threads and available hardware parallelism (e.g., number of cores, number of threads per core, cost of context switch, etc.). Writing an application that is portable across multiple processors would prove extremely challenging.

A recent survey [55] conducted by the Liberty Research Group suggested that most scientists are making minimal effort to parallelize their software, often due to the complexities involved. This survey covered 114 randomly-selected researchers from 20 different departments in science and engineering disciples in Princeton University. Among those

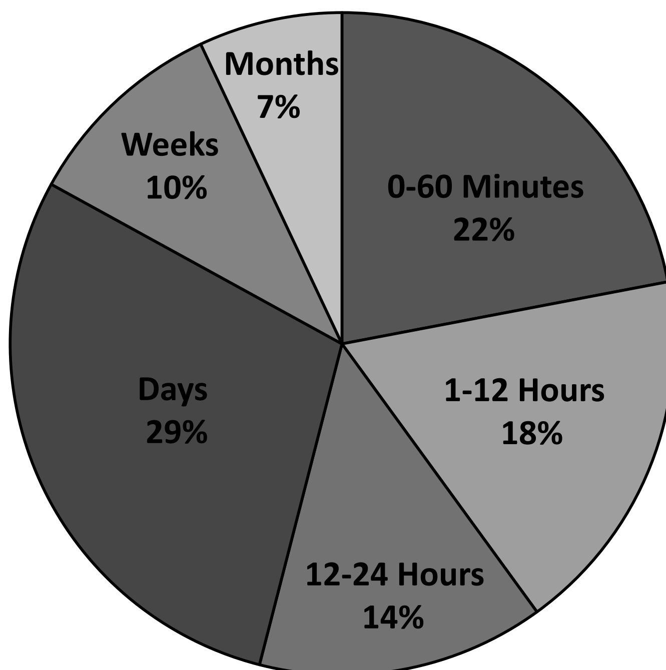



Figure 1.1: Scientists spend large amounts of time waiting for their program to generate results. Among the 114 interviewed researchers from 20 different departments in Princeton University, almost half of them have to wait days, weeks or even months for their simulation programs to finish.

researchers, almost half of them have to wait for days, weeks or even months for their simulation programs to finish (shown in Figure 1.1), although all of them have access to parallel computing resources that could significantly reduce the computational experimentation latency. This delay in their research cycle prevents them from pursuing bigger research plans. For example, they have to limit the parameter space to be explored or sacrifice the accuracy of the simulation results in exchange for tolerable program execution time. Figure 1.2 shows that one third of the researchers do not benefit from the parallel computing resources because they do not use any type of parallelism in their programs. Among researchers who do use parallelism, most only know how to apply job parallelism. Job parallelism speeds up execution with multiple input data sets. However, it does not help execution with a specific input data set, which is agreed to be as important. The rare researchers who take advantage of more advanced forms of parallelism (MPI, GPU) mainly borrow already par-

allelized code and do not know how to further adjust the parallelism to their own computing environment. These survey results imply that researchers need parallelism to achieve bigger research goals. Nevertheless, they are not competent enough to write scalable parallel programs by themselves.

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

Numerous compiler-based automatic parallelization techniques have been proposed in the past. Some of them [1, 15] achieved success in parallelizing array-based programs with regular memory accesses and limited control flow. More recent techniques [41, 52, 54, 59, 61, 64] perform speculation and pipeline style parallelization to successfully parallelize general purpose codes with arbitrary control flow and memory access patterns. However, all these automatic parallelization techniques only exploit loop level parallelism. A loop is a sequence of statements that can be executed 0, 1, or any finite number of times. A single time execution of the sequence of statements is referred to as a loop iteration and one execution of all iterations within a loop is defined as a loop invocation. These techniques parallelize each loop iteration and globally synchronize at the end of each loop invocation. Consequently, programs with many loop invocations will have to synchronize frequently.

These parallelization techniques fail to deliver scalable performance because synchronization forces all threads to wait for the last thread to finish an invocation [44]. At high thread counts, threads spend more time idling at synchronization points than doing useful computation. There is an opportunity to improve the performance by exploiting additional parallelism. Often, iterations from different loop invocations can execute concurrently

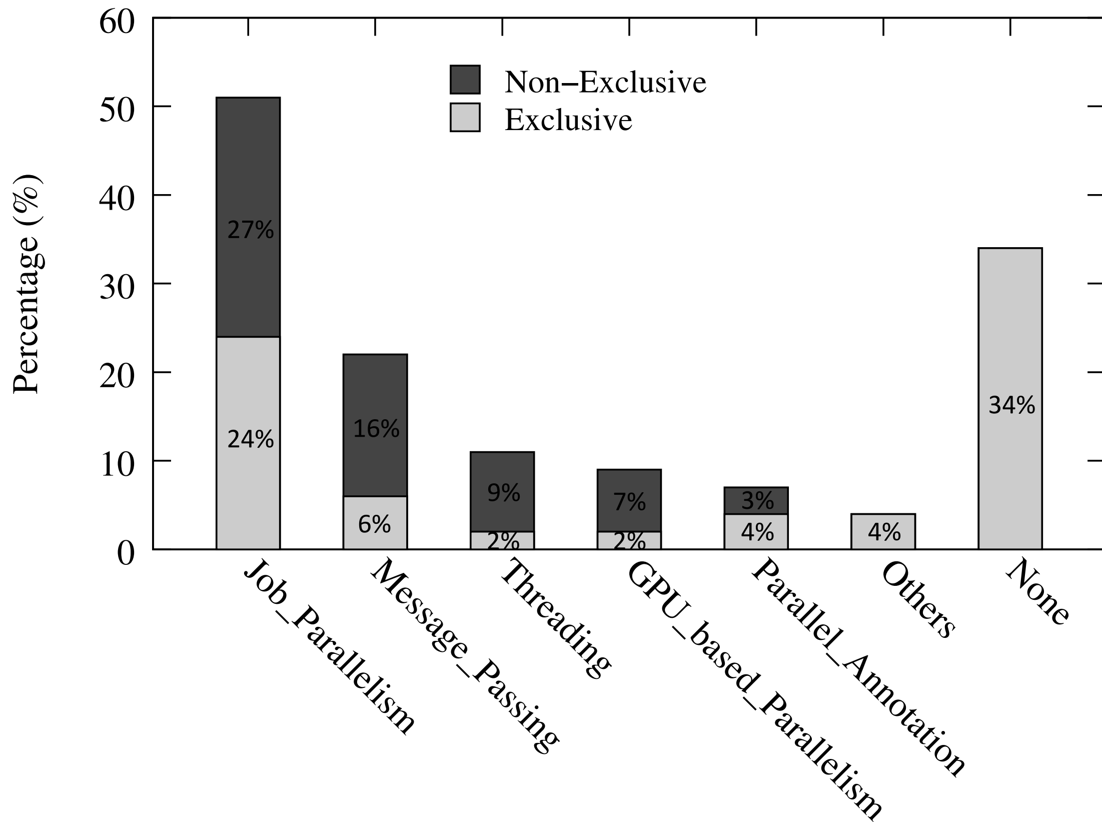



Figure 1.2: Types of parallelism exploited in scientific research programs: one third of the interviewed researchers do not use any parallelism in their programs; others mainly use job parallelism or borrow already parallelized programs.

without violating program semantics. Instead of waiting, threads begin iterations from subsequent invocations.

A simple code example in Figure 1.3(a) can be used to demonstrate the benefits of exploiting cross-invocation parallelism. In this example, inner loop `L1` updates array elements in array `A` while inner loop `L2` reads the elements from array `A` and uses the values to update array `B`. The whole process is repeated `TIMESTEP` times. Both `L1` and `L2` are DOALLable [1]. However, barriers must be placed between these two loops since data dependences exist across iterations of the two loops (e.g., iteration 1 of `L2` depends on iteration 1 and 2 of `L1`).

Figure 1.4(a) shows the execution plan for this program with global synchronizations(barriers). Each block in the graph stands for an iteration in a certain loop invocation (e.g., block `1.5`

is iteration 5 in the first invocation of loop L1). Typically, threads do not reach barriers at the same time for a variety of reasons. For instance, each thread may be assigned different number of tasks and the execution time of each task may vary. All threads are forced to stall at barriers after each parallel invocation, losing potential parallelism. Figure 1.4(b) shows a parallel execution plan after naïvely removing barriers. Without barriers, tasks from before and after a barrier may overlap, resulting in better performance.

A few automatic parallelization techniques exploit cross-invocation parallelism [22, 49, 71, 75]. Cross-invocation parallelization requires techniques for respecting cross-invocation dependences without resorting to coarse-grained barrier synchronization. Some techniques [22, 75] respect dependences by combining several small loops into a single larger loop. This approach side-steps the problem of exploiting cross-invocation parallelism by converting it into cross-iteration parallelism. Other approaches [49, 71] carefully partition the iteration space in each loop invocation so that cross-invocation dependences are never split between threads. However, both techniques rely on static analyses. Consequently, they cannot adapt to the dependence patterns manifested by particular inputs at runtime. Many statically detected dependences may only manifest under certain input conditions. For many programs, these dependences rarely manifest given the most common program inputs. By adapting to the dependence patterns of specific inputs at runtime, programs can exploit additional cross-invocation parallelism to achieve greater scalability.

```
main () {
  f();
}


f() {
  for (t = 0; t < TIMESTEP; t++) {
L1:   for (i = 0; i < M; i++) {
        A[i] = do_work(B[i], B[i+1]);
      }

L2:   for (j = 1; j < M+1; j++) {
        B[j] = do_work(A[j-1], A[j]);
      }
  }
}
```

(a) Sequential Program

```
main () {
  for (i = 0; i < NUM_THREADS; i++)
    create_thread(par_f, i);
}

par_f(threadID) {
  for (t = 0; t < TIMESTEP; t++) {
L1:   for (i = threadID; i < M; i=i+NUM_THREADS) {
        A[i] = do_work(B[i], B[i+1]);
      }
      pthread_barrier_wait(&barrier);

L2:   for (j = threadID; j < M+1; j=j+NUM_THREADS) {
        B[j] = do_work(A[j-1], A[j]);
      }
      pthread_barrier_wait(&barrier);
  }
}
```

(b) Parallelized Program

Figure 1.3: Example of parallelizing a program with barriers

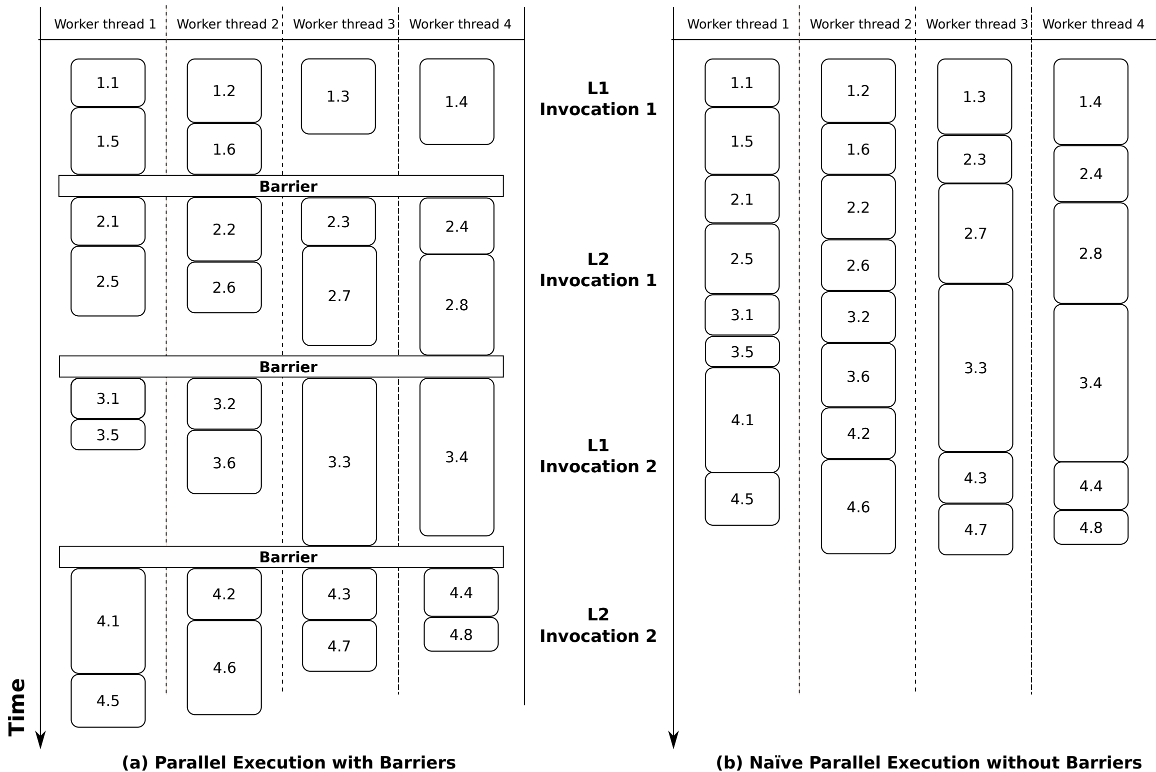



Figure 1.4: Comparison between executions with and without barriers. A block with label $x.y$ represents the $y^{th}$ iteration in the $x^{th}$ loop invocation.

## 1.2 Contributions

Figure 1.5 demonstrates the contribution of this thesis work compared to prior works. This thesis work presents two novel automatic parallelization techniques (DOMORE and SPECCROSS) that capture dynamic cross-invocation parallelism. Unlike existing techniques, DOMORE and SPECCROSS gather cross-invocation dependence information at runtime. Even for programs with irregular dependence patterns, DOMORE and SPECCROSS synchronize iterations which depend on each other and allow iterations without dependences to execute concurrently. As a result, they are able to enable more cross-invocation parallelization and achieves more scalable performance.

As a non-speculative technique, DOMORE first identifies the code region containing the targeted loop invocations, and then transforms the program by dividing the region into a scheduler thread and several worker threads. The scheduler thread contains code to detect memory access conflicts between loop iterations and code to schedule and dispatch loop iterations from different loop invocations to worker threads. In order to detect access violations, the scheduler duplicates the instructions used for calculating the addresses of memory locations to be accessed in each loop iteration. As a result, at runtime, it knows which iterations access the common memory locations and coordinates the execution of these conflicting iterations by generating and forwarding synchronization conditions to the worker threads. A synchronization condition tells the worker thread to wait until another worker thread finishes executing the conflicting iteration. Consequently, only threads waiting on the synchronization conditions must stall, and iterations from consecutive loop invocations may execute in parallel.

SPECCROSS parallelizes independent loops and replaces the barrier synchronization between two loop invocations with its speculative counterpart. Unlike non-speculative barriers which pessimistically synchronize to enforce dependences, speculative techniques allow threads to execute past barriers without stalling. Speculation allows programs to optimistically execute potentially dependent instructions and later check for misspeculation.

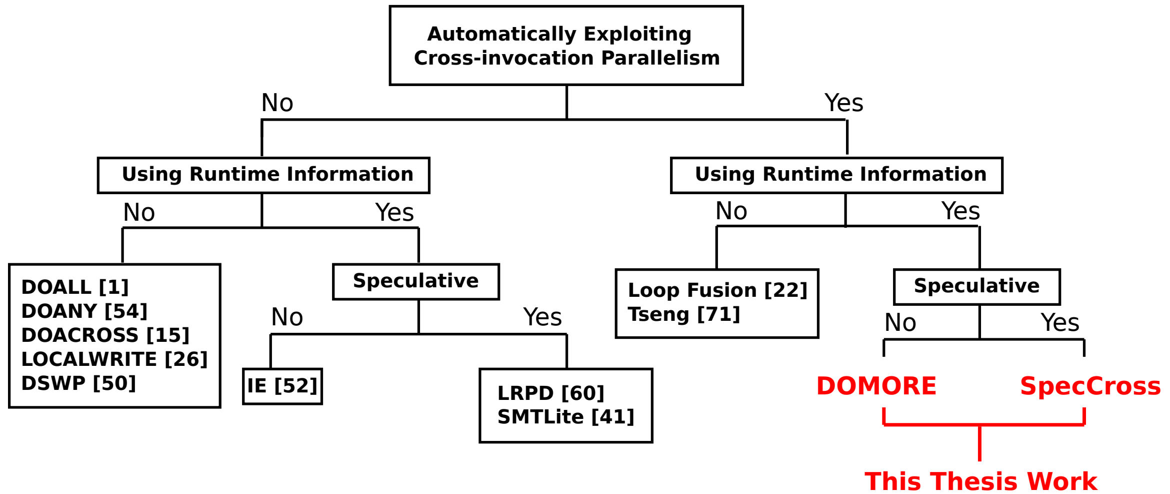



Figure 1.5: Contribution of this thesis work

If misspeculation occurs, the program recovers using checkpointed non-speculative state. Speculative barriers improve performance by synchronizing only on misspeculation.

The two techniques are complimentary in the sense that they can parallelize programs with potentially very different characteristics. SPECCROSS, with less runtime overhead, works best when programs' cross-invocation dependences seldom cause any runtime conflict. While DOMORE has its advantage in handling dependences which cause frequent conflicts. Implementation and evaluation demonstrate that both techniques can achieve much better scalability compared to existing automatic parallelization techniques. DOMORE achieves a geomean speedup of 2.1× over codes without cross-invocation parallelization and 3.2× over the original sequential performance on 24 cores. SPECCROSS achieves a geomean speedup of 4.6× over the best sequential execution, which compares favorably to a 1.3× speedup obtained by parallel execution without any cross-invocation parallelization.

## 1.3 Dissertation Organization

Chapter 2 examines existing intra- and inter- invocation parallelization techniques, characterizing their applicability and scalability. This discussion motives DOMORE and SPEC-

CROSS. Chapter 3 describes the design and implementation details of DOMORE, the first non-speculative runtime technique to exploit cross-invocation parallelism. Chapter 4 introduces its speculative counterpart technique SPECCROSS. A quantitative evaluation of DOMORE and SPECCROSS is given in Chapter 5. Finally, Chapter 6 describes future avenues of research and summarizes the conclusions of this dissertation.

# Chapter 2

# Background

Imprecise and fragile static analyses limit the effectiveness of existing cross-invocation parallelization techniques. Addressing this vulnerability is the focus of this thesis. In this section we explain the state of the art in automatic parallelization techniques. In Section 2.1 we first identify the limitations of conventional analysis-based approaches to automatic parallelization. In Section 2.2 we provide a detailed discussion of current intra-invocation parallelization techniques, explaining how some of them compensate for the conservative nature of their analyses. Finally, in Section 2.3 we present existing cross-invocation parallelization techniques; in particular, how all of them rely on static analysis, which motivates the work of DOMORE and SPECCROSS.

## 2.1 Limitations of Analysis-based Approaches in Automatic Parallelization

Automatic parallelization is an ideal solution which frees programmers from the difficulties of parallel programming and platform-specific performance tuning. Parallelizing compilers can automatically parallelize affine loops [2, 7]. `Loop_A` in Figure 2.1 shows an example loop. If a compiler proves that all memory variables in the body of the function `foo` do not

alias the array `regular` via inter-procedural analysis, the loop can be safely parallelized. Therefore, the utility of an automatic parallelizing compiler is largely determined by the quality of its memory dependence analysis.

In some cases, static analysis may be imprecise. For example, within the function `foo`, assume that there is a read from or write to the array element `regular[i+M]`, (and the size of the array is greater than (`M+N`)), where `M` is an input from the user. In this case, `Loop_A` may not be DOALL-able depending on the value of `M`. If `M` is greater than `N`, the loop is DOALL-able; otherwise, it is not. Some research compilers such as SUIF [2] and Polaris [7, 63] integrate low-cost runtime analysis capabilities to insert a small test code to check the value of `M` at runtime to select either a sequential or parallel version of the loop accordingly. However, the coverage of these techniques is mostly restricted to the cases when a predicate can be extracted outside the analyzed loop and a low cost runtime test can be generated [63]. They cannot be applied to `Loop_B` in Figure 2.1, for example, where an index array is used to access the array `irregular` and a simple predicate cannot be extracted outside the loop to test the `if` condition within the loop body.

Another issue with automatic parallelization is the fragility of static analysis. Figure 2.2 from [32] illustrates how fragile static analysis can be with a small change in the program. In this example, the automatic parallelizer can easily parallelize the unmodified PolyBench benchmarks [53] using static arrays. However, if the static arrays are replaced with dynamically allocated arrays, it not only suppresses some of the optimizations previously applied but also blocks parallelization for several benchmarks since heap objects are generally more difficult to analyze. This leads to runtime performance that is highly sensitive to the implementation style.

Therefore, analysis-based approaches are not sufficient for parallelization of even array-based applications, let alone pointer-based ones, having irregular memory accesses and complex control flows. Moreover, recursive data structures, dynamic memory allocation, and frequent accesses to shared variables pose additional challenges. Imprecise, fragile

```
1: Loop_A:
2: for (int i=0; i<N; i++)
3:  regular[i] += foo(i);
4:
```

```
 5: Loop_B:
 6: for (int i=0; i<N; i++) {
 7:   irregular[idx[i]] += foo(i);
 8:   if (irregular[idx[i]] > error)
 9:     printf("I/O operation!");
10: }
```

Figure 2.1: Sequential Code with Two Loops

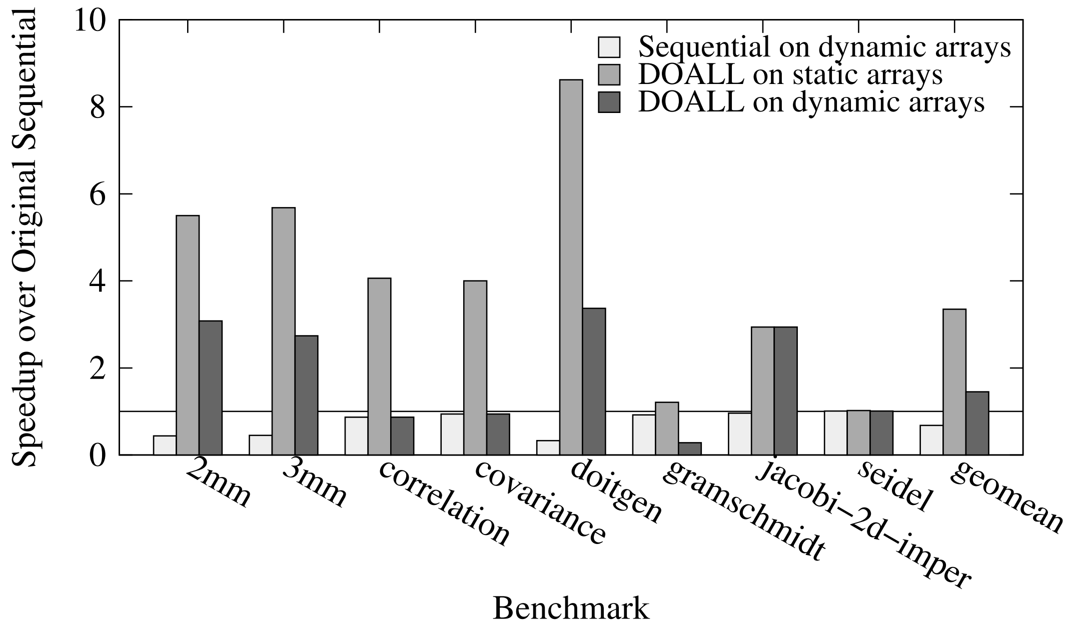



Figure 2.2: Performance sensitivity due to memory analysis on a shared-memory machine

static analysis has severely limited the applicability of conventional automatic parallelization. Therefore, this thesis work is interested in addressing this vulnerability.

## 2.2 Intra-Invocation Parallelization

Intra-invocation parallelization techniques focus on parallelizing iterations within the same loop invocation. They synchronize the parallel execution at the end of a loop invocation to respect cross-invocation dependences. Numerous intra-invocation parallelization techniques have been proposed. We categorize them into three groups.

