UNIVERSITY OF MINNESOTA

This is to certify that I have examined this copy of a doctoral thesis by

Shengyue Wang

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Pen-Chung Yew	Antonia Zhai
Name of Faculty Adviser	Name of Faculty Co-Adviser
Signature of Faculty Adviser	Signature of Faculty Co-Adviser
Date	Date

GRADUATE SCHOOL

Compiler Techniques for Thread-Level Speculation

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

SHENGYUE WANG

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Pen-Chung Yew, Adviser

Antonia Zhai, Co-Adviser

March 2007

©Shengyue Wang 2007

Acknowledgments

First, I am extremely grateful to my adviser, Professor Pen-Chung Yew and co-adviser, Professor Antonia Zhai, for their invaluable guidance, enourmous patience and unwavering supports in all respects during my PhD studies. They have always been generous with their time and advice. Without their help and encouragements, I will never reach this final destination.

I would also like to thank Professors Wei-Chung Hsu and David J. Lilja for their generous help, encouragements and suggestions, for serving as members of my examination committee.

I would also like to thank my fellow colleagues: Tong Chen, Xiaoru Dai, Jinpyo Kim, Jin Lin, Guojin He, Venkatesan Packirisamy and Kiran S. Yellajyosula for their generous help and friendship.

Finally I thank my parents and in-laws, my wife Xuerong Wen for their unconditional love and support over these years.

Abstract

Thread-Level Speculation (TLS) allows potentially dependent threads to execute in parallel, by postponing the detection and verification of inter-thread data dependences until runtime. Although TLS greatly simplifies automatic parallelization, only moderate performance improvements have been achieved for general-purpose applications. Thus adequate compiler techniques must be developed to fully utilize the underlying TLS support. In this thesis, we address the key compiler issues involved in the speculative parallelization and propose several optimization techniques to improve the performance of general-purpose applications under TLS.

One important task of a TLS compiler is to identify speculative threads with good performance potentials. We propose a loop selection algorithm that determines which loops to parallelize to maximize the program performance. For applications with limited loop-level parallelism, a partitioning algorithm is proposed to extract threads from sequential codes.

To further enhance the performance of parallel threads, we propose three compiler optimizations: instruction scheduling, reduction transformation and iteration

merging. Instruction scheduling improves the efficiency of synchronizing frequently occurring memory dependences; reduction transformation reduces the impact of a class of reduction variables whose intermediate results are used by non-reduction operations; iteration merging improves the load balancing by dynamically combining loop iterations to achieve more balanced workloads.

All the proposed techniques are implemented in a TLS compiler framework built on the Intel's Open Research Compiler (ORC). We achieve an average program speedup of 1.38 for the SPEC2000 benchmarks. Our experimental results show that, in the context of TLS, adequate compiler support is crucial in delivering the desirable performance for general-purpose applications, and our proposed techniques are effective