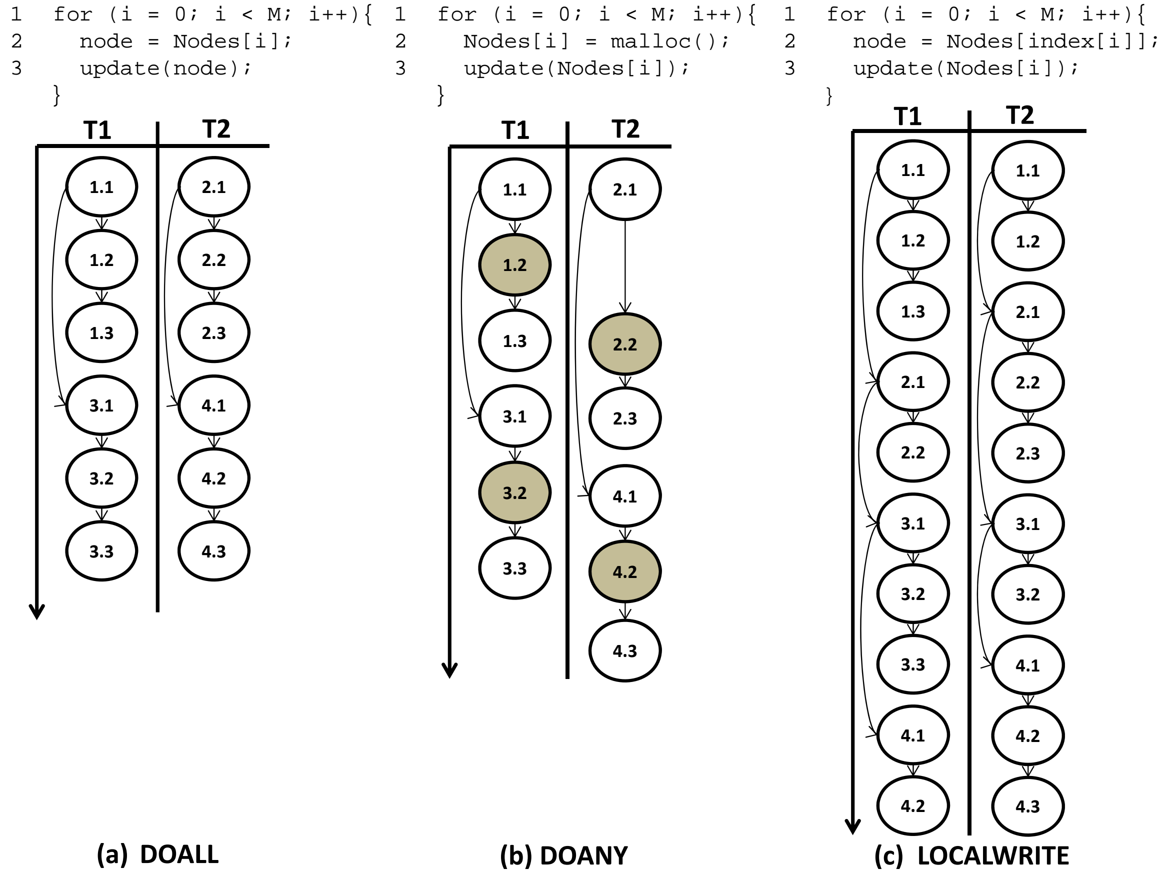



Figure 2.3: Intra-invocation parallelization techniques which rely on static analysis ($X.Y$ refers to the $Y_{th}$ statement in the $X_{th}$ iteration of the loop): (a) DOALL concurrently executes iterations among threads and no inter-thread synchronization is necessary; (b) DOANY applies locks to guarantee atomic execution of function `malloc`; (c) LOCALWRITE goes through each `node` and each worker thread only updates the node belonging to itself.

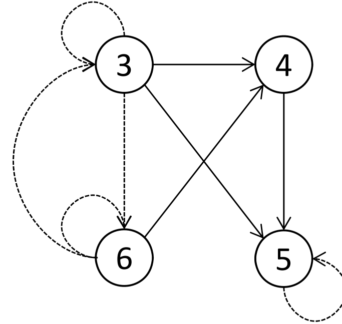
Parallelization techniques such as DOALL [1], DOANY [54, 74], LOCALWRITE [26], DOACROSS [15] and DSWP [50] belong to the first criteria. The applicability and scalability of these techniques fully depend on the quality of static analysis.

DOALL parallelization can be applied to a loop where each iteration is independent of all other loop iterations. Figure 2.3(a) illustrates such a DOALL loop. In the figure, $X.Y$ refers to the $Y_{th}$ statement in the $X_{th}$ iteration of the loop. DOALL loops are parallelized by allocating sets of loop iterations to different threads. Although DOALL parallelization often yields quite scalable performance improvement, its applicability is limited. In

most loops, dependences exist across loop iterations. Figure 2.3(b) shows a slightly different loop. The cross-iteration dependence derived from `malloc()` to itself prohibits DOALL parallelization. DOANY, instead, synchronizes these `malloc()` function calls using locks. Locks guarantee only one thread can execute `malloc()` each time (as shown in Figure 2.3(b)). Locks enforces atomic execution but do not guarantee a specific execution order of those `malloc()` calls. As a result, DOANY requires the protected operations to be commutative. Figure 2.3(c) demonstrates another loop whose cross-iteration dependences are caused by irregular accesses to array elements (through an index array). DOANY fails to parallelize that loop since the execution order of `update()` matters. In this case, LOCALWRITE parallelization technique works if it can find a partition of the shared memory space which guarantees that each thread only accesses and updates the memory partition owned by itself. For this loop example, LOCALWRITE partitions the `Node` array into two sections and assigns each section to one of the worker threads. Each worker thread executes all of the iterations, but before it executes statement `3`, it checks whether that node falls within its own memory partition. If it does not, the worker thread simply skips statement `3` and starts executing the next iteration. LOCALWRITE's performance gain is often limited by the redundant computation among threads (statements `1` and `2` in this example). Meanwhile, a partition of the memory space is not always available at compile time since the dependence patterns may be determined at runtime by specific inputs. Overall, these three parallelization techniques can only handle limited types of cross-iteration dependences. They require the static analysis to prove that no other parallelization prohibiting dependences exist.

In contrast to these three techniques, DOACROSS and DSWP parallelization can handle any type of dependences. For example, consider the loop shown in Figure 2.4(a). Figure 2.4(b) shows the program dependence graph (PDG) corresponding to the code. In the PDG, edges that participate in dependence cycles are shown as dashed lines. Since the statements on lines 3 and 6 and the statement on line 5 each form a dependence cycle,

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




**(a) Loop with cross-iteration dependences** **(b) PDG**

Figure 2.4: Sequential Loop Example for DOACROSS and DSWP

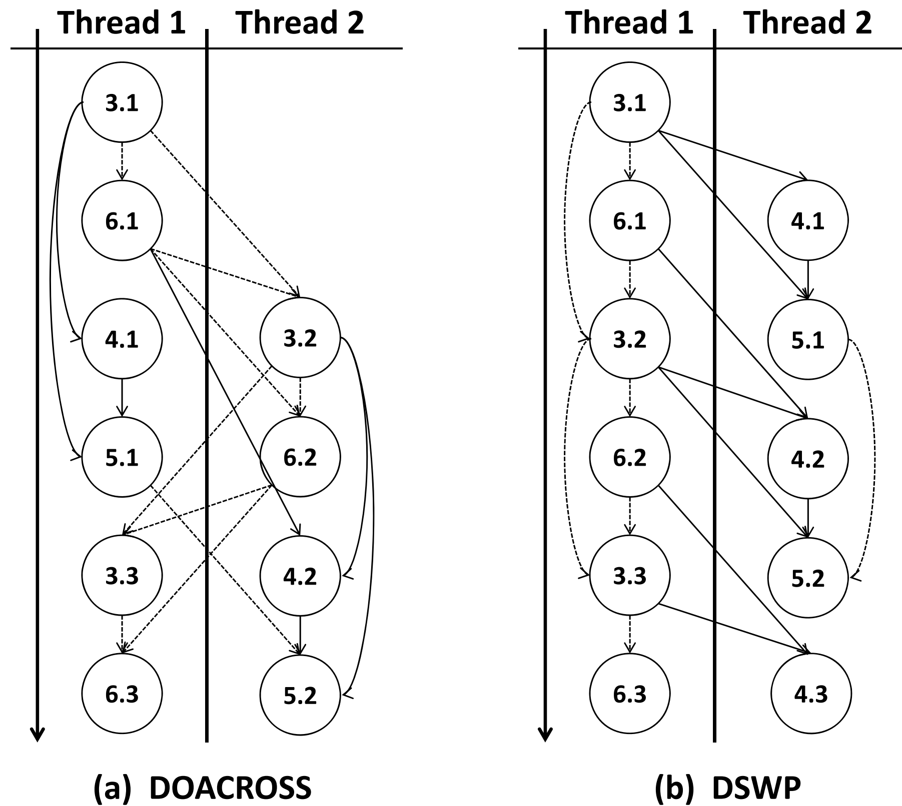
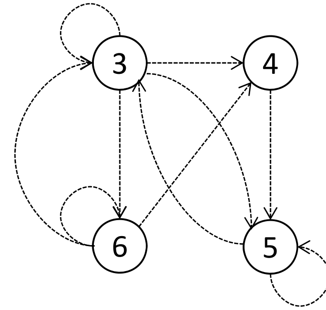



Figure 2.5: Parallelization Execution Plan for DOACROSS and DSWP

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

**(a) Loop cannot benefit from DOACROSS or DSWP**




**(b) PDG**

Figure 2.6: Example Loop which cannot be parallelized by DOACROSS or DSWP

each iteration is dependent on the previous one. The limitations of DOALL, DOANY and LOCALWRITE prevent any of them from parallelizing this loop. Similar to DOALL, DOACROSS parallelizes this loop by assigning sets of loop iterations to different threads. However, since each iteration is dependent on the previous one, later iterations synchronize with earlier ones waiting for the cross-iteration dependences to be satisfied. The code between synchronizations can run in parallel with code executing in other threads (Figure 2.5(a)). Using the same loop, we demonstrate DSWP technique in Figure 2.5(b). Unlike DOALL and DOACROSS parallelization, DSWP does not allocate entire loop iterations to threads. Instead, each thread executes a portion of all loop iterations. The pieces are selected such that the threads form a pipeline. In Figure 2.5(b), thread `1` is responsible for executing statements `3` and `6` for all iterations of the loop, and thread `2` is responsible for executing statements `4` and `5` for all iterations of the loop. Since statement `4` and `5` do not feed statements `3` and `6`, all cross-thread dependences flow from thread `1` to thread `2` forming a thread pipeline. Although neither techniques is limited by the type of dependences, the resulting parallel programs' performance highly rely on the quality of the static analysis. For DOACROSS, excessive dependences lead to frequent synchronization across threads, limiting actual parallelism or even resulting in sequential execution of the program. For DSWP, conservative analysis results lead to unbalanced stage partitions or

small parallel partition, which in turn limit the performance gain. Figure 2.6 shows a loop example which cannot benefit from either techniques. This loop is almost identical to the loop in Figure 2.4(a) except this loop can exit early if the computed cost exceeds a threshold. Since all the loop statements participate in a single dependence cycle (they form a single strongly-connected component in the dependence graph), DSWP is unable to parallelize the loop. Similarly, the dependence height of the longest cycle in the dependence graph is equal to the dependence height of the entire loop iteration rendering DOACROSS ineffective as well.

To overcome the limitation caused by the conservative nature of static analysis, other techniques are proposed to take advantage of runtime information. Among these techniques, some observe dependences at runtime and schedule loop iterations correspondingly. Synchronizations are inserted to respect a dependence between two iterations only if that dependence manifests at runtime. Inspector-Executor (IE) [52, 59, 64] style parallelization techniques are representative of this category of techniques. IE consists of three phases: inspection, scheduling, and execution. A complete dependence graph is built for all iterations during the inspecting process. By topological sorting the dependence graph, each iteration is assigned to a wavefront number for later scheduling. At runtime, iterations with the same wavefront number can execute concurrently while iterations with larger wavefront numbers have to wait till those with smaller wavefront numbers to finish. The applicability of IE is limited by the possibility of constructing an inspector loop at compile time. This inspector loop goes through all memory addresses being accessed in each iteration and determines the dependences between iterations. Since inspector loop is duplicated from the original loop, it is required not to cause any side effect (e.g., update the shared memory). Because of these limitations, example loop in Figure 2.6 cannot be parallelized by IE since without actually updating the values of `node`, we won't know whether the loop will exit.

IE checks dependences at runtime before it enables any concurrent execution. Another group of techniques which also take advantage of runtime information allow concurrent

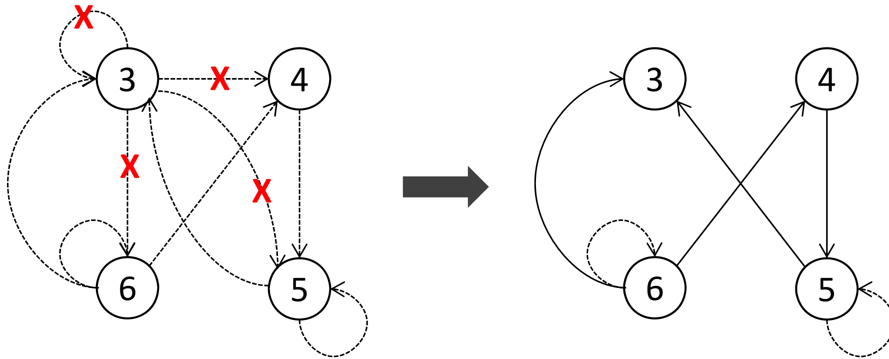



Figure 2.7: PDG after breaking the loop exit control dependence

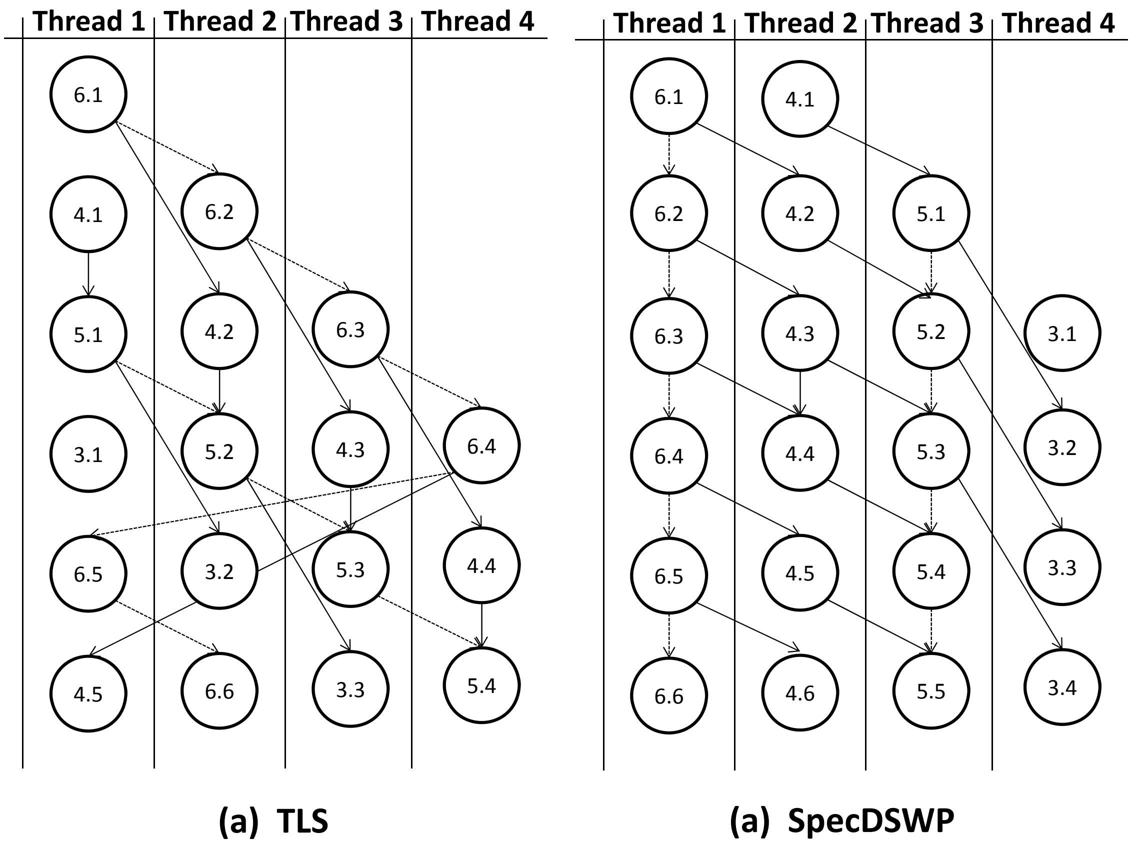



Figure 2.8: TLS and SpecDSWP schedules for the loop shown in Figure 2.6

execution of potentially dependent loop iterations even before they check the dependences. These techniques are referred to as speculative techniques. There have been many proposals for thread-level speculation (TLS) techniques which speculatively break various loop dependences [60, 41, 57]. Once these dependences are broken, effective parallelization becomes possible. Using the same loop in Figure 2.6, if TLS speculatively breaks the loop exit control dependences ( 2.7), assuming that the loop iterates many times, then the execution schedule shown in Figure 2.8(a) is possible. This parallelization offers a speedup of 4 over single threaded execution. Just as in TLS, by speculating the loop exit control dependence, the largest `SCC` is broken allowing SpecDSWP to deliver a speedup of 4 over single-threaded execution (as shown in Figure 2.8(b)).

IE and speculative techniques take advantage of runtime information to more aggressively exploit parallelism. They are able to adapt to dependence patterns manifested at runtime by particular input data sets. Speculative techniques are best when dependences rarely manifest as frequent misspeculation will lead to high recovery cost, which in turn negates the benefit from parallelization. Non-speculative techniques, on the other hand, do not have the recovery cost, and thus are better choices for programs with more frequent manifesting dependences.

## 2.3 Cross-Invocation Parallelization

While intra-invocation parallelization techniques are adequate for programs with few loop invocations, they are not adequate for programs with many loop invocations. All of these techniques parallelize independent loop invocations and use global synchronizations (barriers) to respect the dependences between loop invocations. However, global synchronizations stall all of the threads, forcing them to wait for the last thread to arrive at the synchronization point, causing inefficient utilization of processors and losing the potential cross-invocation parallelism. Studies [44] have shown that in real world applications, syn-

chronizations can contribute as much as 61% to total program runtime. There is an opportunity to improve performance: iterations from different loop invocations can often execute concurrently without violating program semantics. Instead of waiting at synchronization points, threads can execute iterations from subsequent loop invocations. This additional cross-invocation parallelism can improve the processor utilization and help the parallel programs achieve much better scalability.

Prior work has presented some automatic parallelization techniques that exploit cross-invocation parallelism. Ferrero et al. [22] apply affine techniques to aggregate small parallel loops into large ones so that intra-invocation parallelization techniques can be applied directly. Bodin [49] applies communication analysis to analyze the data-flow between each threads. Instead of inserting global synchronizations between invocations, they apply pair-wise synchronization between threads if data-flow exits between them. These more fine-grained synchronization techniques allow for more potential parallelization at runtime since only some of the threads have to stall and wait while the others can proceed across the invocation boundary. Tseng [71] partitions iterations within the same loop invocation so that cross-invocation dependences flow within the same working thread. However, all these techniques share the same limitation as the first group of intra-invocation parallelization techniques: they all rely on static analysis and cannot handle input-dependent irregular program patterns.

Some attempts have been made to exploit parallelism beyond the scope of loop invocation using runtime information. BOP [19] allows programmers to specify all potential concurrent code regions in a sequential program. Each of these concurrent task is treated as a transaction and will be protected from access violations at runtime. TCC [25] requires programmers to annotate the parallel version of program. The programmer can specify the boundary of each transaction and assign each transaction a phase-number to guarantee the correct commit order of each transaction. However, these two techniques both require manual annotation or manual parallelization by programmers. Additionally, these techniques

do not distinguish intra- and inter-invocation dependences, which means they may incur unnecessary overhead at runtime.

Table 2.1 summarizes the related work. All existing cross-invocation parallelization techniques are either limited by the conservative nature of static analysis or require programmers' effort to achieve parallelization. This motives this thesis work to propose novel automatic parallelization techniques which bridge the gap in the existing solution space.

| Technique Name | Synchronization Inserted Statically | Speculative | Runtime System Used | Software Only | Automatic |
|---|---|---|---|---|---|
| **Exploiting Intra-Invocation Parallelization** | | | | | |
| DOALL [1] | None | × | None | ✓ | ✓ |
| DOANY [54, 74] | Lock | × | None | ✓ | ✓ |
| DOACROSS [15] | Thread-wise Syncs | × | None | ✓ | ✓ |
| DSWP [50] | Produce and Consume | × | None | ✓ | ✓ |
| IE [52, 59, 64] | None | × | Shadow Array | ✓ | ✓ |
| LRPD [61] | None | ✓ | Shadow Array | ✓ | ✓ |
| STMLite [41] | None | ✓ | STM | ✓ | ✓ |
| SMTX [57] | None | ✓ | SMTX | ✓ | × |
| **Exploiting Cross-Invocation Parallelization** | | | | | |
| Ferrero et al. [22] | None | × | None | ✓ | ✓ |
| Tseng [71] | None | × | None | ✓ | ✓ |
| TCC [25] | None | ✓ | Hardware TM | × | × |
| BOP [19] | None | ✓ | Software TM | ✓ | × |
| DOMORE (this thesis work) | None | × | Shadow Array | ✓ | ✓ |
| SPECCROSS (this thesis work) | None | ✓ | Software Speculative Barrier | ✓ | ✓ |

Table 2.1: Compared to related work, DOMORE and SPECCROSS improve the performance by automatically exploiting cross-invocation parallelism. Both take advantage of runtime information to synchronize threads only when necessary. And neither requires any special hardware support.

# Chapter 3

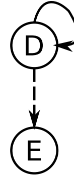
# Non-Speculatively Exploiting Cross-Invocation Parallelism Using Runtime Information

This chapter discusses details about DOMORE, the first non-speculative automatic parallelization technique to capture dynamic cross-invocation parallelism. Unlike existing techniques, DOMORE gathers cross-invocation dependence information at runtime. Even for programs with irregular dependence patterns, DOMORE precisely synchronizes iterations which depend on each other and allows iterations without dependences to execute concurrently. As a result, DOMORE is able to enable more cross-invocation parallelization and achieves more scalable performance.

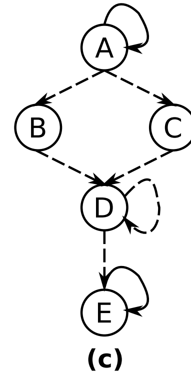
The automatic compiler implementation of DOMORE in LLVM [34] provides significant performance gains over both sequential code and parallel code with barriers. Evaluation on six benchmark programs shows a loop speedup of 2.1× over codes without cross-invocation parallelization and 3.2× over the original sequential performance on 24 cores.

```
A. for (i = 0; i < N; i++) {
B.   start = A[i];
C.   end = B[i];
D.   for (j = start; j < end; j++) {
E.     update (&C[j]);
     }
   }
```

**(a)**




**(b)**




**(c)**



Figure 3.1: Example program: (a) Simplified code for a nested loop in CG (b) PDG for inner loop. The dependence pattern allows DOALL parallelization. (c) PDG for outer loop. Cross-iteration dependence deriving from E to itself has manifest rate 72.4%.

## 3.1 Motivation and Overview

To motivate the DOMORE technique, we present a running example using the program CG from the NAS suite [47]. Figure 3.1 (a) shows a simplified version of CG's performance dominating loop nest. The outer loop computes the loop bounds of the inner loop, and the inner loop calls the `update` function, updating values in array `C`. For the outer loop, aside from induction variables, the only cross-iteration dependence is between calls to the `update` function. Profiling reveals that this dependence manifests across 72.4% of outer loop iterations. The inner loop has no cross-iteration dependence since no two iterations in the same invocation update the same element in array `C`.

The `update` dependence prevents DOALL parallelization [1] of the outer loop. Spec-DOALL [61] can parallelize outer loop by speculating that the `update` dependence does not occur. However, speculating the outer loop dependence is not profitable, since the `update` dependence frequently manifests across outer loop iterations. As a result, DOALL will parallelize the inner loop and insert barrier synchronizations between inner loop invocations to ensure the dependence is respected between invocations.

Figure 3.2(a) shows the execution plan for a DOALL parallelization of CG. Iterations in the same inner loop invocation execute in parallel. After each inner loop invocation, threads synchronize at the barrier. Typically, threads do not reach barriers at the same time

for a variety of reasons. For instance, each thread may be assigned different number of tasks and the execution time of each task may vary. Threads that finish the inner loop early may not execute past the barrier, resulting in very poor scalability. Experimental results in Figure 3.3 show that performance is worse than the sequential version and worsens as the number of threads increases.

Figure 3.2(b) shows the execution plan after DOMORE's partitioning phase (Section 3.3.1). The first thread executes code in the outer loop (statement A to D) and serves as the scheduler. The other threads execute `update` code in the inner loop concurrently and serve as workers. Overlapping the execution of scheduler and worker threads improves the performance, however, without enabling cross-invocation parallelism, clock cycles are still wasted at the synchronization points.

Figure 3.2(c) shows the execution plan after the DOMORE transformation completes and enables cross-invocation parallelization. The scheduler sends synchronization information to the worker threads and worker threads only stall when a dynamic dependence is detected. In the example, most of CG's iterations are independent and may run concurrently without synchronization. However, iteration 1.5 (i.e. when $i$=1, $j$=5) updates memory locations which are later accessed by iteration 2.2. At runtime, the scheduler discovers this dependence and signals thread one to wait for iteration 1.5. After synchronization, thread one proceeds to iteration 2.2. As shown in Figure 3.3, DOMORE enables scalable loop speedup for CG up to 24 threads on an 24-core machine.

Figure 3.4 shows a high-level overview of the DOMORE transformation and runtime synchronization scheme. DOMORE accepts a sequential program as input and targets hot loop nests within the program. At compile-time, DOMORE first partitions the outer loop into a scheduler thread and several worker threads (Section 3.3.1). The scheduler thread contains the sequential outer loop while worker threads contain the parallel inner loop. Based on the parallelization technique used to parallelize the inner loop, the code generation algorithm generates a multi-threaded program with cross-invocation parallelization

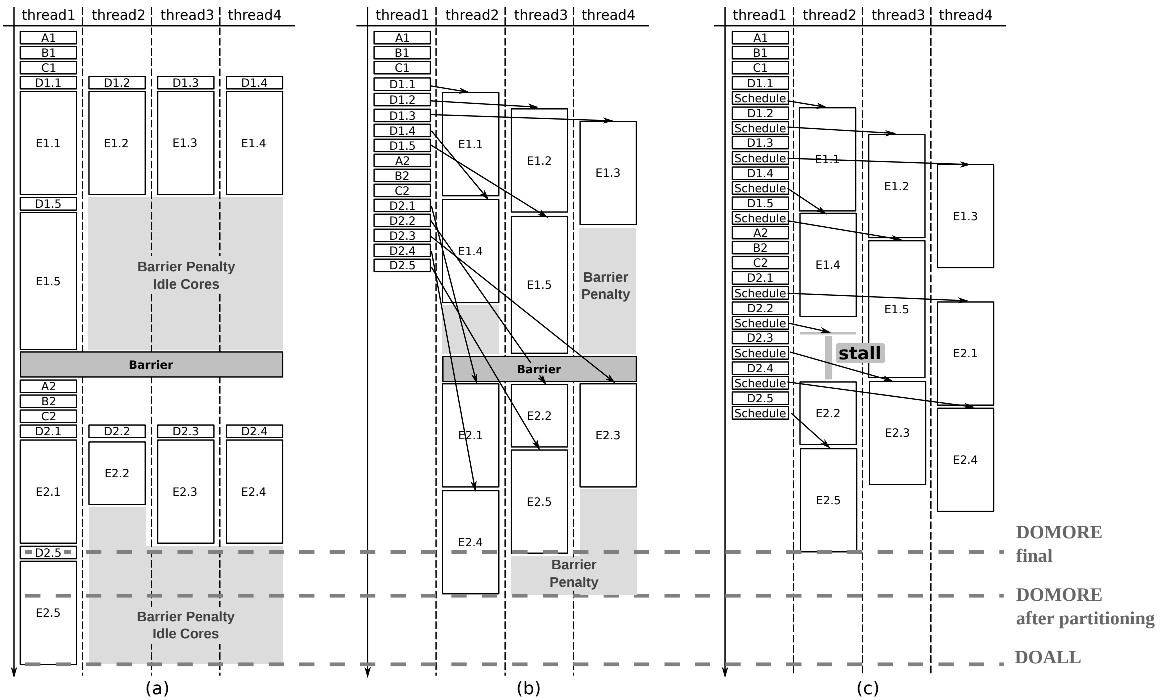



Figure 3.2: Comparison of performance with and without cross-invocation parallelization : (a) DOALL is applied to the inner loop. Frequent barrier synchronization occurs between the boundary of the inner and outer loops. (b) After the partitioning phase, DOMORE has partitioned the code without inserting the runtime engine. A scheduler and three workers execute concurrently, but worker threads still synchronize after each invocation. (c) DOMORE finalizes by inserting the runtime engine to exploit cross-invocation parallelism. Assuming iteration 2 from invocation 2 (2.2) depends on iteration 5 from invocation 1 (1.5). Scheduler detects the dependence and synchronizes those two iterations.

(Section 3.3.5). At runtime, the scheduler thread checks for dynamic dependences, schedules inner loop iterations, and forwards synchronization conditions to worker threads (Section 4.2). Worker threads use the synchronization conditions to determine when they are ready to execute.

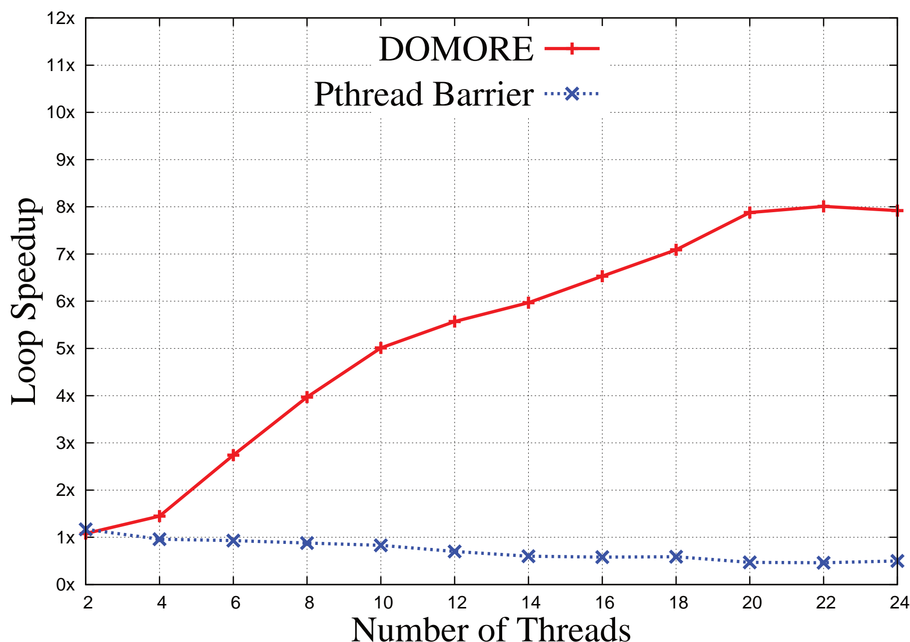



Figure 3.3: Performance improvement of CG with and without DOMORE.

## 3.2 Runtime Synchronization

DOMORE's runtime synchronization system consists of three parts: detection of dependences at runtime, generation of synchronization conditions, and synchronization of iterations across threads. The pseudo-code for the scheduler and worker threads appear in Algorithms 1 and 2.

### 3.2.1 Detecting Dependences

DOMORE's scheduler thread detects dependences which manifest at runtime. Shadow memory is employed for determining memory dependences. Each entry in shadow memory contains a tuple consisting of a thread ID (`tid`) and an iteration number (`iterNum`). For each iteration, the scheduler determines which memory addresses the worker will access using `computeAddr` function. The `computeAddr` function collects these addresses by redundantly executing related instructions duplicated from the inner loop. Details of automatic generation of `computeAddr` can be found in Section 3.3.4. The scheduler

maps each of these address to a shadow memory entry and updates that entry to indicate that the most recent access to the respective memory location is by worker thread `tid` in iteration `iterNum`. When an address is accessed by two different threads, the scheduler synchronizes the affected threads.

Although the use of shadow memory increases memory overhead, our experiments demonstrate it is an efficient method for detecting dynamic dependences. However, a more space efficient conflict detecting scheme can also be used by DOMORE. For example, Mehrara et al. [41] propose a lightweight memory signature scheme to detect memory conflicts. The best time-space trade-off depends on end-user requirements.

### 3.2.2 Generating Synchronization Conditions

If two iterations dynamically depend on each other, worker threads assigned to execute them must synchronize. This requires collaboration between scheduler and worker threads.

The scheduler constructs synchronization conditions and sends them to the scheduled worker thread. A synchronization condition is a tuple also consisting of a thread ID (`depId`) and an iteration number (`depIterNum`). A synchronization condition tells a worker thread to wait for another worker (`depId`) to finish a particular iteration (`depIterNum`).

To indicate that a worker thread is ready to execute a particular iteration, the scheduling thread sends the worker thread a special tuple. The first element is a token (`NO_SYNC`) indicating no further synchronization is necessary to execute the iteration specified by the second element (`iterNum`).

Suppose a dependence is detected while scheduling iteration `i` to worker thread `T1`. `T1` accesses the memory location `ADDR` in iteration `i`. Shadow array (`shadow[ADDR]`) records that the same memory location is most recently accessed by worker thread `T2` in iteration `j`. The scheduler thread will send `(T2, j)` to thread `T1`. When the scheduler thread finds no additional dependences, it will send `(NO_SYNC, i)` to thread `T1`.

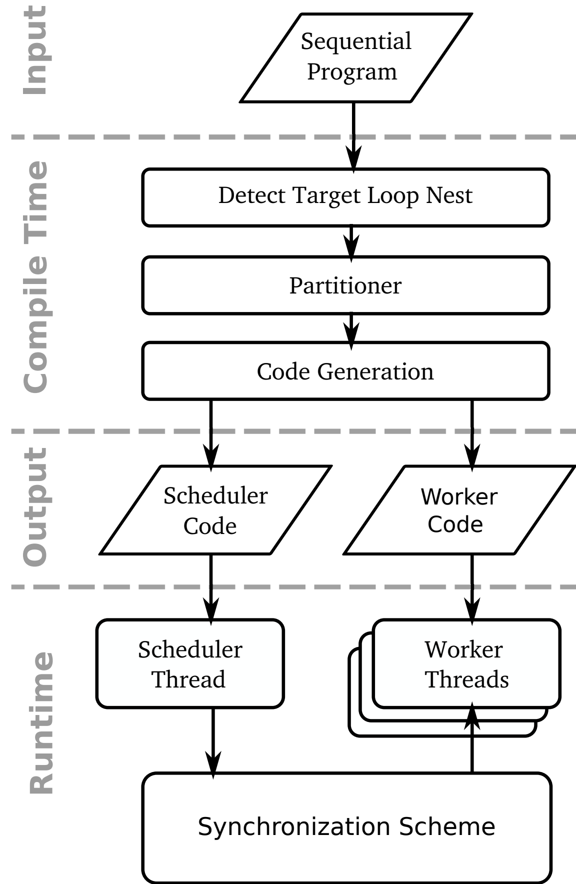



Figure 3.4: Overview of DOMORE compile-time transformation and runtime synchronization

### 3.2.3 Synchronizing Iterations

Workers receive synchronization conditions and coordinate with each other to respect dynamic dependences. A status array is used to assist this: `latestFinished` records the latest iteration finished by each thread. A worker thread waiting on synchronization condition `(depId, depIterNum)` will stall until `latestFinished[depId]` $\geq$ `depIterNum`. After each worker thread finishes an iteration, it needs to update its status in `latestFinished` to allow threads waiting on it to continue executing.

Synchronization conditions are forwarded using `produce` and `consume` primitives provided by a lock-free queue design [30], which provides an efficient way to communicate information between scheduler and worker threads.

**Algorithm 1**: Pseudo-code for scheduler synchronization

```
Input: iterNum : global iteration number
addrSet ← computeAddr(iterNum)
tid ← schedule(iterNum, addrSet)
foreach addr ∈ addrSet do
    < depTid, depIterNum > ← shadow[addr]
    if depIterNum ≠ −1 then
        if depTid ≠ tid then
            produce(tid, < depTid, depIterNum >)
    shadow[addr] ← < tid, iterNum >
produce(tid, < NO_SYNC, iterNum >)
```

**Algorithm 2**: Pseudo-code for worker

```
< depTid, depIterNum > ← consume()
while depTid ≠ NO_SYNC do
    while latestFinished[depTid] < depIterNum do
        sleep()
    < depTid, depIterNum > ← consume()
doWork(depIterNum)
latestFinished[getTid()] ← depIterNum
```

### 3.2.4 Walkthrough Example

The program CG illustrates DOMORE's synchronization scheme. Figure 3.5(a) shows the access pattern (value *j*) in each iteration for two invocations. Iterations are scheduled to two worker threads in round-robin order. Figure 3.5(b) shows the change of the helper data structures throughout the execution.

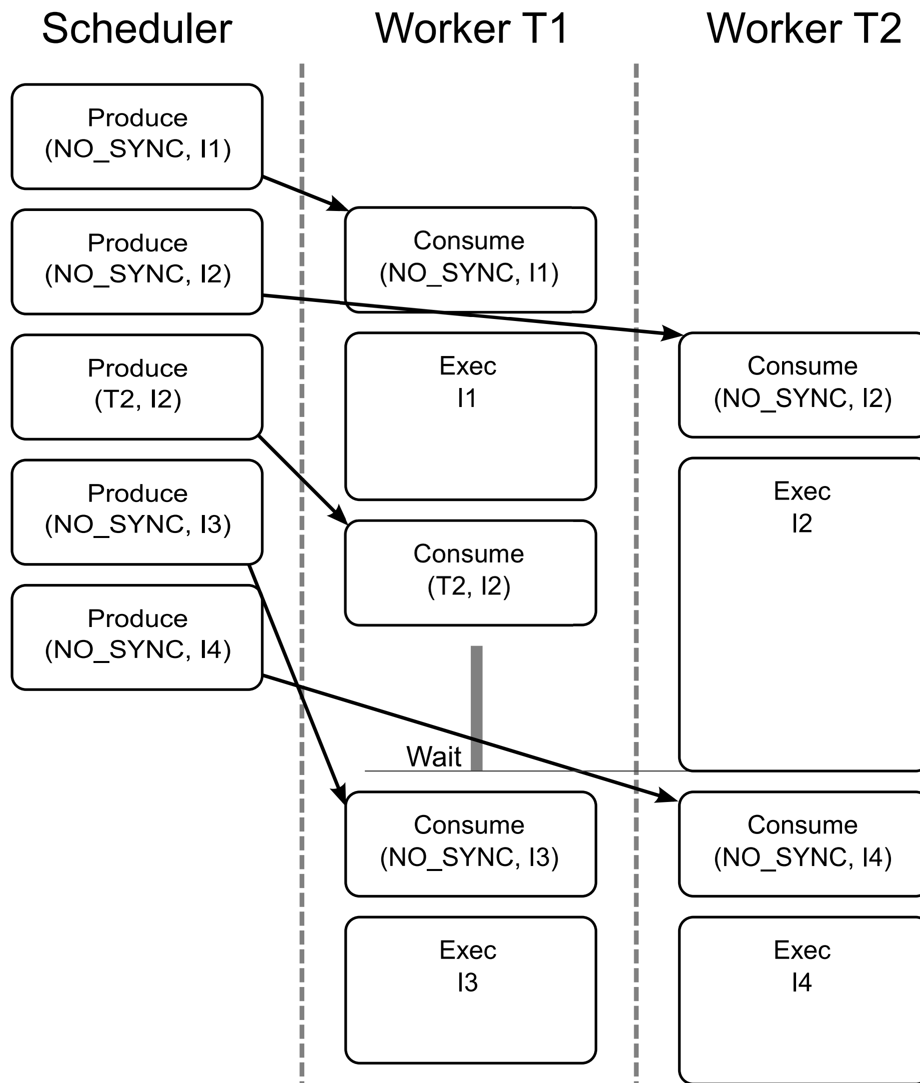
Iteration I1 accesses array element A1, the scheduler finds `shadow[A1] = <⊥,⊥>` meaning no dependence exists. It constructs a synchronization condition `(NO_SYNC,I1)` and produces it to worker thread T1. It then updates `shadow[A1]` to be `<T1,I1>`, implying thread T1 has accessed array element A1 in iteration I1. Worker thread T1 consumes the condition and executes iteration I1 without waiting. After it finishes, it updates `latestFinished[T1]` to be I1. Iteration I2 accesses array element A3 and no dependence is detected. A synchronization condition `(NO_SYNC,I2)` is produced to worker thread T2, and worker thread T2 consumes the condition and executes itera-

tion I2 immediately. Iteration I1 in the second invocation accesses element A3 again. Since shadow[A3] = <T2,I2>, a dependence is detected. So the scheduler produces (T2,I2) and (NO_SYNC,I3) to worker thread T1. Worker thread T1 then waits for worker thread T2 to finish iteration I2 (wait until latestFinished[T2] $\geq$ I2). Worker thread T1 then consumes the (NO_SYNC,I3) and begins execution of iteration I3.

Using this synchronization scheme, instead of stalling both threads to wait for first invocation to finish, only thread T1 needs to synchronize while thread T2 can move on to execute iterations from the second invocation.

| Original | | | Generated | | |
|---|---|---|---|---|---|
| Invoc. | Iter. | Access | Sched. | Combined Iter. | `shadow` |
| - | - | - | - | initialize | $\langle\bot,\bot\rangle,\langle\bot,\bot\rangle,\langle\bot,\bot\rangle,\langle\bot,\bot\rangle$ |
| 1 | 1 | A1 | T1 | I1 | $\langle\bot,\bot\rangle,\langle \text{T1},\text{I1}\rangle,\langle\bot,\bot\rangle,\langle\bot,\bot\rangle$ |
| 1 | 2 | A3 | T2 | I2 | $\langle\bot,\bot\rangle,\langle \text{T1},\text{I1}\rangle,\langle\bot,\bot\rangle,\langle \text{T2},\text{I2}\rangle$ |
| 2 | 1 | A3 | T1 | I3 | $\langle\bot,\bot\rangle,\langle \text{T1},\text{I1}\rangle,\langle\bot,\bot\rangle,\langle \text{T1},\text{I3}\rangle$ |
| 2 | 2 | A2 | T2 | I4 | $\langle\bot,\bot\rangle,\langle \text{T1},\text{I1}\rangle,\langle \text{T2},\text{I4}\rangle,\langle \text{T1},\text{I3}\rangle$ |

(a)




(b)

Figure 3.5: Scheduler scheme running example: (a) Table showing original invocation/iteration, array element accessed in iteration, thread the iteration is scheduled to, combined iteration number, and helper data structure values (b) Execution of the example.

## 3.3 Compiler Implementation

The DOMORE compiler generates scalable parallel programs by exploiting both intra-invocation and cross-invocation parallelism. DOMORE first detects a candidate code region which contains a large number of loop invocations. DOMORE currently targets loop nest whose outer loop cannot be efficiently parallelized because of frequent runtime dependences, and whose inner loop is invoked many times and can be parallelized easily. For each candidate loop nest, DOMORE generates parallel code for the scheduler and worker threads. This section uses the example loop from CG (Figure 3.1) to demonstrate each step of the code transformation. Figure 3.6(a) gives the pseudo IR code of the CG example.

### 3.3.1 Partitioning Scheduler and Worker

DOMORE allows threads to execute iterations from consecutive parallel invocations. However, two parallel invocations do not necessarily execute consecutively; typically a sequential region exists between them. In CG's loop, statement A, B and C belong to the sequential region. After removing the barriers, threads must execute these sequential regions before starting the iterations from next parallel invocation.

DOMORE executes the sequential code in the scheduler thread. This provides a general solution to handle the sequential code enclosed by the outer loop. After partitioning, only the scheduler thread executes the code. There is no redundant computation and no need for special handling of side-effecting operations. If a data flow dependence exists between the scheduler and worker threads, the value can be forwarded to worker threads by the same queues used to communicate synchronization conditions.

The rule for partitioning code into worker and scheduler threads is straightforward. The inner loop body is partitioned into two sections. The loop-traversal instructions belong to the scheduler thread, and the inner loop body belongs to the worker thread. Instructions outside the inner loop but enclosed by the outer loop are treated as sequential code and thus

belong to the scheduler.

To decouple the execution of the scheduler and worker threads for better latency tolerance, they should communicate in a pipelined manner. Values are forwarded in one direction, from the scheduler thread to the worker threads.

The initial partition may not satisfy this pipeline requirement. To address this problem, DOMORE first builds a program dependence graph (PDG) for the target loop nest (Figure 3.6(b)), including both cross-iteration and cross-invocation dependences for the inner loop. Then DOMORE groups the PDG nodes into strongly connected components (SCC) and creates a $DAG_{SCC}$ (Figure 3.6(c)) which is a directed acyclic graph for those SCCs.

DOMORE goes through each SCC in $DAG_{SCC}$: (1) If an SCC contains any instruction that has been scheduled to the scheduler, all instructions in that SCC should be scheduled to the scheduler partition. Otherwise, all instructions in that SCC are scheduled to the worker partition; (2) If an SCC belonging to the worker partition causes a backedge towards any SCC belonging to the scheduler partition, that SCC should be re-partitioned tothe scheduler. Step (2) is repeated until both partitions converge.

### 3.3.2 Generating Scheduler and Worker Functions

After computing the instruction partition for scheduler and worker, DOMORE generates code for scheduler and worker threads. Multi-Threaded Code Generation algorithm (MTCG) used by DOMORE builds upon the algorithm proposed in [50]. The major difference is that [50] can only assign a whole inner loop invocation to one thread while DOMORE can distribute iterations in the same invocation to different worker threads. The following description about DOMORE's MTCG highlights the differences:

1. **1. Compute the set of relevant basic blocks (BBs) for scheduler ($T_s$) and worker ($T_w$) threads.** According to algorithm in [50]: A basic block is relevant to a thread $T_i$ if it contains either: (a) an instruction scheduled to $T_i$; or (b) an instruction on which any of $T_i$'s instruction depends; or (c) a branch instruction that controls a relevant BB

to $T_i$. DOMORE's MTCG follows these three rules, and requires two more rules: (d) a BB is relevant to $T_w$ only if it belongs to the original inner loop; and (e) inner loop header is always relevant to both $T_s$ and $T_w$. Rule (d) simplifies the control flow of code generated for $T_w$. However, since `produce` and `consume` instructions are placed at the point where dependent values are defined, worker thread may not contain the corresponding BB because of rule (d). Rule (e) guarantees that any value that is defined in BBs which are not duplicated in $T_w$ can be communicated at the beginning of the duplicated inner loop headers.

2. **2. Create the BBs for each partition.** Place instructions assigned to the partition in the corresponding BB, maintaining their original relative order within the BB. Add a loop preheader BB and a loop return BB to $T_w$.

3. **3. Fix branch targets.** In cases where the original target does not have a corresponding BB in the same thread, the new target is set to be the BB corresponding to the closest relevant post-dominator BB of the original target. Insert a sync BB to $T_w$, which serves as the closest post-dominator BB for BBs which do not have a relevant post-dominator BB in $T_w$. Branch the sync BB in $T_w$ to the loop header.

4. **4. Insert `produce` and `consume` instructions.** For Loop flow dependences, `produce` and `consume` instructions are inserted in the BB where the value is defined, if that BB is duplicated in both threads. Since inner loop live-in values are used but not defined inside the inner loop, the respective BB are not duplicated in $T_w$. To reduce the amount of communications, live-in values which are outer loop invariants are communicated at the end of the inner loop preheader. And the other live-ins will be communicated at the beginning of inner loop header. According to the partition rules, since instructions generating inner loop live-outs to the outer loop are partitioned to the scheduler thread, DOMORE does not need to handle those live-out values. Finally, a timestamp is communicated at the beginning of the inner loop

header. This timestamp value gives a global order for iterations from all invocations and will be used for scheduling and synchronizing iterations.

5. **5. Finalize the communication.** To control when each worker thread should return, an END_TOKEN is broadcasted when exiting the outer loop in $T_s$. That value will be captured by the first consume instruction in $T_w$'s duplicated loop header. Two instructions are inserted to decide when to return from $T_w$: (1) a comparison instruction to check whether that value is an END_TOKEN; (2) a branch instruction targeting the return BB if the comparison instruction generates true value.

Up to this point, DOMORE has generated the initial code for scheduler thread and worker thread (Figure 3.6(d) and (e)). Later steps generate scheduling code, computeAddr code which will be inserted into the scheduler function and workerSync code which will be inserted into the worker function.

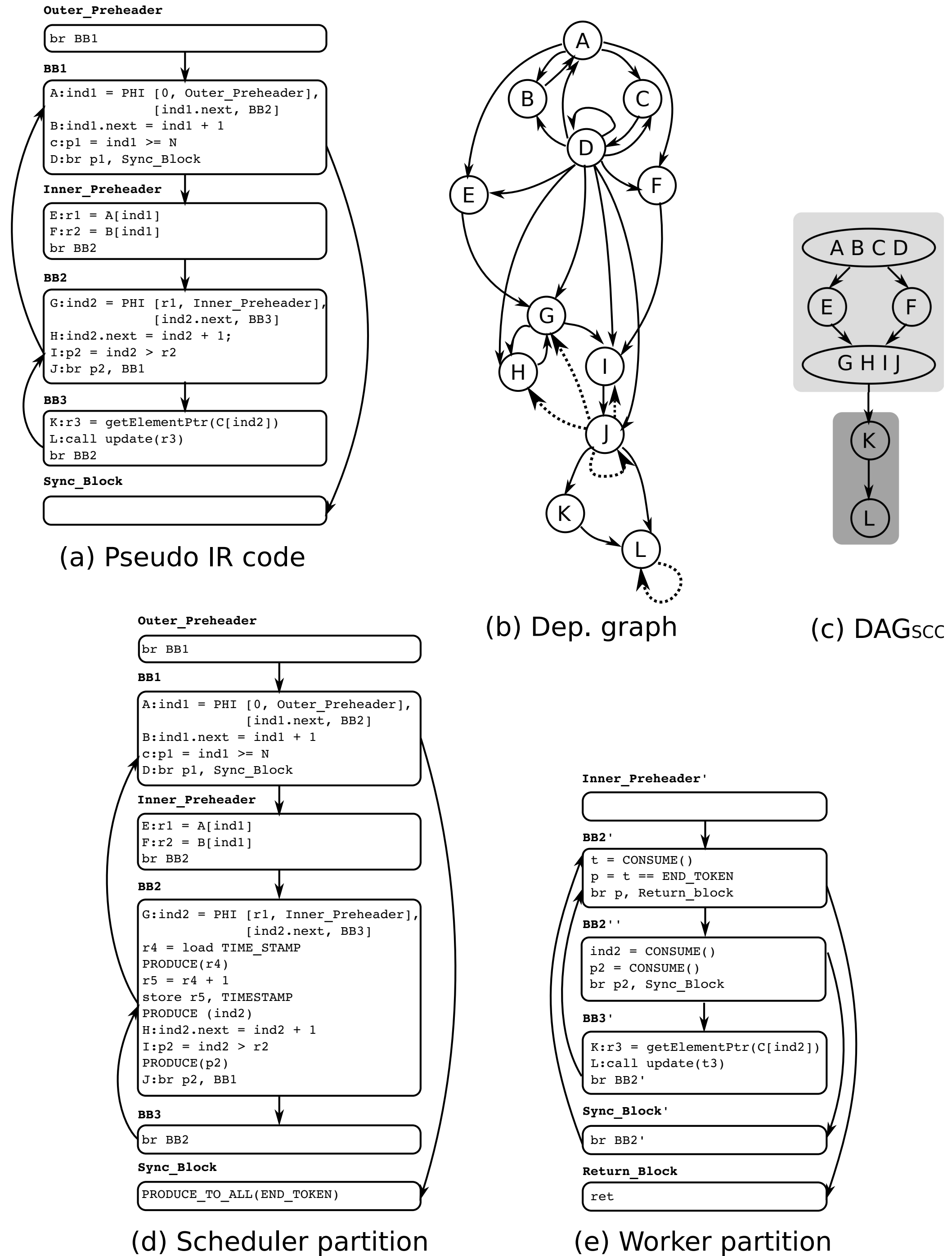



Figure 3.6: Running example for DOMORE code generation: (a) Pseudo IR for CG code; (b) PDG for example code. Dashed lines represent cross-iteration and cross-invocation dependences for inner loop. Solid lines represent other dependences between inner loop instructions and outer loop instructions. (c) $DAG_{SCC}$ for example code. $DAG_{SCC}$ nodes are partitioned into scheduler and worker threads. (d) and (e) are code generated by DOMORE MTCG algorithm (3.3.2).

### 3.3.3 Scheduling Iterations

DOMORE currently supports two scheduling strategies, round-robin and memory partition based scheduling. Round-robin is used by many parallelization techniques. Memory partitioning (LOCALWRITE [26]) divides the memory space into disjoint chunks and assigns each chunk to a different worker thread, forming a one-to-one mapping. Iterations are scheduled to threads that own the memory locations being touched by that iteration. If multiple threads own the memory locations, that iteration is scheduled to all of them. LOCALWRITE guarantees that each thread only updates its own memory space. DOMORE allows for the easy integration of other "smarter" scheduling techniques. Integration of a work stealing scheduler similar to Cilk [8] is planned as future work.

### 3.3.4 Generating the computeAddr function

The scheduler thread uses the `computeAddr` function to determine which addresses will be accessed by worker threads. DOMORE automatically generates the `computeAddr` function from the worker thread function using Algorithm 3. The algorithm takes as input the worker thread's IR in SSA form and a program dependence graph (PDG) describing the dependences in the original loop nest. The compiler uses the PDG to find all instructions with memory dependences across the inner loop iterations or invocations. These instructions will consist of loads and stores. In the worker thread, program slicing [73] is performed to create the set of instructions required to generate the address of the memory being accessed. Presently, the DOMORE transformation does not handle `computeAddr` functions with side-effects. If program slicing duplicates instructions with side-effects, the DOMORE transformation aborts. After the transformation, a performance guard compares the weights of the `computeAddr` function and the original worker thread. If the `computeAddr` function is too heavy relative to the original worker, the scheduler would be a bottleneck for the parallel execution, so the performance guard reports DOMORE is inapplicable.

**Algorithm 3**: Pseudo-code for generating the computeAddr function from the worker function

**Input**: worker : worker function IR
**Input**: pdg : program dependence graph
**Output**: computeAddr : computeAddr function IR
depInsts ← getCrossMemDepInsts(pdg)
depAddr ← getMemOperands(depInsts)
computeAddr ← reverseProgramSlice(worker, depAddr)

**Algorithm 4**: Final Code Generation

**Input**: program : original program IR
**Input**: partition : Partition of scheduler and worker code
**Input**: parallelPlan : parallelization plan for inner loop
**Input**: pdg : program dependence graph
**Output**: multi − threaded scheduler and worker program
scheduler, worker ← MTCG(program, partition)
scheduler ← generateSchedule(parallelPlan)
computeAddr ← generateComputeAddr(worker, pdg)
scheduler ← generateSchedulerSync()
worker ← generateWorkerSync()

### 3.3.5 Putting It Together

Algorithm 4 ties together all the pieces of DOMORE's code-generation. The major steps in the transformation are:

1. The Multi-Threaded Code Generation algorithm (MTCG) discussed in Section 3.3.2 generates the initial scheduler and worker threads based on the partition from Section 3.3.1.

2. The appropriate `schedule` function (Section 3.3.3) is inserted into the scheduler based upon the parallelization plan for the inner loop.

3. Create and insert the `computeAddr` (Algorithm 3) `schedulerSync` (Algorithm 1), `workerSync` (Algorithm 2), functions into the appropriate thread to handle dependence checking and synchronizing.

Figure 3.7 shows the final code generated for CG.

**Scheduler Function**

```
1   void scheduler () {
2     iternum = 0;
3     for (i = 0; i < N; i++) {
4       start = A[i];
5       end = B[i];
6       for (j = start; j < end; j++) {
7         addr_set = computeAddr(iternum);
8         tid = schedule(iternum, addr_set);
9         tid_queue = getQueue(tid);
10        schedulerSync(iternum, tid, tid_queue, addr_set);
11        produce(&C[j], tid_queue);
12        iternum++;
        }
      }
    }
13  produce_to_all(END_TOKEN);
```

**SchedulerSync Function**

```
1   void schedulerSync(iternum, tid, queue, addr_set) {
2     while (addr = get_next(addr_set)) {
3       depTid = getTid(shadow[addr]);
4       depIterNum = getIterNum(shadow[addr]);
5       if (depTid != tid && depIterNum != -1) {
6         produce(depTid, queue);
7         produce(depIterNum, queue);
        }
8       shadow[addr] = (tid, iternum);
      }
9     produce(NO_SYNC, queue);
10    produce(iternum, queue);
    }
```

**Worker Function**

```
1   void worker() {
2     while (1) {
3       depTid = consume();
4       if (depTid == END_TOKEN)
5         return;
6       if (depTID == NO_SYNC) {
7         doWork();
        }
8       else
9         workerSync(depTid);
      }
    }
```

**doWork Function**

```
1   void doWork() {
2     iternum = consume();
3     tid = getTid();
4     addr = consume();
5     update(addr);
6     latestFinished[tid] = iternum;
    }
```

**workerSync Function**

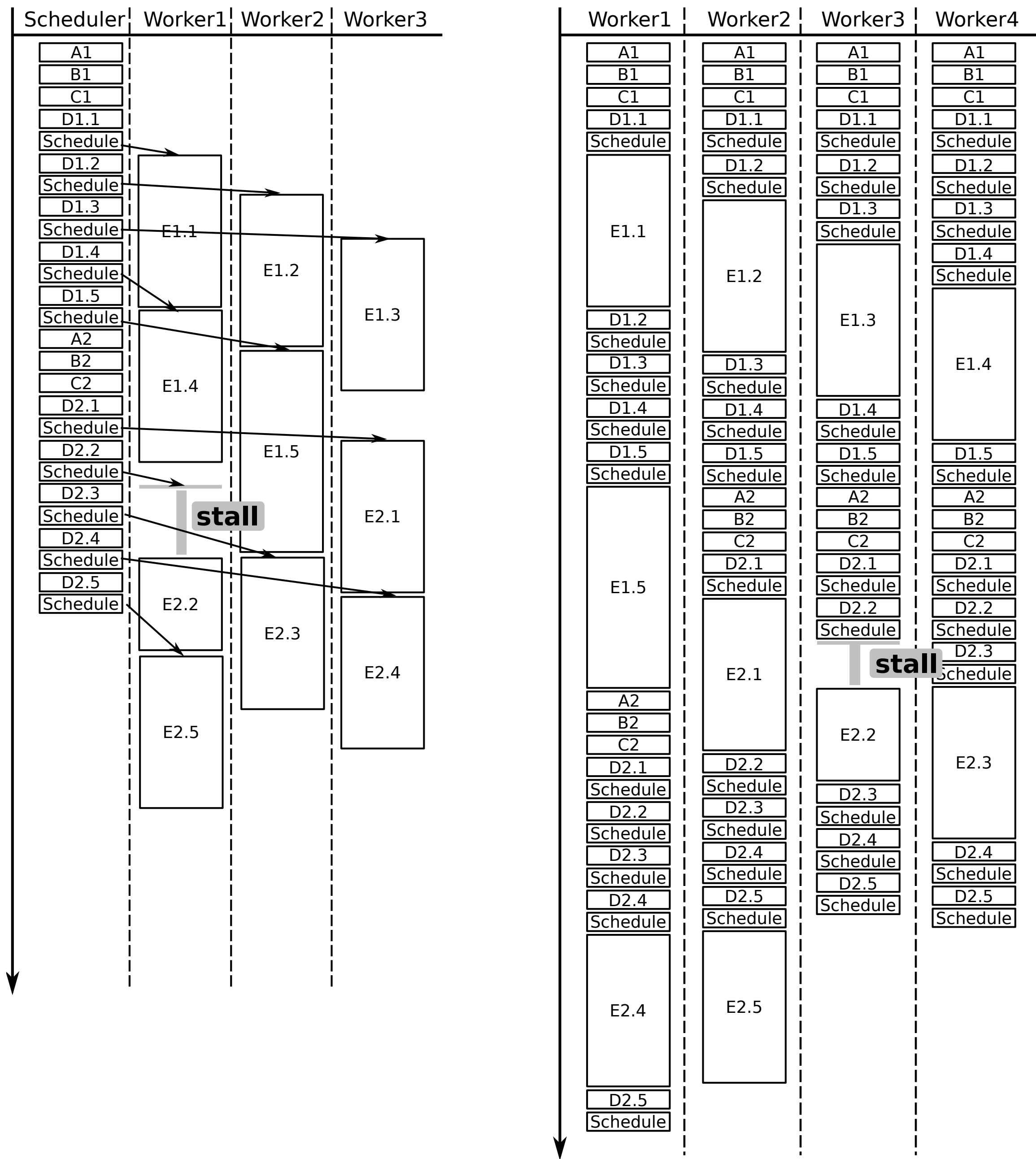
```
1   void workerSync(depTid) {
2     iternum = consume();
3     while (latestFinished[depTid] < iternum)
4       sleep();
    }
```

Figure 3.7: Generated code for example loop in CG. Non-highlighted code represents initial code for scheduler and worker functions generated by DOMORE's MTCG (Section 3.3.2). Code in grey is generated in later steps for iteration scheduling and synchronization.

## 3.4 Enable DOMORE in SPECCROSS

DOMORE transformation partitions the sequential program into a scheduler thread and multiple worker threads. The scheduler thread executes the sequential code region, computes the dependences between inner loop iterations and schedules iterations correspondingly. This design provides a general solution to handle the sequential code enclosed by the outer loop. There is no redundant computation and no need for special handling of side-effecting operations. However, this design prohibits DOMORE to be integrated into SPECCROSS framework which will be introduced in the next chapter. As a result, we trade the benefits from having a separate scheduler thread for the applicability of DOMORE parallelization in SPECCROSS framework by duplicating the scheduler code to all worker threads. Figure 3.8 demonstrates the parallel execution plan after the duplication. Only worker threads are spawned for the parallel execution. Each worker thread computes dependences and schedules iterations independently. Each worker only executes the iterations scheduled to it, but it executes all of the scheduler code to keep a record of the iteration dependences.

Figure 3.9 shows the new generated code. Compared to the original code in Figure 3.7, the major differences include: (1) Only worker threads are spawned and each of them starts by executing the `Scheduler` function. If an iteration is scheduled to the executing worker thread, the `worker` function is invoked to do the actual work for that iteration. (2) Every worker thread executes `computeAddr`, `schedule` and `schedulerSync` functions independently. To avoid access conflicts, each worker thread has its own shadow memory and only updates that shadow memory. (3) Synchronization conditions are still produced to and consumed from the communication queues; while value dependences between the original scheduler and the worker threads are passed on as function parameters instead.




(a) Before duplication

(b) Before duplication

Figure 3.8: Execution plan for DOMORE before and after duplicating scheduler code to worker threads.

**Scheduler Function**

```
1   void scheduler () {
2     iternum = 0;
3     threadID = getTid();
4     for (i = 0; i < N; i++) {
5       start = A[i];
6       end = B[i];
7       for (j = start; j < end; j++) {
8         addr_set = computeAddr(iternum);
9         tid = schedule(iternum, addr_set);
10        tid_queue = getQueue(tid);
11        schedulerSync(iternum, tid, tid_queue, addr_set);
12        if (threadID == tid)
13          worker(iternum, &C[j]);
14        iternum++;
      }
    }
  }
```

**SchedulerSync Function**

```
1   void schedulerSync(iternum, tid, queue, addr_set) {
2    threadID = getTid();
3    while (addr = get_next(addr_set)) {
4      depTid = getTid(shadow[threadID][addr]);
5      depIterNum = getIterNum(shadow[threadID][addr]);
6      if (depTid != tid && depIterNum != -1 && threadID == tid) {
7        produce(depTid, queue);
8        produce(depIterNum, queue);
       }
9      shadow[threadID][addr] = (tid, iternum);
     }
10   if (threadID == tid) {
11     produce(NO_SYNC, queue);
     }
  }
```

**Worker Function**

```
1   void worker(iternum, addr) {
2     while (1) {
3       depTid = consume();
4       if (depTID == NO_SYNC) {
5         doWork(iternum, addr);
6         return;
        }
7       else
8         workerSync(depTid);
      }
  }
```

**doWork Function**

```
1   void doWork(iternum, addr) {
2     tid = getTid();
3     update(addr);
4     latestFinished[tid] = iternum;
  }
```

**workerSync Function**

```
1   void workerSync(depTid) {
2     iternum = consume();
3     while (latestFinished[depTid] < iternum)
4       sleep();
  }
```

Figure 3.9: Optimization for DOMORE technique: duplicating scheduler code on all worker threads to enable DOMORE in SPECCROSS framework.

## 3.5 Related Work

### 3.5.1 Cross-invocation Parallelization

Loop fusion techniques [22, 75] aggregate small loop invocations into a large loop invocation, converting the problem of cross-invocation parallelization into the problem of cross-iteration parallelization. The applicability of these techniques is limited to mainly affine loops due to their reliance upon static dependence analysis. Since DOMORE is a runtime technique, it is able to handle programs with input-dependent dynamic dependences. Tseng [71] partitions iterations within the same loop invocation so that cross-invocation dependences flow within the same working thread. Compared to DOMORE, this technique is much more conservative. DOMORE allows dependences to manifest between threads and synchronizations are enforced only when real conflicts are detected at runtime.

While manually parallelizing a sequential program, programmers can use annotations provided by BOP [19] or TCC [25] systems to specify the potential concurrent code regions. Those code regions will be speculatively executed in parallel at runtime. Both techniques can be applied to exploit cross-invocation parallelism. However, they require manual annotation or parallelization by programmers while DOMORE is a fully automatic parallelization technique.

### 3.5.2 Synchronization Optimizations

Optimization techniques are proposed to improve the performance of parallel programs with excessive synchronizations (e.g, locks, flags and barriers).

Fuzzy Barrier [24] specifies a synchronization range rather than a specific synchronization point. Instead of waiting, threads can execute some instructions beyond the synchronization point. Speculative Lock Elision [56] and speculative synchronizations [39] design hardware units to allow threads to speculatively execute across synchronizations. Grace [4] wraps code between fork and join points into transactions, removing barrier synchroniza-

tions at the join points and uses a software-only transactional memory system to detect runtime conflicts and do recovery.

These techniques are designed to optimize already parallelized programs. DOMORE, instead, takes a sequential program as input and automatically transforms it into a scalable parallel program. DOMORE's runtime engine synchronizes two iterations only when necessary, and thus, does not require further optimization for synchronizations.

### 3.5.3 Runtime Dependence Analysis

Within the category of runtime dependence analysis, there are techniques which perform preprocessing of loops to identify dependences (i.e. scheduling based) and those which identify dependences in parallel with execution of the loop (i.e. speculative techniques such as transactional memory [27, 33, 66] and the LRPD family of tests [16, 61]). DOMORE is a scheduling based technique.

Generally, scheduling techniques have a non-negligible fixed overhead that changes very little based upon the number of data dependences in the program. For DOMORE, this is the overhead introduced by the scheduler. Speculative techniques typically have a small amount of fixed overhead with a highly variable amount of dynamic overhead based upon the number of data dependences, which translate to misspeculation, in a program. Therefore, for programs with a small number of dynamic dependences, speculative techniques will typically see better performance improvements. However, for programs that have more than some small number of dynamic dependences, the fixed scheduling overhead can prove to be much less than the overhead of mis-speculation recovery.

DOMORE instruments the program to detect dynamic dependences between iterations at runtime. A similar idea has been used to exploit parallelism by the Inspector-executor (IE) model [52, 59, 64], which was first proposed by Saltz et al. IE consists of three phases: inspection, scheduling, and execution. A complete dependence graph is built for all iter-

ations during the inspecting process. By topological sorting the dependence graph, each iteration is assigned to a wavefront number for later scheduling. There are two important differences between DOMORE and IE. First, DOMORE is able to exploit cross-invocation parallelism while IE is a parallelization technique limited to iterations from the same invocation. Second, IE's inspection process is serialized with the scheduling process. DOMORE overlaps the inspecting and scheduling processes for efficiency.

Cilk [8] uses a work stealing scheduler to increase load balance among processors. DOMORE can use a similar work stealing technique as an alternative scheduling policy. Baskaran et al. [3] proposed a technique that uses an idea similar to IE to remove barriers from automatically parallelized polyhedral code by creating a DAG of dependences at runtime time and using it to self-schedule code. This technique can only be used for regular affine codes whose dependences are known at compile-time while DOMORE is designed for irregular codes with dependences that cannot be determined statically. However, the DAG scheduling technique could also be integrated into DOMORE as another potential scheduling choice.

Predicate-based techniques resolve dependences at runtime by checking simple conditions. Moon et al. [42] inserts predicates before the potential parallel region. If the predicates succeed, the parallel version is executed. If they fail, the sequential version will be used instead. The same idea is used by Nicolau et al. [46] to remove synchronizations between threads. This predicate-based dependence analysis can be used by DOMORE as an efficient way to detect conflict between two iterations.

# Chapter 4

# Speculatively Exploiting Cross-Invocation Parallelism

DOMORE provides a non-speculative solution to exploiting cross-invocation parallelism using runtime information. However, DOMORE's transformation requires the construction of a scheduler thread. If the code duplication causes any side effect, DOMORE cannot be applied. Figure 4.1 demonstrates such a loop nest. The cross-invocation dependence pattern between loop invocations `L1_1` and `L1_0` is determined by the index array `C`. However, array `C` itself is updated in Loop `L2`, preventing the inspector from getting the addresses accessed in `L1` without updating the values in array `C`. Alternatively, since speculative solutions do not check the accessed memory addresses before scheduling and executing a loop iteration, they do not have the same constraints and can achieve better applicability.

This motivates the idea of SPECCROSS, the first automatic parallelization technique designed to aggressively exploit cross-invocation parallelism using high-confidence speculation. SPECCROSS parallelizes independent loops and replaces the barrier synchronization between two loop invocations with its speculative counterpart. Unlike non-speculative barriers which pessimistically synchronize to enforce dependences, speculative techniques allow threads to execute past barriers without stalling. Speculation allows programs to op-

```
   for (t = 0; t < STEP; t++) {
L1:  for (i = 0; i < M; i++) {
       A[i] = update_1(B[C[i]]);
       B[C[i]] = update_2(i);
     }
L2:  for (k = 0; k < M; k++)
       C[D[k]] = update_3(k);
   }
```

**(a) Original program**

```
t = 0
L1_0:     for (i = 0; i < M; i++) {
            A[i] = update_1(B[C[i]]);
            B[C[i]] = update_2(i);
          }
L2_0:     for (k = 0; k < M; k++)
            C[D[k]] = update_3(k);
t = 1
L1_1:     for (i = 0; i < M; i++) {
            A[i] = update_1(B[C[i]]);
            B[C[i]] = update_2(i);
          }
L2_1:     for (k = 0; k < M; k++)
            C[D[k]] = update_3(k);
                  .
                  .
                  .
t = STEP
L1_STEP:  for (i = 0; i < M; i++) {
            A[i] = update_1(B[C[i]]);
            B[C[i]] = update_2(i);
          }
L2_STEP:  for (k = 0; k < M; k++)
            C[D[k]] = update_3(k);
```

**(b) After unrolling the outermost loop**

Figure 4.1: Example program demonstrating the limitation of DOMORE transformation

timistically execute potentially dependent instructions and later check for misspeculation. If misspeculation occurs, the program recovers using checkpointed non-speculative state. Speculative barriers improve performance by synchronizing only on misspeculation.

An evaluation over eight benchmark applications on a 24-core machine shows that SPECCROSS achieves a geomean speedup of 4.6× over the best sequential execution. This compares favorably to a 1.3× speedup obtained by parallel execution without any cross-invocation parallelization.

In the following sections, we first motivate software-only speculative barrier by comparing it with all other options including non-speculative barrier and hardware-based speculative barrier. Then details about design and implementation of SPECCROSS are given, after which we describe how SPECCROSS is applied for automatic parallelization.

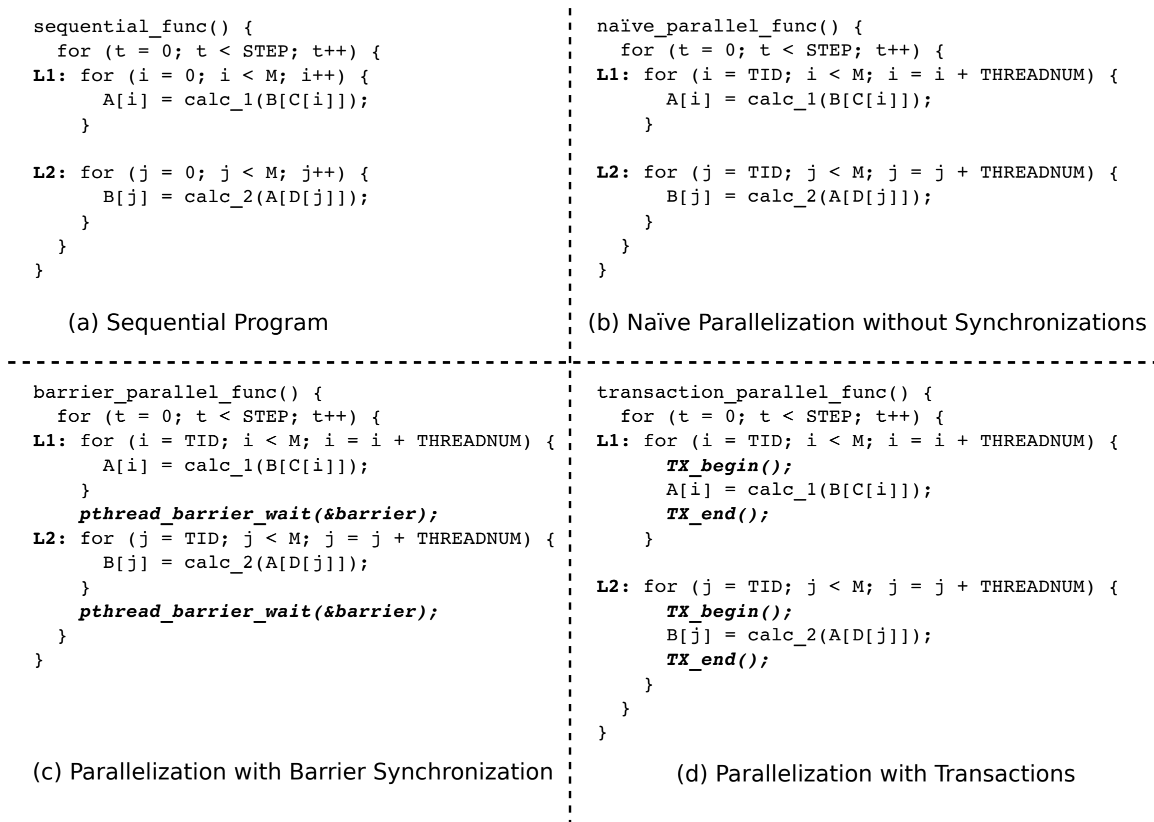



Figure 4.2: Example of parallelizing a program with different techniques

## 4.1 Motivation and Overview

### 4.1.1 Limitations of analysis-based parallelization

Figure 4.2(a) shows a code example before parallelization. In this example, loop `L1` updates array elements in array `A` while loop `L2` reads the elements from array `A` and uses the values to update array `B`. The whole process is repeated `STEP` times. Both `L1` and `L2` can be individually parallelized using DOALL [1]. However, dependences between `L1` and `L2` prevent the outer loop from being parallelized. Ideally, programmers should only synchronize iterations that depend on each other, without stalling the execution of independent iterations. If static analysis [49, 71, 77] could prove that each thread accesses a separate section of arrays `A` and `B`, no synchronization is necessary between two adjacent loop invocations (Figure 4.2(b)). However, since arrays `A` and `B` are accessed in an irregular manner

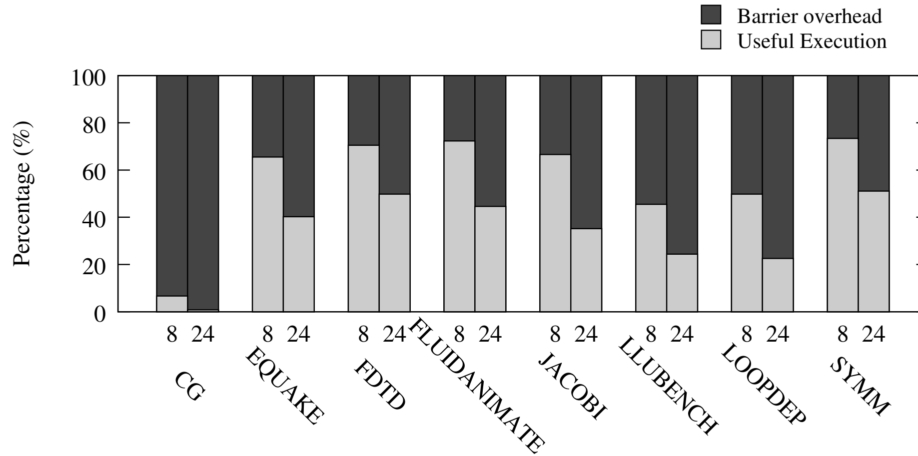



Figure 4.3: Overhead of barrier synchronizations for programs parallelized with 8 and 24 threads

(through index arrays `C` and `D`), static analysis cannot determine the dependence pattern between `L1` and `L2`. As a result, this naïve parallelization may lead to incorrect runtime behavior.

Alternatively, if static analysis could determine a dependence pattern between iterations from two invocations, e.g., iteration `1` from `L2` always depends on iteration `2` from `L1`, then fine-grained synchronization can be used to synchronize only those iterations. But this requires accurate analysis about the dependence pattern, which is in turn, limited by the conservative nature of static analysis. Instead, barrier synchronization is used to globally synchronize all threads (Figure 4.2(c)). Barrier synchronization conservatively assumes dependences between any pair of iterations from two different loop invocations. All threads are forced to stall at barriers after each parallel invocation, which greatly diminishes effective parallelism. Figure 4.3 shows the overhead introduced by barrier synchronizations on eight programs parallelized with 8 and 24 threads. Barrier overhead refers to the total amount of time threads sit idle waiting for the slowest thread to reach the barrier. For most of these programs, barrier overhead accounts for more than 30% of the parallel execution time and increases with the number of threads. This overhead translates into an Amdahl's law limit of 3.33× max program speedup.

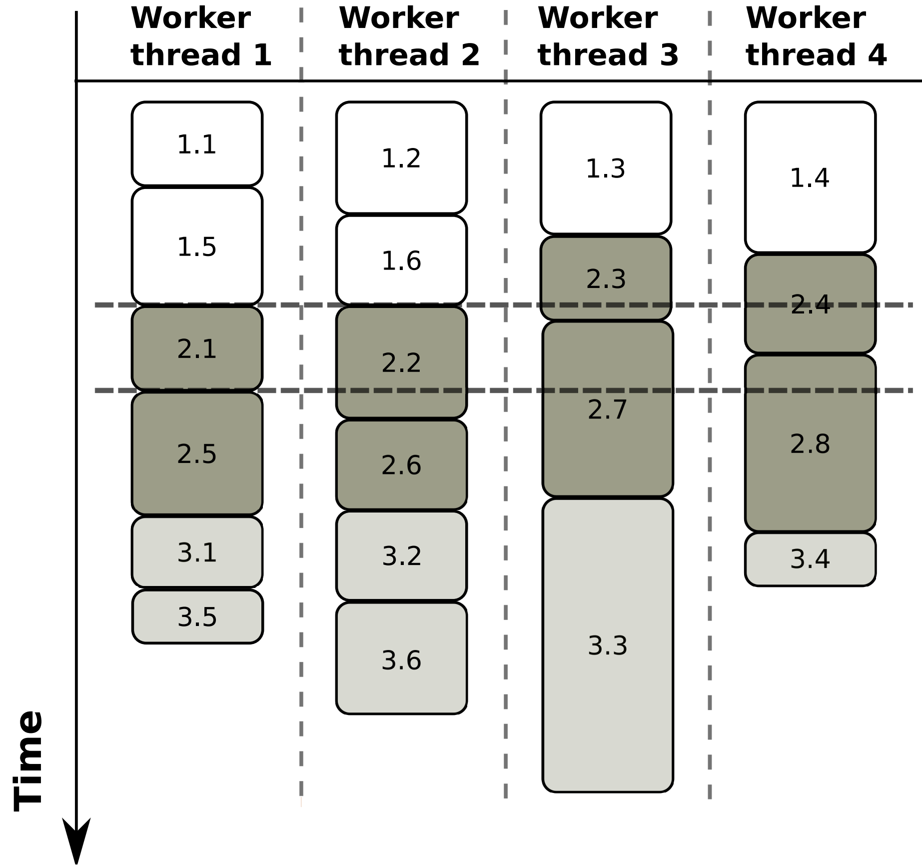



Figure 4.4: Execution plan for TM-style speculation: each block $A.B$ stands for the $B_{th}$ iteration in the $A_{th}$ loop invocation: iteration $2.1$ overlaps with iterations $2.2, 2.3, 2.4, 2.7, 2.8$, thus its memory accesses need to be compared with theirs even though all these iterations come from the same loop invocation and are guaranteed to be independent.

### 4.1.2 Speculative cross-invocation parallelization

The conservativeness of barrier synchronization prevents any cross-invocation parallelization and significantly limits performance gain. Alternatively, the optimistic approach unlocks potential opportunities for cross-invocation parallelization. It allows the threads to execute across the invocation boundary under the assumption that the dependences rarely manifest at runtime. Two major optimistic solutions have been proposed in literature so far: one relies on transactional memory and the other uses speculative barriers.

Figure 4.2(d) demonstrates the basic idea of speculative parallelization using transactional memory systems (TM) [27]. In this example, each inner loop iteration is treated as

a separate transaction. The commit algorithms proposed in Grace [4] and TCC [25] allow transactions within the same inner loop invocation to commit out of order but guarantee transactions from later invocations should commit after those from earlier ones. However, this approach assumes that every transaction may conflict with another transaction and must be compared against each other for violation detection. It ignores the important fact that for most programs which could benefit from cross-invocation parallelization, all iterations from the same invocation are often guaranteed to be independent at compile time and wrapping them in separate transactions introduces unnecessary runtime checking and commit overhead. For example, in Figure 4.4, the execution of the first iteration of the second loop invocation (annotated as `2.1`) overlaps with that of iterations `2.2`, `2.3`, `2.4`, `2.7` and `2.8`. Even though they come from the same loop invocation, the TM framework needs to check them for access violations before `2.1` can commit. More coarse-grained transactions can reduce this checking overhead, but they also increase the possibility of misspeculation between two transactions.

Speculative barrier synchronization, on the other hand, preserves the DOALL property of each loop invocation while still allowing threads to execute past the barrier without stalling. Since all existing speculative barrier synchronization techniques [29, 39, 44] require specialized hardware support, we refer to these techniques collectively as hardware-based barrier speculation (HWBS).

The work of Martínez et al. [39] is representative of the HWBS techniques. In this work, all worker threads start executing non-speculatively and are allowed to update the shared memory concurrently. A worker thread becomes speculative once it exceeds the barrier boundary while other worker threads are still executing before the barrier. When a worker thread becomes speculative, it first executes a checkpoint instruction to back up the architectural register values. During the speculative execution, values updated by the speculative thread are buffered in the cache and corresponding cache lines are marked as speculative. Later, access to a speculative cache line by other threads will trigger an access

conflict. When that happens, the speculatively executing thread is squashed and resteered to the checkpoint. Compared to execution models supported by TM, HWBS distinguishes speculative and non-speculative threads to avoid unnecessary value buffering and violation checking. Non-speculative worker threads can commit their writes concurrently without waiting. As a result, HWBS is regarded as a better solution for a program pattern, where tasks in the same loop invocation are independent while tasks from different invocations may depend on each other.

Despite its effectiveness, HWBS requires specialized hardware to detect misspeculation and recover. Programs parallelized for commodity hardware cannot benefit from it. This limitation motivates us to design and implement the first software-only speculative barrier for SPECCROSS, which aims at generating scalable parallel programs for commodity multicore machines.

### 4.1.3 Automatic cross-invocation parallelization with software-only speculative barrier

SPECCROSS is implemented to generate multi-threaded programs running on commodity multi-core machines. SPECCROSS consists of three components: a parallelizing compiler, a profiling library and a runtime library. The parallelizing compiler automatically detects and parallelizes a code region with many loop invocations. The profiling library determines how aggressively to speculate and is the key to achieving high confidence speculation. The runtime library provides support for speculative execution, misspeculation detection, and recovery.

SPECCROSS's runtime library implements a software-only speculative barrier, which works as a light-weight substitution for HWBS on commodity hardware. This runtime library provides efficient misspeculation detection. In order to check for misspeculation, all threads periodically send memory access signatures to a dedicated checker thread. A memory access signature summarizes which bytes in memory have been accessed since the

last check. The checker thread uses signatures to verify that speculative execution respects all memory dependences. If the checker thread detects misspeculation, it kills all the speculative threads and resumes execution from the last checkpoint. If two threads calculate and send signatures while they are executing in code regions guaranteed to be independent, these signatures will be safely skipped to avoid unnecessary checking overhead.

To avoid high penalty from misspeculation, SPECCROSS uses a profiler to determine the *speculative range*, which controls how aggressively to speculate. The speculative range is the distance between a thread with a high *epoch number* and thread with a low *epoch number*. In SPECCROSS, an epoch number counts the number of barriers each thread has passed. Non-speculative barriers ensure that all threads have the same epoch number, whereas speculative barriers allow each thread's epoch number to increase independently. When the difference of two threads' epoch numbers is zero, misspeculation is impossible, since nothing is speculated. When the difference is high, misspeculation is more likely, since more intervening memory accesses are speculated. The SPECCROSS runtime system uses profiling data to compute a limit for the speculative range. When the program reaches the limit, SPECCROSS stalls the thread with the higher epoch number until the thread with the lower epoch number has caught up. Choosing the correct limit keeps misspeculation rates low.

Figure 4.5 gives an overview of the SPECCROSS system. The SPECCROSS compiler takes a sequential program as input and detects a SPECCROSS code region as the transformation target. A SPECCROSS code region is often an outermost loop composed of consecutive parallelizable inner loops. SPECCROSS parallelizes each independent inner loop and then inserts SPECCROSS library function calls to enable cross-invocation parallelization. At runtime, the original process spawns multiple worker threads and a checker thread for misspeculation detection. Each worker thread executes its code section and sends requests for misspeculation detection to the checker thread. The checker thread detects dependence violations using access signatures computed by each worker thread. When the checker

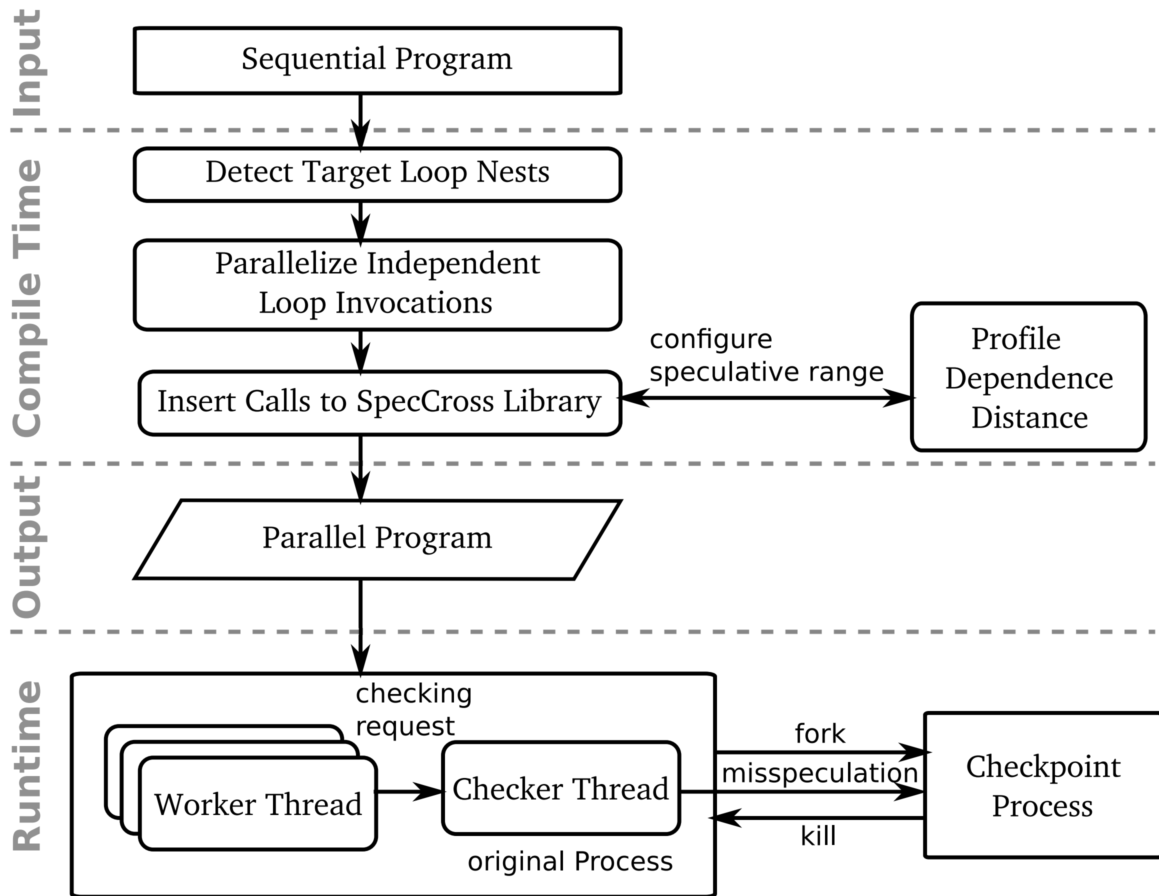



Figure 4.5: Overview of SPECCROSS: At compile time, the SPECCROSS compiler detects code regions composed of consecutive parallel loop invocations, parallelizes the code region and inserts SPECCROSS library functions to enable barrier speculation. At runtime, the whole program is first executed speculatively without barriers. Once misspeculation occurs, the checkpoint process is woken up. It kills the original child process and spawns new worker threads. The worker threads will re-execute the misspeculated epochs with non-speculative barriers.

thread detects a violation, it signals a separate checkpoint process for misspeculation recovery. The checkpoint process is periodically forked from the original process. Once recovery is completed by squashing all speculative workers and restoring state to the last checkpoint, misspeculated epochs are re-executed with non-speculative barriers.

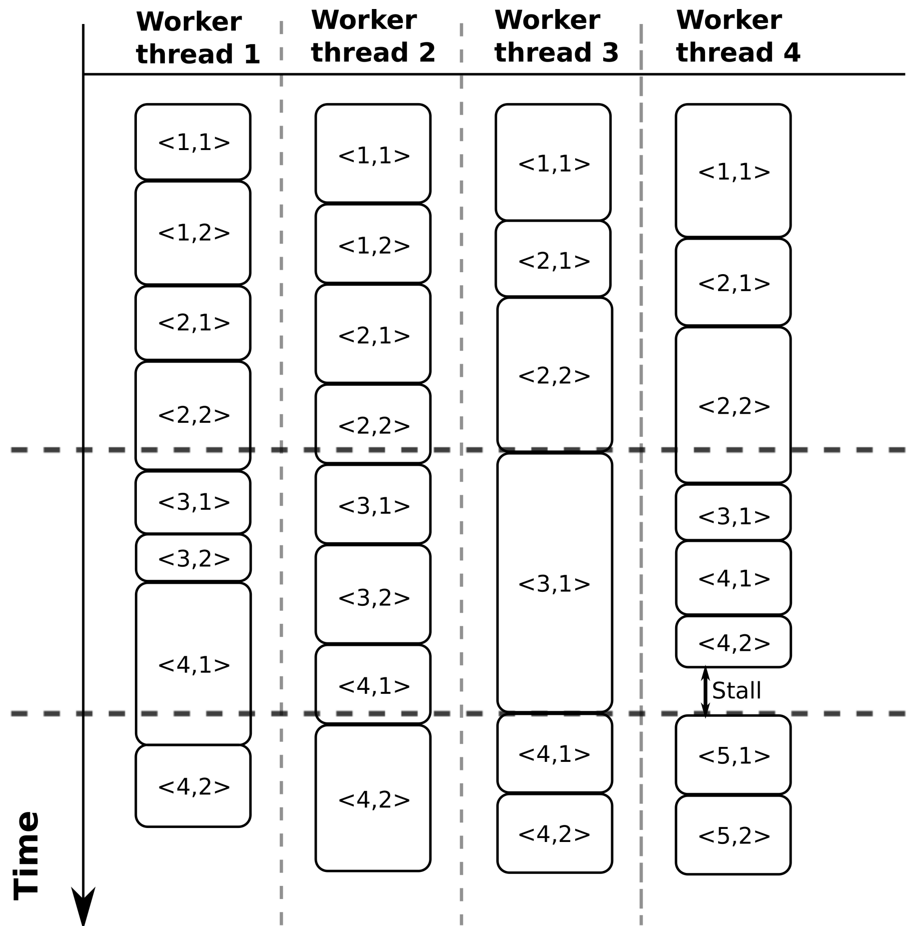



Figure 4.6: Timing diagram for SPECCROSS showing epoch and task numbers. A block with label $<A, B>$ indicates that the thread updates its epoch number to $A$ and task number to $B$ when the task starts executing.

**Worker thread :**

```
for each epoch i {
  for each task j in epoch i {
    update(epoch_number, task_number);
    other_epoch_task_nums = collect_other_threads();
    signature = do_task(i, j);
    store_signature(signature, SIGLOG);
    request = compose_request(other_epoch_task_nums);
    send_to_checker(request);
  }
}
send_to_checker(END_TOKEN);
```

**Checker thread :**

```
while (true) {
  for each worker thread w {
    if (finished[w] != true) {
      request = consume_request();
      if (request == END_TOKEN)
        finished[w] = true;
      ret = check_request(request);
      if (ret == MISSPECULATION)
        send_signal(checkpoint_process, FAIL);
    }
  }
  for each worker thread w
    if (all finished[w] == true)
      return SUCCESS;
}
```

Figure 4.7: Pseudo-code for worker threads and checker thread

## 4.2 SpecCross Runtime System

### 4.2.1 Misspeculation Detection

The runtime system of SpecCross implements a software-only speculative barrier. Non-speculative barriers guarantee all code before a barrier executes before any code after the barrier. Speculative barriers violate this guarantee to achieve higher processor utilization and thus higher performance. However, to maintain the same semantics as non-speculative barriers, speculative barriers check for runtime dependence violations. Upon detecting a violation, the runtime system triggers a misspeculation and rolls back to non-speculative state using the most recent checkpoint.

SpecCross uses signature-based violation detection [12, 41, 67, 76], which avoids the overhead of logging each memory access. A signature is an approximate summary of memory accesses, providing a customizable tradeoff between signature size and false positive rates. SpecCross provides a general API for users to provide their own signature generators. However, our experiments have shown very simple signatures are effective in practice. By default, SpecCross summarizes the accesses by saving the minimum and maximum addresses speculatively accessed. In this signature scheme, threads are independent if their signatures do not overlap. Range-based signature schemes work well when memory accesses are clustered. For random access patterns, a Bloom filter-based signature offers lower false positive rates.

To determine when checking is needed, SpecCross assigns each thread an *epoch number* and a *task number*. An *epoch* is defined as the code region between two consecutive speculative barriers, and the *epoch number* counts how many speculative barriers a thread has passed. A *task* is the smallest unit of work that can be independently assigned to a thread, and the *task number* counts how many tasks a thread has executed since the last barrier. For many parallel programs, a task is one loop iteration. When a thread begins a new task, it reads the current epoch and task numbers of all other threads and later com-

municates these numbers to the checker thread. The checker compares the task's access signature with all signatures from epochs earlier than the signature's epoch, but at least as recent as the epoch-task number pair recorded when the task began. Signatures from the same epoch are safely ignored since they are not separated by a barrier. This avoids unnecessary comparisons between two tasks in the same epoch to reduce the runtime overhead of the checker thread.

Figure 4.6 is a timing diagram for speculative barrier execution. In the timing diagram, a task marked as <A,B> indicates that the thread updates its epoch number to A and task number to B when the task starts executing. Epoch number-task number pairs are not unique across threads, since each thread counts tasks independently. When worker thread 3 starts task <3,1>, the other worker threads are still executing task <2,2> in the second epoch. The access signatures of these three tasks need to be checked against that of task <3,1>. Before task <3,1> finishes, all other threads have already begun the fourth epoch. As a result, the access signatures of task <4,1> in threads 1, 2, and 4 and <4,2> in thread 4 need to be checked against that of task <3,1>.

Figure 4.7 shows the order of operations for worker threads. At the beginning of a task, the worker thread updates the epoch and task numbers, then records the current epoch and task numbers for the other threads. Next, the worker executes the task. While executing the task, the worker computes the memory access signature. Finally, the worker thread saves its signature in a signature log and sends information to the checker thread so the checker thread can detect misspeculation asynchronously. Figure 4.8 demonstrates the data structure used for signature logging. Each worker threads has one thousand entries in the signature log because checkpointing is operated every thousandth epochs. Worker threads synchronize at the checkpoint and signature log entries can be re-used after checkpointing. Each entry of the signature log contains a pointer to an array used for saving signatures within a single epoch. The size of the array is initialized to store 1024 signatures and expand if the number of tasks exceeds the array size.

To detect misspeculation, the checker thread needs the memory access signatures, the worker thread's epoch and task numbers and the epoch and task numbers of the other worker threads when the task began. After sending the data to the checker thread, a worker thread may stall to wait for other worker threads if continuing execution would exceed the speculative range. In Figure 4.6, thread 4 stalls after executing task `<4,2>`. When thread 4 tries to start task `<5,1>`, it determines the distance to thread 3's task `<3,1>` is three. In the example, the speculative range limit is two, so thread 4 stalls until thread 3 finishes task `<3,1>`. The example is simplified, since in real programs the speculative range is always at least the number of threads and usually much larger.

There are two subtle memory consistency issues with the checking scheme described above. First, the checking scheme assumes that updates to the epoch and task numbers will be globally visible after all other stores in the previous task. If the memory consistency model allows the architecture to reorder stores, this assumption will be false. In other words, the checking methodology assumes a Total Store Order (TSO) architecture. Modern TSO architectures include: x86, x86-64, SPARC, and the IBM zSeries [40]. For architectures that do not support TSO, such as ARM and POWER, each thread should execute a memory fence before updating the epoch and task numbers. The costs of memory fences may be greater than the costs of speculative barriers when the number of tasks per epoch is high.

Second, the epoch and task numbers must update together atomically. The easiest way to accomplish this is to store these numbers as the high and low bits of a 64-bit word and use an atomic write operation. For x86-64, 64-bit writes are atomic by default, so no special handling is required.

### 4.2.2 Checkpointing and Recovery

Periodically, the worker threads checkpoint so that their state can be restored after misspeculation. Infrequent checkpointing reduces the overhead of the runtime system, but

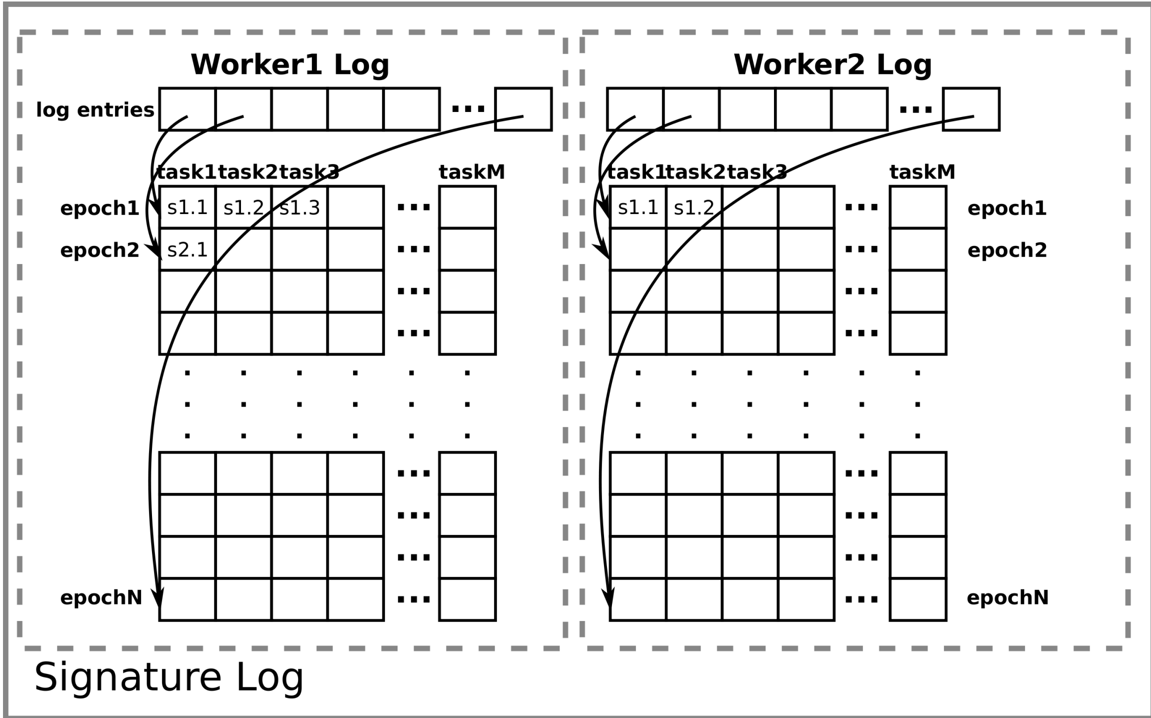



Figure 4.8: Data structure for Signature Log

increases the cost of misspeculation. SPECCROSS's profiling library enables very low rates of misspeculation, thus infrequent checkpointing is efficient in practice. By default, SPECCROSS checkpoints at every thousandth speculative barrier, though it can be re-configured based on program characteristics. Checkpoints act as non-speculative barriers. All worker threads synchronize at the checkpoint waiting for the checker thread to finish all checking requests before the checkpoint, ensuring the checkpoint's state is safe.

In SPECCROSS, checkpointing is divided into two parts. SPECCROSS first saves the register state of each thread, and then saves the memory of the entire process. The C standard library's `setjmp` function is used to save the register state of each thread. After each thread is saved, the process forks, duplicating the state of the entire process. The newly forked child sleeps until the checker thread detects misspeculation. To restore a checkpoint, the child spawns new worker threads, and each newly spawned thread executes `longjmp` to inherit the program state of an original thread.

SPECCROSS explicitly allocates worker threads' stacks to ensure they will not be deal-

located by `fork`. This is necessary, because the POSIX standard allows processes to deallocate the stacks of associated threads after forking. Explicitly allocating the worker threads' stacks ensures that `longjmp` will restore a valid stack.

Experiments show recovering from misspeculation requires about one millisecond. The execution time for recovering non-speculative state is dominated by the `kill` and `clone` syscalls. The main thread invokes `kill` to asynchronously terminate misspeculating threads and awaken the checkpoint process, then invokes `clone` (internally called by `pthread_create`) to create new worker threads. The number of syscalls scales linearly with the number of threads, but performance is not affected by the program's memory footprint or the size of speculative state.

Some epochs contain irreversible operations (for example I/O) which must be executed non-speculatively. Before entering an irreversible epoch, all worker and checker threads synchronize. Just like checkpointing, synchronizing all threads ensures non-speculative execution. After exiting the irreversible region, the program checkpoints. Otherwise, later misspeculation could cause the irreversible region to be repeated.

There are three conditions which trigger misspeculation. First, the checker thread triggers misspeculation if it finds a pair of conflicting signatures. Second, misspeculation occurs if any of the worker threads triggers a segmentation fault. Third, misspeculation can be triggered in response to a user defined timeout. Timeouts are necessary, since speculative updates to the shared memory may change the exit condition of a loop and cause infinite execution. After detecting misspeculation, the most recent checkpoint is restored. After restoring the checkpoint, the misspeculated epochs are re-executed with all speculative barriers replaced with their non-speculative counterparts.

### 4.2.3 Runtime Interface

This section describes the runtime interface exposed by SPECCROSS. Table 4.1 lists the interface functions along with a short description of each function's semantics. Figure 4.9

shows an instantiation of these functions in a parallel program which will serve as a running example. In the example, each inner loop invocation is treated as an epoch, and each inner loop iteration is considered a task. SPECCROSS provides the same interface functions for both profiling and speculation purposes. As a result, these function calls only need to be inserted once for both profiling run and speculative execution. Whether to do profiling or speculation is decided by defining the environment variable `MODE`. Depending on the current `MODE`, functions will either speculate or profile. The `MODE` value can also be set to NON-SPECULATIVE. In non-speculative mode, most interface functions do nothing and speculative barriers are replaced with non-speculative ones. This non-speculative mode is enabled when re-executing the misspeculated epochs after recovering from misspeculation. The details of SPECCROSS's profiling and speculation functions are as follows:

| **Operation** | **Description** |
|---|---|
| **Functions for Both Profiling and Speculation** | |
| init() | Initialize data structures used for barrier profiling or speculative execution. If in speculation mode, checkpoint the program before beginning the parallel execution. |
| exit_task(threadID) | Record the signature of the current task in global signature log. Increment the *task number*. If in profiling mode, compare the signature of current task with signatures of tasks belonging to previous epochs and return the minimum dependence distance. If in speculation mode, return value is 0. |
| spec_access(threadID, callback, addr_list) | Apply callback function to each address in the addr_list to compute the signature. |
| enter_barrier(threadID, loop_name) | Increment the *epoch number*. If in profiling mode, execute the actual non-speculative barrier operation. If in speculation mode, checkpoint according to the checkpointing frequency. |
| create_threads(threads, attrs, start_routines, args) | Create worker threads and if in speculation mode, create a checker thread for violation detection. |
| cleanup() | Wait for worker threads and checker thread to finish. Free data structures allocated for profiling or speculation. |
| **Functions for Speculation Only** | |
| enter_task(threadID, spec_distance) | Collect *epoch number* and *task number* of other worker threads and send them to the checker thread. The parameter spec_distance specifies the speculation distance between two tasks. |
| send_end_token(threadID) | Send an END_TOKEN to checker thread to inform it of the completion of a worker thread. |
| sync() | Synchronize all threads before entering the next epoch. |
| checkpoint() | Checkpoint program state before entering the next epoch. |

Table 4.1: Interface for SPECCROSS Runtime Library

1. **Initialization**. The `init` function performs initialization by setting up bookkeeping data structures, including allocating memory for signatures, the epoch number, and the task number. In speculation mode, `init` checkpoints the program before beginning parallel execution.

2. **Thread Creation**. The `create_threads` function spawns new threads in both profiling and speculation modes. Unlike pthreads, a SPECCROSS thread's stack is not deallocated after the thread exits. This implementation detail is vital for checkpointing and recovery, because it allows a new thread to reclaim an old thread's stack using `longjmp`.

3. **Barrier Entry**. The `enter_barrier` function increments the *epoch number* upon entry into a new epoch. In profiling mode or non-speculative mode, it executes the non-speculative barrier. The parameter loop_name is set to be the name of the loop header basic block so that profiling could compute a minimum dependence distance for each independent loop. While in speculation mode, it skips barrier synchronization. The `enter_barrier` function is also responsible for periodically checkpointing speculative state.

4. **Dependence Tracking**. To compute the access signatures for each task, its memory accesses (loads and stores) are instrumented using the `spec_access` function. Loads and stores need to be instrumented if they belong to different epochs and may alias with each other. For typical programs, only a few memory accesses require instrumentation. The `spec_access` function takes as input the memory location referred in the memory access instruction and a pointer to the signature generator function. Passing a pointer to the signature generator provides users with the flexibility to choose customized functions that are specific to each parallel program. This in turn enables SPECCROSS to achieve low conflict rate across programs with differing memory access patterns.

**Main thread:**

```
main() {
  init();
  create_threads(threads, attrs, SpecCross_parallel_func, args);
  cleanup();
}
```

**Worker thread:**

```
SpecCross_parallel_func(threadID) {
  for (t = 0; t < STEP; t++) {
    enter_barrier(threadID, "L1");
L1: for (i = threadID; i < M; i = i + THREADNUM) {
      enter_task(threadID, minimum_distance_L1);
      spec_access(threadID, callback, &A[i], &C[i]);
      A[i] = do_work(B[C[i]]);
      exit_task(threadID);
    }
    enter_barrier(threadID, "L2");
L2: for (j = threadID; j < M; j = j + THREADNUM) {
      enter_task(threadID, minimum_distance_L2);
      spec_access(threadID, callback, &C[D[j]], &A[j]);
      B[j] = do_work(A[D[j]]);
      exit_task(threadID);
    }
  }
  send_end_token(threadID);
}
```

Figure 4.9: Demonstration of using SPECCROSS runtime library in a parallel program

5. **Task Initialization**. Function `enter_task` is inserted before each task. It is not used in profiling mode. In speculation mode, it collects epoch number and task number of worker threads other than the caller thread and sends the number pairs to the checker thread. The value of spec_distance specifies the speculation distance between two tasks. This value is set according to the minimum distance returned by the profiling run. The worker thread is stalled if it is executing beyond the minimum distance compared to other worker threads.

6. **Task Finalization**. Once a task completes execution, `exit_task` is called. In profiling mode, this function logs the access signature of the task, compares this ac-

cess signature with signatures of tasks executed by other threads in previous epochs. When detecting memory access conflicts, it records the *dependence distance* between pairs of conflicting tasks. After recording the minimum dependence distances between all conflicting tasks, it increments the *task number*. In speculation mode, it simply increments the *task number* of the caller thread.

7. **Thread Completion**. In speculation mode, after completing the execution of all epochs, function `send_end_token` sends an `END_TOKEN` to the checker thread. In profiling mode, it does nothing.

8. **Enforcing Synchronization**. Two functions, `sync` and `checkpoint`, can be inserted before any epoch to synchronize all threads or to do an extra checkpoint. This is useful when a certain epoch is known to cause conflicts or contain irreversible operations (e.g. I/O). Users can use `sync` to synchronize all threads before entering this epoch and do an extra checkpoint operation before starting the execution of the next epoch.

9. **Speculation Finalization**. Function `cleanup` is inserted at the end of the program which waits for worker threads and checker thread to exit and then frees memory allocated for profiling or speculation.

## 4.3 SPECCROSS Parallelizing Compiler

SPECCROSS targets a code region composed of large amounts of parallel loop invocations. Often, such a code region is an outermost loop which contains multiple parallelizable inner loops. To locate these code regions, we analyze all hot loops within the sequential program. A lot loop accounts for at least 10% of the overall execution time during the profiling run. A hot loop is a candidate for SPECCROSS if it satisfies three conditions: first, the outermost loop itself cannot be successfully parallelized by any parallelization technique

implemented in the Liberty parallelizing compiler infrastructure; second, each of its inner loop can be independently parallelized by a none-speculative and none partition-based parallelization technique such as DOALL and LOCALWRITE; finally, the sequential code between two inner loops can be privatized and duplicated among all worker threads.

After locating the candidate loops, SPECCROSS transformation is applied on each of them. SPECCROSS transformation consists of two major steps: first, we parallelize the hot loop by applying DOALL or LOCALWRITE to each inner loop and duplicate the sequential code among all worker threads; then SPECCROSS library function calls are inserted in the parallel program to enable barrier speculation. Algorithm 5 demonstrates how these function calls are inserted:

1. A code region between two consecutive `enter_barrier` functions is considered as an epoch. For simplicity, each inner loop invocation is usually treated as an epoch. As a result, `enter_barrier` functions are inserted at the beginning of each inner loop's preheader basic block.

2. Function `enter_task` marks the beginning of each task. Since an inner loop iteration is usually treated as a separate task, `enter_task` functions are inserted at the beginning of each inner loop's header.

3. Function `exit_task` is invoked at the end of each task. In other words, it is invoked before the execution exits an inner loop iteration or before the execution enters another inner loop iteration. Algorithm 5 (line 18-36) goes through each basic block in an inner loop, checking its terminator instruction. If a terminator instruction is an unconditional branch which either exits the loop or branches back to the header of the loop, an `exit_task` call is inserted right before the terminator instruction. If the terminator instruction is a conditional branch instruction and (1) if one of its targets is a basic block outside the loop and the other is the loop header, an `exit_task` function is also inserted before the terminator instruction; (2) if one target is a basic block

outside the loop and the other is a basic block within the loop except the loop header, the `exit_task` function is invoked only when the execution exits the loop; (3) if one target is the header of the loop and the other is some other basic block within the loop, an `exit_task` function is invoked only when the execution branches back to the loop header.

4. Function `spec_access` is used to calculate the access signature for each task. It is inserted before each memory operation (store or load) that is involved in a cross-invocation dependence (line 40-49).

**Algorithm 5**: Pseudo-code for SPECCROSS Library Function Calls Insertion

```
1: Input ParallelLoops: Doallable or Localwritable loops in a SPECCROSS code region
2: Input CrossInvocationDeps: cross-invocation dependences
3: for all Loop l ∈ ParallelLoops do
4:    blocks = getBasicBlocks(l)
5:    // get the loop preheader and header
6:    preheader = getLoopPreheader(l)
7:    header = getLoopHeader(l)
8:    // get the loop exit basic blocks
9:    loopExits = getLoopExitBlocks(l)
10:   // get the basic blocks containing the loop backedge
11:   backEdgeBlocks = getBackEdgeBlocks(l)
12:   // insert enter_barrier
13:   insertPt = BeginningOfBlock(preheader);
14:   insertEnterBarrier(insertPt);
15:   // insert enter_task
16:   insertPt = BeginningOfBlock(header);
17:   insertEnterTask(insertPt);
18:   // insert exit_task
19:   for all basic block block ∈ blocks do
20:      termInst = getTerminatorInst(block)
21:      inertPt = Before(termInst);
22:      if block ∈ loopExits then
23:         if isUnconditional(termInst) || block ∈ backEdgeBlocks then
24:            insertExitTask(insertPt);
25:         else
26:            insertInvokeExitTaskWhenExitTaken(insertPt);
27:         end if
28:      else if block ∈ backEdgeBlocks then
29:         if isUnconditional(termInst) then
30:            insertExitTask(insertPt);
31:         else
32:            insertInvokeExitTaskWhenBackEdgeTaken(insertPt);
33:         end if
34:      end if
35:   end for
36: end for
37: // insert spec_access
38: for all Dependence dep ∈ CrossInvocationDeps do
39:   srcInst = getSrcInstruction(dep);
40:   dstInst = getDstinstruction(dep);
41:   srcAddress = getAddress(srcInst);
42:   dstAddress = getAddress(dstInst);
43:   insertPt = Before(srcInst);
44:   insertSpecAccess(insertPt);
45:   insertPt = Before(dstinst);
46:   insertSpecAccess(insertpt);
47: end for
```

## 4.4 SPECCROSS Profiling

SPECCROSS provides a profiling API to determine whether speculation is profitable or not. In order to detect dependences, the signature of each task is compared with signatures of tasks belonging to earlier epochs. After dependences are detected between two tasks, the profiling function records the dependence distance between them. Dependence distance is defined as the number of tasks between two conflicting tasks. After profiling, a minimum dependence distance is determined. If the minimum dependence distance is smaller than a threshold value, speculation will not be done. By default, the threshold value is set to be equal to the number of worker threads.

If the dependence distance is large, it means the program has an access pattern suitable for barrier speculation. To reduce the possibility of misspeculation, the minimum dependence distance is passed as an input parameter to the speculation runtime library. At runtime, the leading thread stalls if it executes beyond this distance.

## 4.5 Related Work

### 4.5.1 Barrier Removal Techniques

Barrier synchronization is often inserted in parallel applications conservatively. Some barriers can be removed using static analysis if there is no cross-thread data flow. Several barrier removal techniques are based on disproving cross-thread dependences [49, 71, 77]. Other techniques restructure code to avoid the need for barrier synchronization. Zhao et al. [78] transform an inner DOALL loop to an outer DOALL loop in order to remove barriers between two consecutive parallel loop invocations. Ferrero et al. [22] aggregate small parallel loops into large ones for the same purpose. All these barrier removal techniques rely on static analysis to remove barriers. Compared to them, SPECCROSS is not limited by the conservativeness of static analysis. These approaches are complementary to SPEC-

CROSS, since SPECCROSS can be applied on code that has already been optimized using static barrier removal techniques.

Some techniques speculatively remove barriers using specialized hardware. Nagarajan and Gupta [44] speculatively execute parallel programs past barriers and detect conflicts by re-designing the Itanium processor's *Advanced Load Address Table* (ALAT) hardware. Martínez and Torrellas [39] apply *thread level speculation* [48, 69] to speculatively remove barriers. A speculative synchronization unit (SSU) is designed to detect conflict and recover from misspeculation at runtime. ECMon [43] exposes cache events to software so that barrier speculation could efficiently detect violations at runtime. These techniques are not limited by complicated dependence patterns; however, they require special hardware support. By contrast, SPECCROSS is a software-based technique that runs on existing multi-core machines.

### 4.5.2 Alternative Synchronizations

The Fuzzy Barrier [24] specifies a synchronization range rather than a specific synchronization point. However, it also relies on the static analysis to identify instructions that can be safely executed while a thread is waiting for other threads to reach the barrier. Fuzzy barriers cannot reduce the number of barriers; it can only reschedule barriers for more efficient execution.

### 4.5.3 Transactional Memory Supported Barrier-free Parallelization

Transactional Memory (TM) [27] was first introduced as a hardware technique providing a way to implement lock-free data structures and operations. The idea was later implemented as a software-only construct [66]. The key idea behind TM is to make a sequence of memory reads and writes appear as a single transaction; all intermediate steps are hidden from the rest of the program. A log of all memory accesses is kept during the transaction. If a transaction reads memory that has been altered between the start and end of the transac-

tion, execution of the transaction is restarted. This continues until the transaction is able to complete successfully, and its changes are committed to memory. Original TM systems do not support inter-loop speculation since it does not enforce the commit order between transactions from different loop invocations. Grace [4] and TCC [25] extend existing TM systems to support inter-loop speculation. Grace automatically wraps code between fork and join points into transactions, removing barrier synchronizations at the join points. Each transaction commits according to its order in the sequential version of code. TCC [25] relies on programmers to wrap concurrent tasks into transactions. It requires special hardware support for violation checking and transaction numbering, which ensures the correct commit order of each transaction. As discussed in Section 4.1, both systems ignore the important fact that large amounts of independent tasks do not need to be checked for access violations. Instead, SPECCROSS is customized for this program pattern, thus it avoids unnecessary overhead in checking and committing.

### 4.5.4 Load Balancing Techniques

Work stealing techniques, implemented in many parallel subsystems like Cilk [8], Intel TBB [62], and X10 [13], balance load amongst parallel threads by allowing one thread to steal work from another thread's work queue. Balancing workloads in turn reduces the impact of barrier synchronization on program performance. However, existing work stealing implementations only allow workers to steal work from the same epoch at any given time. This is because these implementations do not leverage knowledge about program dependences to steal work from multiple epochs (across barriers) at the same time. As a result, programs that have a limited number of tasks in a single epoch (for example, CG in our evaluation) do not benefit from work stealing techniques. In contrast, SPECCROSS allows tasks from different epochs to overlap and achieve better load balance across epochs. Other load balancing techniques such as guided-self scheduling [51], affinity-based scheduling [38], and trapezoidal scheduling [72] also suffer from the same limitations as the work stealing

techniques.

Synchronization via scheduling [5] is a method of load balancing that employs static and dynamic analyses to capture runtime dependences between tasks in a task graph. The task graph is exposed to a scheduler that schedules the tasks onto threads in a way so as to minimize idling times of each thread. While being able to handle more general dependence patterns than working stealing, this technique still does not overlap tasks from successive executions of the same task graph in parallel.

### 4.5.5 Multi-threaded Program Checkpointing

Several prior techniques implement multithread program checkpointing. Dieter et al. [18] propose a user-level checkpoint system for multi-threaded applications. Carothers et al. [11] implement a system call to transparently checkpoint multi-threaded applications. SPECCROSS checkpoints the multi-threaded programs for misspeculation recovery. These checkpointing techniques could be merged into the SPECCROSS framework.

### 4.5.6 Dependence Distance Analysis

If dependence distance manifests in a regular manner, programs can still be parallelized by assigning dependent tasks to the same worker thread. Dependence distance analysis [70] has been proposed to achieve that purpose. As other static analysis, these techniques tend to be conservative and cannot handle irregular dependence patterns. The SPECCROSS takes advantage of profiling information to get a minimum distance and speculates it remains with other input set to further reduce the misspeculation rate. Since the information comes from profiling, it applies to programs with irregular dependence patterns as well.

# Chapter 5

# Evaluation

The implementations of DOMORE and SPECCROSS systems are evaluated on a single platform: a 24-core shared memory machine which has four Intel 6-core Xeon X7460 processors running at 2.66 GHz with 24 GB of memory. Its operating system is 64-bit Ubuntu 9.10. The sequential baseline compilations are performed by the clang compiler version 3.0 at optimization level three.

The benchmark programs evaluated in this dissertation are from seven benchmark suites. Table 5.1 gives their details. These programs were automatically chosen because they share two characteristics: their performance dominating loop nests cannot be successfully parallelized by parallelization techniques implemented in Liberty parallelizing compiler infrastructure, including DOALL, LOCALWRITE, DSWP and PS-DSWP. Meanwhile, although these loop nests contain parallelizable inner loops, inner loop parallelization introduces frequent barrier synchronizations limiting overall scalability. These two characteristics are required for DOMORE and SPECCROSS to have a potential benefit.

In section 5.1 and 5.2, we demonstrate the performance improvement achieved by DOMORE and SPECCROSS and discuss the applicability and scalability of both techniques. In section 5.3, we compare DOMORE and SPECCROSS with previous works, showing that through exploiting additional cross-invocation parallelism, DOMORE and SPECCROSS are

| Source suite | Benchmark program | Function | % of execution time | Parallelization plan for inner loop | DOMORE applicable | SPECCROSS applicable |
|---|---|---|---|---|---|---|
| PolyBench [53] | FDTD | main | 100 | DOALL | × | ✓ |
| | JACOBI | main | 100 | DOALL | × | ✓ |
| | SYMM | main | 100 | DOALL | ✓ | ✓ |
| OMPBench [20] | LOOPDEP | main | 100 | DOALL | × | ✓ |
| Parsec [6] | BLACKSCHOLES | bs_thread | 99 | Spec-DOALL | ✓ | × |
| | FLUIDANIMATE-1 | ComputeForce | 50.2 | LOCALWRITE | ✓ | × |
| | FLUIDANIMATE-2 | main | 100 | LOCALWRITE | × | ✓ |
| SpecFP [68] | EQUAKE | main | 100 | DOALL | × | ✓ |
| LLVMBENCH [36] | LLUBENCH | main | 50.0 | DOALL | ✓ | ✓ |
| NAS [47] | CG | sparse | 12.2 | LOCALWRITE | ✓ | ✓ |
| MineBench [45] | ECLAT | process_inverti | 24.5 | Spec-DOALL | ✓ | × |

Table 5.1: Details about evaluated benchmark programs.

| Benchmark Program | % of Scheduler/Worker |
|---|---|
| BLACKSCHOLES | 4.5 |
| CG | 4.1 |
| ECLAT | 12.5 |
| FLUIDANIMATE-1 | 21.5 |
| LLUBENCH | 1.7 |
| SYMM | 1.5 |

Table 5.2: Scheduler/worker ratio for benchmarks

able to achieve better or at least competitive performance results than previous works. Then in section 5.4, we provide a detailed case study of program FLUIDANIMATE, whose performance improvement requires the integration of both DOMORE and SPECCROSS techniques. Finally, a discussion is given in section 5.5 about the next step to be done to further improve the applicability of the Liberty parallelizing compiler infrastructure.

## 5.1 DOMORE Performance Evaluation

Six programs are chosen to evaluate the DOMORE system (as shown in Table 5.1). We compare the performance of inner loop parallelization with pthread barriers [9] with the performance of outer loop parallelization with DOMORE. Figure 5.1 shows the evaluation results for the outer loop speedup relative to the best sequential execution. For the original parallelized version with pthread barriers between inner loop invocations, none scale beyond a small number of cores. DOMORE shows scalable performance improvements

for CG, LLUBENCHMARK and BLACKSCHOLES because their scheduler threads are quite small compared to the worker threads ($< 5\%$ runtime, as shown in Table 5.3) and processor utilization is high. ECLAT, FLUIDANIMATE and SYMM do not show as much improvement. The following paragraphs provide details about those programs.

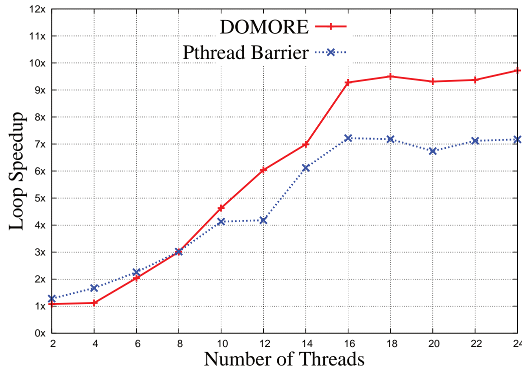
**ECLAT** from MineBench [45] is a data mining program using a vertical database format. The target loop is a two-level nested-loop. The outer loop traverses a graph of nodes. The inner loop traverses a list of items in each node and appends each item to corresponding list(s) in the database based upon on the item's transaction number. Since two items might share the same transaction number, and the transaction number is calculated non-linearly, static analysis cannot determine the dependence pattern. Profiling information shows that there is no dynamic dependence in the inner loop. For the outer loop, the same dependence manifests in each iteration (99%). As a result, Spec-DOALL is chosen to parallelize the inner loop and a barrier is inserted after each invocation. Spec-DOALL achieves its peak speedup at 3 cores. For DOMORE, a relatively large scheduler thread (12.5% scheduler/-worker ratio) limits scalability. As we can see in Figure 5.1, DOMORE achieves scalable performance up to 5 processors. After that, the sequential code becomes the bottleneck and no more speedup is achieved.

**FLUIDANIMATE** from the PARSEC [6] benchmark suite uses an extension of the Smoothed Particle Hydrodynamics (SPH) method to simulate an incompressible fluid for interactive animation purposes. The target loop is a six-level nested-loop. The outer loop goes through each particle while an inner loop goes through the nearest neighbors of that particle. The inner loop calculates influences between the particle and its neighbors and updates all of their statuses. One particle can be neighbor to multiple particles, resulting in statically unanalyzable update patterns. LOCALWRITE chooses to parallelize the inner loop to attempt to reduce some of the computational redundancy. Performance results show that parallelizing the inner loop does not provide any performance gain. Redundant computation and barrier synchronizations negate the benefits of parallelism. DOMORE is

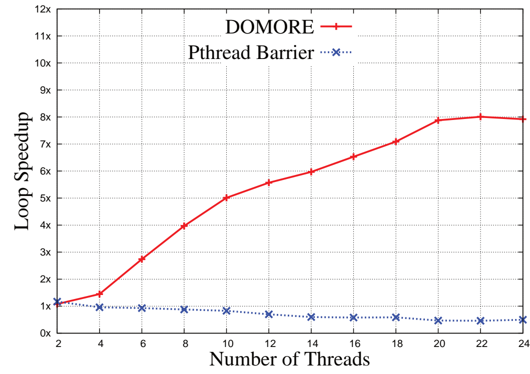
applied to the outermost loop, generating a parallel program with the redundant code in the scheduler thread and each inner loop iteration is scheduled only to the appropriate owner thread. Although DOMORE reduces the overhead of redundant computation, partitioning the redundant code to the scheduler increases the size of the sequential region, which becomes the major factor limiting the scalability in this case.

**SYMM** from the PolyBench [53] suite demonstrates the capabilities of a very simple multi-grid solver in computing a three dimensional potential field. The target loop is a three-level nested-loop. DOALL applicable to the second level inner loop. As shown in the results, even after DOMORE optimization, the scalability of SYMM is poor. The major cause is that the execution time of each inner loop invocation only takes about 4,000 clock cycles. With increasing number of threads, the overhead involved in multi-threading outweighs all performance gain.

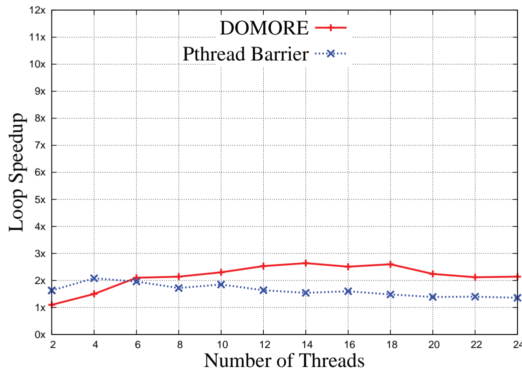
The performance of DOMORE is limited by the sequential scheduler thread at large thread counts. To address this problem, we could parallelize the `computeAddr` function. The algorithm proposed in [35] can be adopted to achieve that purpose. This will be the future work.



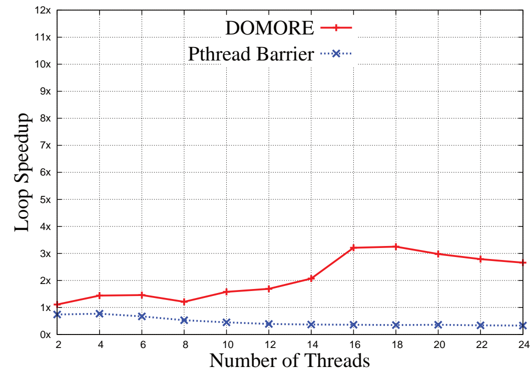

(a) BLACKSCHOLES



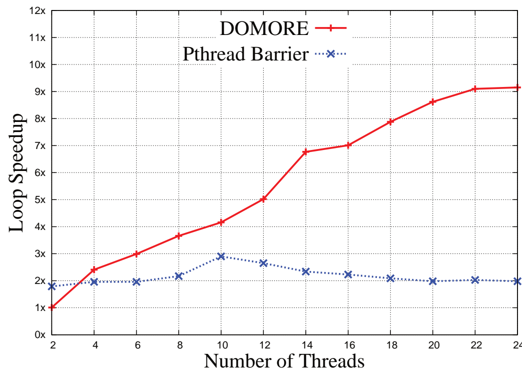

(b) CG



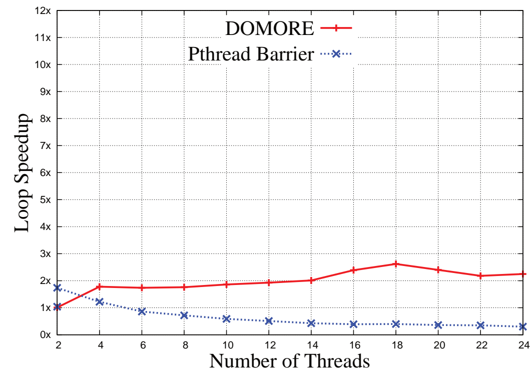

(c) ECLAT




(d) FLUIDANIMATE-1




(e) LLUBENCH




(f) SYMM

Figure 5.1: Performance comparison between code parallelized with pthread barrier and DOMORE.

## 5.2 SPECCROSS Performance Evaluation

Eight programs are evaluated for SPECCROSS. As DOMORE evaluation, We compared two parallel versions of these programs: (a) pthreads-based [9] parallelization with non-speculative pthread barriers; and (b) pthreads-based parallelization with SPECCROSS. SPECCROSS is used with a pthreads-based implementation, since the recovery mechanism relies on the properties of POSIX threads. For the performance measurements, the best sequential execution of the parallelized loops is considered the baseline.

For most of these programs, the parallelized loops account for more than 90% of the execution time. When parallelizing using SPECCROSS, each loop iteration is regarded as a separate task and the custom hash function used for calculating the access signatures keeps track of the range of memory locations (or array indices) accessed by each task. This choice is guided by the predominantly array-based accesses present in these programs. Each parallel program is first instrumented using the profiling functions provided by SPECCROSS. The profiling step recommends a minimum dependence distance value for use in speculative barrier execution. All benchmark programs have multiple input sets. We chose the training input set for profiling run. Table 5.3 shows the minimum dependence distance results for the evaluated programs using two different input sets (a training input set for profiling run and another reference input set for performance run). Four of the eight programs had runtime dependences detected by profiling functions while the rest do not. The minimum dependence distance between two inner loops in program FLUIDANIMATE varies a lot. Some of the loops do not cause any runtime access conflicts while others have a very small minimum dependence distance. For the latter case, SPECCROSS basically serves as a non-speculative barrier. The results of the profiling run were passed to speculative barrier execution which used the minimum dependence distance value to avoid misspeculation.

Figure 5.2 compares the speedups achieved by the parallelized loops using pthread barriers and SPECCROSS. It demonstrates the benefits of reducing the overhead in barrier synchronization. The best sequential execution time of the parallelized loops is considered

| Benchmark | # of tasks | # of epochs | # of checking requests | Minimum Distance | |
|---|---|---|---|---|---|
| | | | | train | ref |
| CG | 63000 | 7000 | 40609 | * | * |
| EQUAKE | 66000 | 3000 | 55181 | * | * |
| FDTD | 200600 | 1200 | 96180 | 599 | 799 |
| FLUIDANIMATE-2 | 1379510 | 1488 | 295000 | 54 / * | 54 / * |
| JACOBI | 99400 | 1000 | 67163 | 497 | 997 |
| LLUBENCH | 110000 | 2000 | 81965 | * | * |
| LOOPDEP | 245000 | 1000 | 98251 | 500 | 800 |
| SYMM | 500500 | 2000 | 369731 | * | * |

Table 5.3: Details of benchmark programs. * indicates no access conflicts are detected in profiling.

as the baseline. The original execution with pthread barriers does not scale well beyond a small number of cores. Among the eight programs in barriers, CG performs the worst since each of its epochs only contains nine iterations. With higher thread counts, the overhead caused by barriers increases without any gains in parallelism.

The speculative barrier solution provided by SPECCROSS enables all programs to scale to higher thread counts when compared to an equivalent execution with pthread barriers. At lower thread counts, pthread barrier implementation for some programs yields better performance than SPECCROSS. This happens for two reasons: (a) SPECCROSS requires an extra thread for violation detection. At lower thread counts, one fewer thread is available to do actual work which is significant when total number of threads is small; (b) the overhead of barrier synchronization increases with increasing thread counts. As a result, the effectiveness of SPECCROSS is more pronounced at higher thread counts.

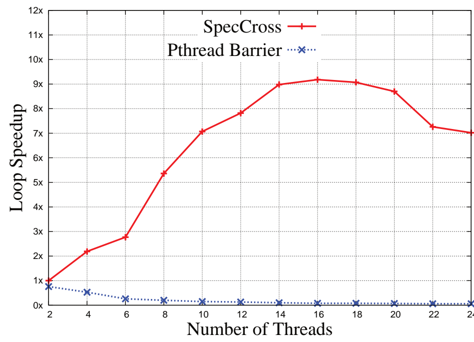
In our evaluation, the program state is checkpointed once every 1000 epochs. For the eight programs evaluated, profiling results are accurate enough to result in high-confidence speculation and no misspeculation is recorded at runtime. As a result, the operations that contribute to major runtime overheads include computing access signatures, sending checking requests, detecting dependence violation, and checkpointing.

Table 5.3 shows for each program, the number of tasks executed, the number of epochs,

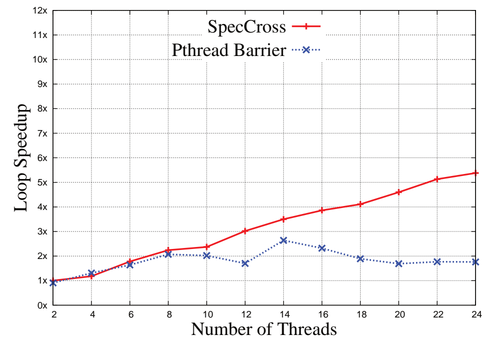
and the number of checking requests for execution with 24 threads. The performance results (Figure 5.2) indicate that with higher thread counts, the checker thread may become the bottleneck. In particular, the performance of SPECCROSS scales up to 18 threads and either flattens or decreases after that. The effects of checker thread in limiting performance can be illustrated by considering the example of LLUBENCH. The number of checking requests for LLUBENCH increases by 3.3× when going from 8 threads to 24 threads, with the resulting performance improvements being minimal. Parallelizing dependence violation detection in the checker thread is one option to solve this problem and is part of future work.

Checkpointing is much more expensive than signature calculation or checking operations and hence is done infrequently. For benchmark programs evaluated, there are less than 10 checkpoints, since SPECCROSS by default checkpoints every 1000 epochs. However, frequency of checkpointing can be reconfigured depending on desired performance characteristics. As a demonstration of the impact of checkpointing on performance, Figure 5.3 shows the geomean speedup results of increasing the number of checkpoints from 2 to 100, for all of the eight benchmark programs.

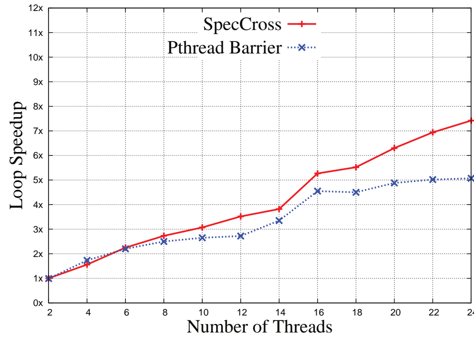
In order to evaluate the overhead of the whole recovery process, we randomly triggered a misspeculation during the speculative parallel execution. Evaluation results are shown in Figure 5.3. As can be seen, more checkpoints increases the overhead at runtime, however also reduce the time spent in re-execution once misspeculation happens. Finding an optimal configuration for them is important and will be part of the future work.



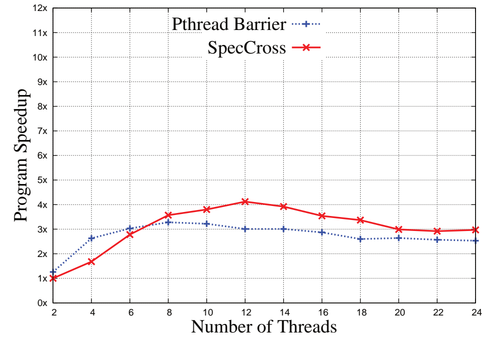

(a) CG



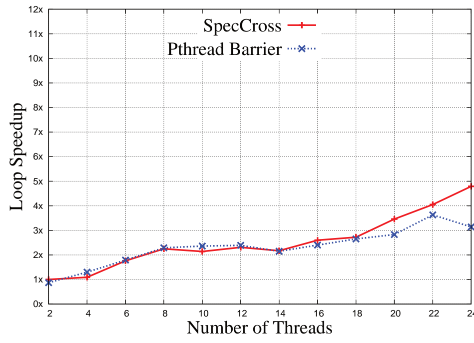

(b) EQUAKE



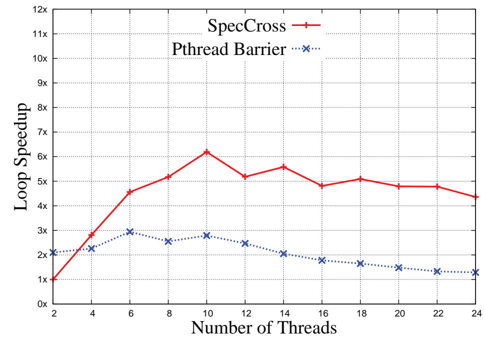

(c) FDTD



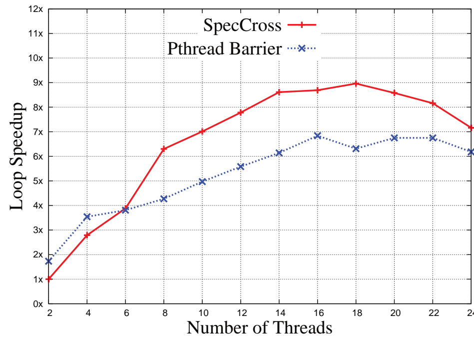

(d) FLUIDANIMATE-2



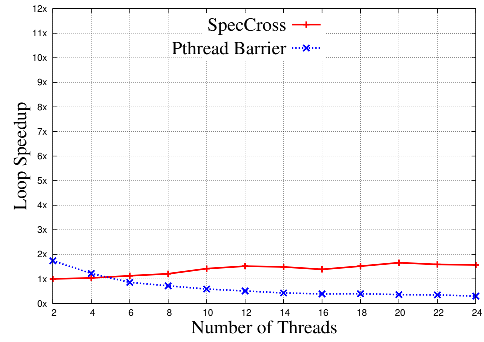

(e) JACOBI




(f) LLUBENCH




(g) LOOPDEP




(h) SYMM

Figure 5.2: Performance comparison between code parallelized with pthread barrier and SPECCROSS.

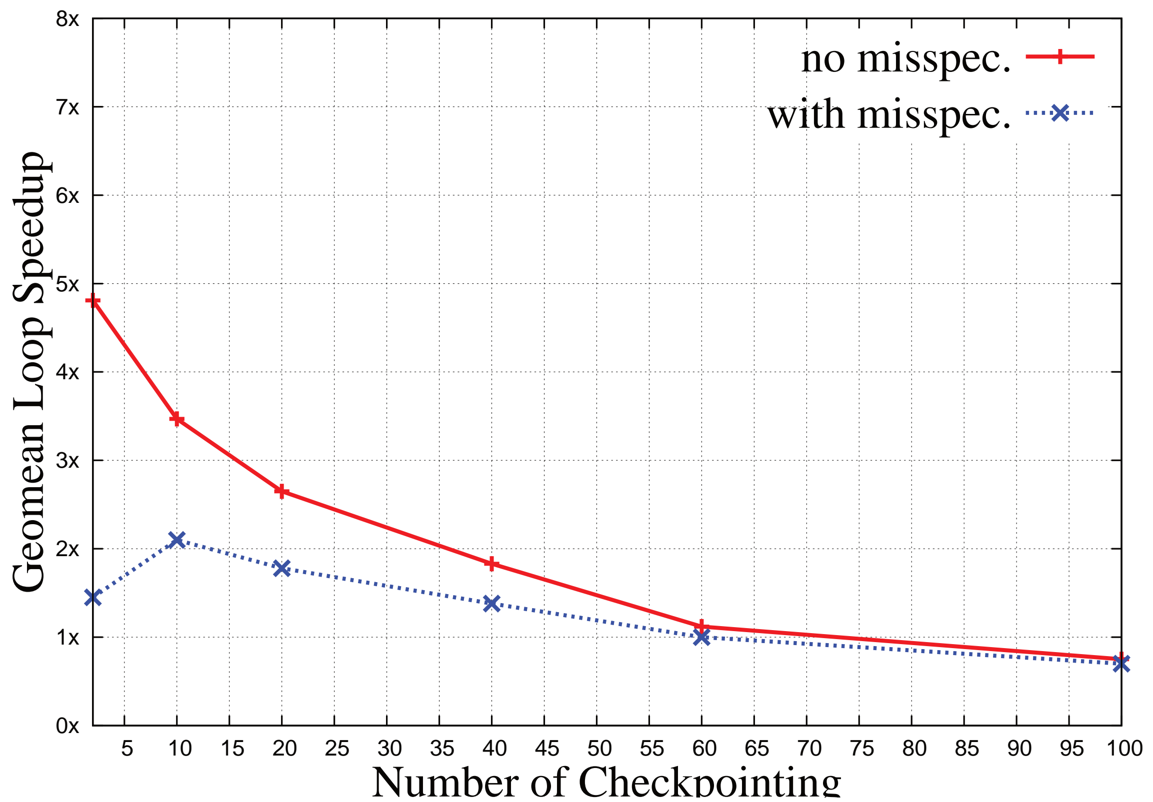



Figure 5.3: Loop speedup with and without misspeculation for execution with 24 threads: the number of checkpoints varies from 2 to 100. A misspeculation is randomly triggered during the speculative execution. With more checkpoints, overhead in checkpointing increases; however overhead in re-execution after misspeculation reduces.

## 5.3 Comparison of DOMORE, SPECCROSS and Previous Work

In this section, we compare the performance improvement achieved by DOMORE and SPECCROSS with the best performance improvement reported in previous work [6, 10, 20, 23, 28, 57]. Figure 5.4 shows the comparison results. For most programs evaluated, DOMORE and SPECCROSS achieve better, or at least competitive performance improvement.

Programs JACOBI, FDTD and SYMM were originally designed for Polyhedral optimizations [23]. They can be automatically parallelized by Polly optimizer in LLVM compiler infrastructure. Polly uses an abstract mathematical representation to analyze the memory access pattern of a program and automatically exploits thread-level and SIMD parallelism. Polly successfully achieves promising speedup for 16 out of 30 Polyhedral benchmark programs. However, it fails to extract enough parallelism for programs JACOBI, FDTD or SYMM due to the irregular access patterns of the outermost loops. Compared to Polly, SPECCROSS applies DOALL to the inner loops and exploits cross-invocation parallelism using speculative barriers, and therefore achieves much better performance.

Raman et al. manually parallelized BLACKSCHOLES using pipeline style parallelization with a multi-threaded transactional memory runtime system (SMTX [57]). They report quite scalable performance improvement on BLACKSCHOLES. DOMORE is limited by the runtime overhead at large thread counts. Potential performance improvement is possible if the scheduler thread could be parallelized.

CG and ECLAT were manually parallelized by DSWP+ technique [28]. DSWP+ performs as a manual equivalent of DOMORE parallelization. As shown in the graph, DOMORE is able to achieve close performance gain as the manual parallelization.

Both LOOPDEP and FLUIDANMIATE have a parallel implementation in their benchmark suites. We compare the best performance results of their parallel versions with the best performance achieved by DOMORE and SPECCROSS. Helix parallelized EQUAKE

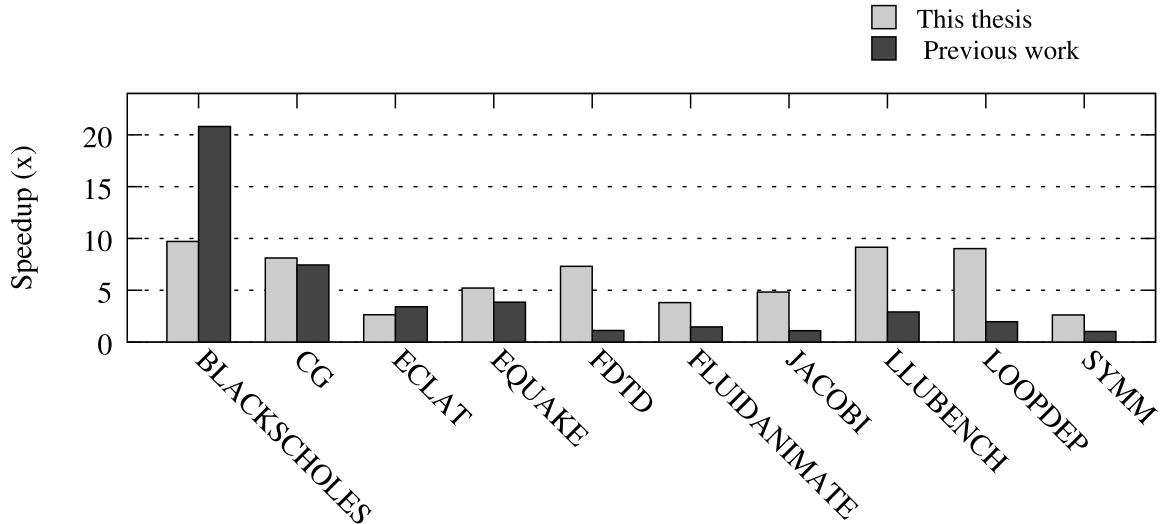



Figure 5.4: Best performance achieved by this thesis work and previous work

and its performance gain is presented in [10]. We compare the best performance reported with that achieved by DOMORE and SPECCROSS. All these three programs are parallelized in a similar way: each inner loop within the outermost loop is independently parallelized by DOALL or DOACROSS and non-speculative barrier synchronizations are inserted between inner loops to respect cross-invocation dependences. Those synchronizations limit the scalability of the performance. More details about `FLUIDANMIATE` will be given in section 5.4.

We cannot find any existing parallelization result for `LLUBENCH`, so in Figure 5.4, best performance result of DOMORE and SPECCROSS is compared against the best performance achieved by inner loop DOALL with pthread barrier synchronizations. These comparison results again demonstrate the benefits from exploiting cross-invocation parallelism.

## 5.4 Case Study: FLUIDANIMATE

Figure 5.5 shows the simplified code for the outermost loop in program FLUIDANIMATE. This outermost loop cannot be parallelized successfully by parallelization techniques sup-

```
     for (int i = 0; i < framenum; i++) {
L1:    ClearParticles();
L2:    RebuildGrid();
L3:    InitDensitiesAndForces();
L4:    ComputeDensities();
L5:    ComputeDensities2();
L6:    ComputeForces();
L7:    ProcessCollisions();
L8:    AdvanceParticles();
     }
```

Figure 5.5: Outermost loop in FLUIDANIMATE

ported in the Liberty parallelizing compiler infrastructure due to frequent manifesting cross-iteration dependences and irregular memory access patterns. The outermost loop is composed of eight consecutive inner loops. Inner loop `L4` and `L6` can be parallelized by DOANY, LOCALWRITE or DOMORE, while all the other inner loops can be parallelized by DOALL. As a result, a variety of parallelization plans can be applied.

The manually parallelized version of FLUIDANMIATE in Parsec benchmark suite divides the shared data grids among threads. It applies DOANY to inner loop `L4` and `L6` and DOALL to the other inner loops. Pthread barriers are inserted between two inner loop invocations to respect cross-invocation dependences. To protect the shared data structure, DOANY applies locks to guarantee atomic accesses. Meanwhile, to avoid over synchronization caused by locks, it allocates an array of locks instead of using a single global lock. Each lock protects a section of the shared data structure. Threads accessing different sections of the shared data do not have to synchronize at the same lock. Figure 5.6 demonstrates the performance improvement of the manual parallelization. Since the manual parallelization only supports thread number that is to the power of 2, only three performance results are shown in the figure.

Other than DOANY, inner loops `L4` and `l6` can also be parallelized by LOCALWRITE

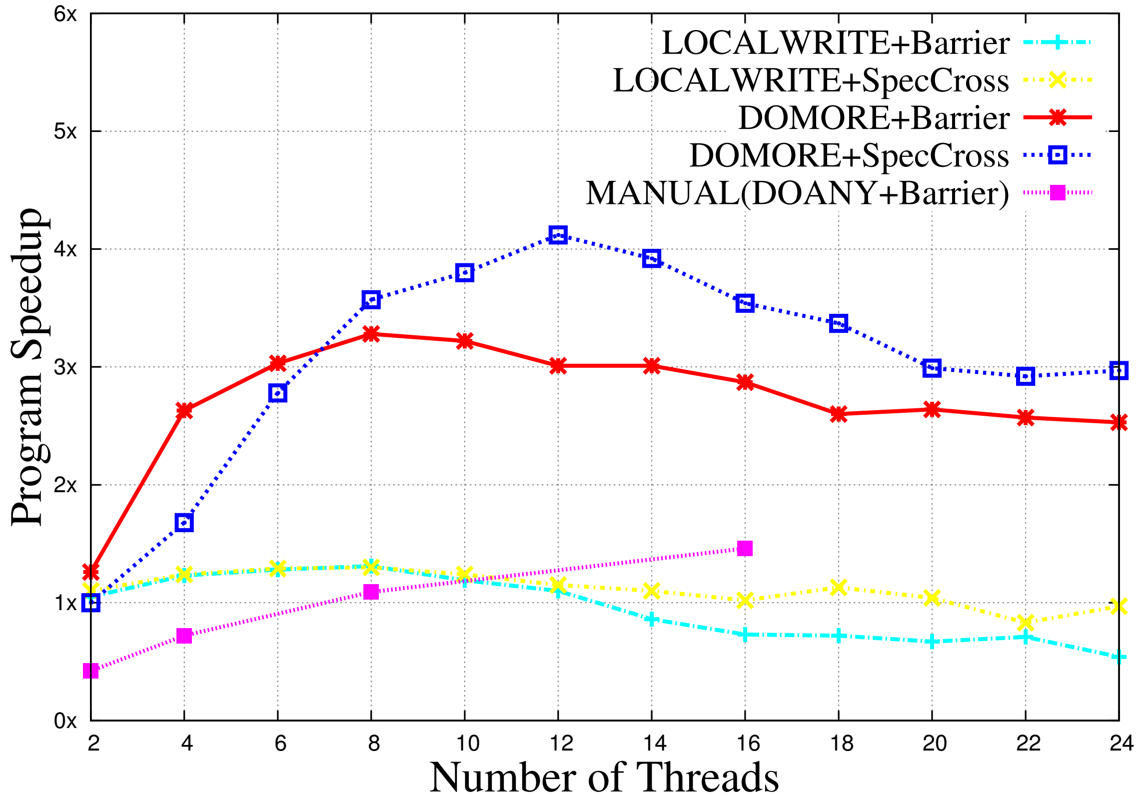



Figure 5.6: Performance improvement of FLUIDANIMATE using different techniques.

and DOMORE. We apply both of them and compare these two implementations with the manual one. As shown in Figure 5.6, DOMORE + Barriers yields the best performance among these three. DOMORE does not have the overhead in redundant computation or the overhead in locking. With small number of threads, LOCALWRITE + Barriers performs better than the manual parallelization, implying the overhead in redundant computation is less than the overhead in locking. However, since the redundancy problem deteriorates with increasing number of threads, the manual implementation scales better than the LOCALWRITE + Barrier one.

All these three parallelization implementations use pthread barriers between inner loops, prohibiting potential parallelism across loop invocations. SPECCROSS can be applied to further improve the performance. Figure 5.6 demonstrates the performance gain by using LOCALWRITE + SPECCROSS. LOCALWRITE + SPECCROSS is always better than

LOCALWRITE + Barriers because of the additional parallelism enabled by SPECCROSS. However, LOCALWRITE + SPECCROSS does not perform as well as DOMORE + Barrier. The benefits from cross-invocation parallelism is negated by the overhead in redundant computation. This also explains why with large thread counts, LOCALWRITE + SPECCROSS does not perform as well as the manual parallelization.

DOMORE is capable of reducing the redundancy execution and improving the scalability of performance. However, SPECCROSS transformation does not support partition-based parallelization technique such as DOMORE. To make SPECCROSS and DOMORE work together to achieve better performance scalability, we modify the DOMORE code generation (section 3.4). Instead of having a separate scheduler thread, the scheduler code is duplicated on each worker thread. This optimization works for FLUIDANIMATE because the duplication of scheduler code does not cause any side effect. Figure 5.6 shows the combiniation of SPECCROSS and DOMORE achieves the best performance among all.

Another interesting thing to notice is that compared to DOMORE with pthread barriers, DOMORE with SPECCROSS does not yield much better performance gain. For high confidence speculation, a speculative distance is applied to avoid conflict-prone speculation. According to the profiling results, some of the loop invocations have very small speculative range. In that case, speculative barriers basically serve as a non-speculative barrier and the effect of SPECCROSS is limited.

## 5.5 Limitations of Current Parallelizing Compiler Infrastructure

In the previous sections, we've demonstrated the applicability and scalability of DOMORE and SPECCROSS systems using ten programs. Besides these ten programs, DOMORE and SPECCROSS evaluated many other programs. Some of those programs can be directly parallelized by DOALL, DOANY or PS-DSWP [31, 32, 54], so they are not good candidates

for DOMORE and SPECCROSS. Others (e.g., GROMACS and LBM in SpecBench [68]) can potentially benefit from DOMORE and SPECCROSS if the following limitations are addressed.

SPECCROSS does not support speculative or partition-based inner loop parallelization. For some programs, the inner loops have to be parallelized using Spec-DOALL, PS-DSWP or DOMORE. To enable speculative parallelization, SPECCROSS runtime must check both inter- and intra-invocation dependences of the inner loops. This is not a trivial extension. The evaluation results already show that runtime violation checking may become the major bottleneck at large thread counts. Checking both types of dependences will further increase the workload of the checker thread and dramatically degrade the performance. Besides, SPECCROSS calculates a signature to summarize the memory access pattern in a loop iteration. Considering the intra-invocation dependences adds extra work to the signature calculation and increases the possibility of false positive. To address the first concern, one possible solution is to parallelizing the checker thread. For the other concern, a better signature mapping function, which distinguishes the intra- and inter-invocation dependences, will be essential to avoiding high runtime overhead or high false positive rate.

Supporting partition-based parallelization also adds various complexities. For example, after partition, cross-invocation dependences may unevenly distributed among stages: one stage never causes any runtime conflict while another stage causes frequent conflicts. Since SPECCROSS sets one speculative range for each loop, the latter stage will limit the potential parallelism within the former stage. Current SPECCROSS profiler does not understand code partitioning, and therefore cannot provide speculative range information per partition. Meanwhile, PS-DSWP partitioner does not take the speculative range into consideration either, thus cannot achieve an optimal partition for the programs. As a result, SPECCROSS profiler must collaborate with the partitioner to achieve a balanced partition without limiting too much potential parallelism.

# Chapter 6

# Future Directions and Conclusion

## 6.1 Future Directions

Most existing automatic parallelization techniques focus on loop level parallelization and ignore the potential parallelism across loop invocations. This limits the potential performance scalability especially when there are many loop invocations causing frequent synchronizations. A promising research direction is to extend the parallelization region beyond the scope of a single loop invocation. This thesis work takes one step forward but there are still numerous exciting avenues for future work.

Interesting research could be done in designing and implementing efficient and adaptive runtime systems for region parallelization. A limitation of both the DOMORE and SPECCROSS runtime systems is that they do not scale well enough with increasing number of threads. The overhead in violation checking and iteration scheduling ultimately becomes the performance bottleneck at high thread counts. One possible solution, as discussed above, is to parallelizing the checker thread. Instead of assigning all checking tasks to a single thread, multiple threads are used for checking concurrently. However, this optimization brings about other questions such as how to optimally allocate threads to workers and checkers. The optimal trade-off could vary significantly for different runtime environment.

An adaptive runtime system such as DoPE [58] could potentially be exploited to help the parallel execution adjust to the actual runtime environment and to achieve more scalable performance. Additionally, DOMORE and SPECCROSS have a lot of configurable parameters that are currently specified at compile time. The checkpointing frequency and the speculative range are both set in advance using profiling information. However, profiling information is not necessarily consistent with actual execution. Ideally, these parameters should also adapt to real execution (e.g., the speculative range could change to a larger value based on the actual input data set to enable more parallelism).

Current profiling techniques are not sufficient for automatic region parallelization. First, it takes too long for most profilers to generate profiling results. These profilers simply examine all dependences that cannot be broken at compile time without considering which dependences are actually of interest for parallelization techniques to be applied. For example, DOALL parallelization only cares about cross-iteration dependences while for SPECCROSS, it requires a minimum dependence distance between two iterations instead. By profiling "smartly", we could dramatically reduce the time spent on profiling and make it feasible for much larger programs. Another interesting research direction is asking the profiler to return useful information to the parallelizer, helping it achieve a better parallelization plan. For example, COMMSET [54] implements a profiler which examines the dependences and hints the programmers where to insert the annotations so that parallelization prohibiting dependences could be broken. Similarly, the profiler could tell the SPECCROSS scheduler that if a certain iteration is scheduled to another worker thread, the speculative range will be enlarged significantly to enable more potential parallelization. We envision a system in which the profiler collaborates with the parallelizer to iteratively improve the parallelization.

Other possible research directions include designing a parallelizer which can extract parallelism on whole-program scope, or evaluating approaches for effectively integrating various intra- and inter-invocation parallelization techniques to work on the same program.

We believe any of these avenues for future work could meaningfully improve the state of the art in automatic software parallelization.

## 6.2 Conclusion

Exploiting the performance of multi-core processors requires scalable parallel programs. Most automatic parallelization techniques parallelize iterations within the same loop invocation and synchronize threads at the end of each parallel invocation. Unfortunately, frequent synchronization limits the scalability of many software programs. In practice, iterations in different invocations of a parallel loop are often independent. In order to exploit additional cross-invocation parallelism, this thesis work proposes two novel automatic parallelization techniques. DOMORE and SPECCROSS synchronizes iterations using runtime information, and can therefore adapt to dependence patterns manifested by particular inputs. As a non-speculative technique, DOMORE is designed for programs with more frequently manifesting cross-invocation dependences. SPECCROSS, on the other hand, yields better performance when those dependences seldom manifest at runtime. Evaluation demonstrates that DOMORE achieves a geomean speedup of 2.1× over codes without cross-invocation parallelization and 3.2× over the original sequential performance on 24 cores. SPECCROSS achieves a geomean speedup of 4.6× over the best sequential execution, which compares favorably to a 1.3× speedup obtained by parallel execution without any cross-invocation parallelization.

# Bibliography


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

[2] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the 14th ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 1993.

[3] M. M. Baskaran, N. Vydyanathan, U. K. R. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2009.

[4] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *Proceeding of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2009.

[5] M. J. Best, S. Mottishaw, C. Mustard, M. Roth, A. Fedorova, and A. Brownsword. Synchronization via scheduling: techniques for efficiently managing shared state. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2011.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[7] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The next generation in parallelizing compilers. In *Proceedings of the 6th workshop on Languages and Compilers for Parallel Computing (LCPC)*, 1994.

[8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1995.

[9] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[10] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks. Helix: automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the 10th international symposium on Code Generation and Optimization (CGO)*, 2012.

[11] C. D. Carothers and B. K. Szymanski. Checkpointing multithreaded programs. *Dr. Dobbs*, August 2002.

[12] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd annual International Symposium on Computer Architecture (ISCA)*, 2006.

[13] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving large, irregular graph problems using adaptive work-stealing. In *Proceedings of the 37th International Conference on Parallel Processing (ICPP)*, 2008.

[14] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, volume 22, pages 161–180, 1996.

[15] R. Cytron. DOACROSS: Beyond vectorization for multiprocessors. In *Proceedings of the 11th International Conference on Parallel Processing (ICPP)*, 1986.

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

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

[18] W. R. Dieter and J. E. Lumpp Jr. A user-level checkpointing library for posix threads programs. In *Proceedings of the 29th annual international symposium on Fault-Tolerant Computing (FTCS)*, 1999.

[19] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *Proceedings of the 28th ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2007.

[20] A. J. Dorta, C. Rodriguez, F. de Sande, and A. Gonzalez-Escribano. The OpenMP source code repository. In *Proceedings of the 13th Euromicro conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2005.

[21] P. A. Emrath and D. A. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and Distributed Debugging*, 1988.

[22] R. Ferrer, A. Duran, X. Martorell, and E. Ayguadé. Unrolling loops containing task parallelism. In *Proceedings of the 21st workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2009.

[23] T. C. Grosser. Enabling polyhedral optimizations in llvm. Diploma thesis, Department of Informatics and Mathematics, University of Passau, Germany, April 2011.

[24] R. Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. In *Proceedings of the 3rd international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1989.

[25] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st annual International Symposium on Computer Architecture (ISCA)*, 2004.

[26] H. Han and C.-W. Tseng. Improving compiler and run-time support for irregular reductions using local writes. In *Proceedings of the 11th international workshop on Languages and Compilers for Parallel Computing (LCPC)*, 1999.

[27] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual International Symposium on Computer Architecture (ISCA)*, 1993.

[28] J. Huang, A. Raman, Y. Zhang, T. B. Jablin, T.-H. Hung, and D. I. August. Decoupled Software Pipelining Creates Parallelization Opportunities. In *Proceedings of the 8th international symposium on Code Generation and Optimization (CGO)*, 2010.

[29] K. Z. Ibrahim and G. T. Byrd. On the exploitation of value predication and producer identification to reduce barrier synchronization time. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS)*, 2001.

[30] T. B. Jablin, Y. Zhang, J. A. Jablin, J. Huang, H. Kim, and D. I. August. Liberty Queues for EPIC Architectures. In *Proceedings of the 8th workshop on Explicitly Parallel Instruction Computing Techniques (EPIC)*, 2010.

[31] N. P. Johnson, H. Kim, P. Prabhu, A. Zaks, and D. I. August. Speculative separation for privatization and reductions. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2012.

[32] H. Kim, N. P. Johnson, J. W. Lee, S. A. Mahlke, and D. I. August. Automatic speculative doall for clusters. In *Proceedings of the 10th international symposium on Code Generation and Optimization (CGO)*, 2012.

[33] T. Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, 1986.

[34] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd international symposium on Code Generation and Optimization (CGO)*, 2004.

[35] S.-T. Leung and J. Zahorjan. Improving the performance of runtime parallelization. In *Proceedings of the 4th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1993.

[36] LLVM Test Suite Guide.
http://llvm.org/docs/TestingGuide.html.

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

[38] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing (SC)*, 1992.

[39] J. F. Martínez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[40] P. E. Mckenney. Memory barriers: a hardware view for software hackers, 2009.

[41] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 30th ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2009.

[42] S. Moon, B. So, M. W. Hall, and B. R. Murphy. A case for combining compile-time and run-time parallelization. In *Selected papers from the 4th international workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR)*, 1998.

[43] V. Nagarajan and R. Gupta. Ecmon: exposing cache events for monitoring. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[44] V. Nagarajan and R. Gupta. Speculative optimizations for parallel programs on multi-cores. In *Proceedings of the 21st workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2009.

[45] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. *IEEE Workload Characterization Symposium*, 2006.

[46] A. Nicolau, G. Li, A. V. Veidenbaum, and A. Kejariwal. Synchronization optimizations for efficient execution on multi-cores. In *Proceedings of the 23rd International conference on Supercomputing (ISC)*, 2009.

[47] NAS Parallel Benchmarks 3.
http://www.nas.nasa.gov/Resources/Software/npb.html.

[48] C. E. Oancea and A. Mycroft. Software thread-level speculation: an optimistic library implementation. In *Proceedings of the 1st International Workshop on Multicore Software Engineering (IWMSE)*, 2008.

[49] M. F. P. O'Boyle, L. Kervella, and F. Bodin. Synchronization minimization in a SPMD execution model. *J. Parallel Distrib. Comput.*, volume 29, pages 196–210, September 1995.

[50] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th annual IEEE/ACM international symposium on Microarchitecture (MICRO)*, 2005.

[51] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, volume C-36, December 1987.

[52] R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing (SC)*, 1993.

[53] L.-N. Pouchet. PolyBench: the Polyhedral Benchmark suite. http://www-roc.inria.fr/ pouchet/software/polybench/download.

[54] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August. Commutative set: A language extension for implicit parallel programming. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2011.

[55] P. Prabhu, T. B. Jablin, A. Raman, Y. Zhang, J. Huang, H. Kim, N. P. Johnson, F. Liu, S. Ghosh, S. Beard, T. Oh, M. Zoufaly, D. Walker, and D. I. August. A survey of the practice of computational science. In *State of the Practice Reports*, SC '11, pages 19:1–19:12, 2011.

[56] R. Rajwar and J. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th international symposium on Microarchitecture (MICRO)*, 2001.

[57] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the 15th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[58] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August. Parallelism orchestration using DoPE: the Degree of Parallelism Executive. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2011.

[59] L. Rauchwerger, N. M. Amato, and D. A. Padua. A scalable method for run-time loop parallelization. *International Journal of Parallel Programming (IJPP)*, volume 26, pages 537–576, 1995.

[60] L. Rauchwerger and D. Padua. The Privatizing DOALL test: A run-time technique for DOALL loop identification and array privatization. In *Proceedings of the 8th International Conference on Supercomputing (ICS)*, 1994.

[61] L. Rauchwerger and D. Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. *ACM SIGPLAN Notices*, volume 30, pages 218–232, 1995.

[62] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in TBB. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2008.

[63] S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. *Int. J. Parallel Program.*, volume 31, pages 251–283, August 2003.

[64] J. Saltz, R. Mirchandaney, and R. Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, volume 40, 1991.

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

[66] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th annual ACM symposium on Principles of Distributed Computing (PODC)*, 1995.

[67] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. In *Proceedings of the 20th annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2008.

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

[69] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, volume 23, pages 253–300, February 2005.

[70] P. Swamy and C. Vipin. Minimum dependence distance tiling of nested loops with non-uniform dependences. In *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing (IPDPS)*, 1994.

[71] C.-W. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the 5th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPOPP)*, 1995.

[72] T. Tzen and L. Ni. Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers. *Parallel and Distributed Systems, IEEE Transactions on*, volume 4, January 1993.

[73] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE)*, 1981.

[74] M. Wolfe. Doany: Not just another parallel loop. In *Proceedings of the 4th workshop on Languages and Compilers for Parallel Computing (LCPC)*, 1992.

[75] M. J. Wolfe. *Optimizing Compilers for Supercomputers*. PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, October 1982.

[76] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th IEEE international symposium on High Performance Computer Architecture (HPCA)*, 2007.

[77] N. Yonezawa, K. Wada, and T. Aida. Barrier elimination based on access dependency analysis for openmp. In *Parallel and Distributed Processing and Applications*. 2006.

[78] J. Zhao, J. Shirako, V. K. Nandivada, and V. Sarkar. Reducing task creation and termination overhead in explicitly parallel programs. In *Proceedings of the 19th international conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.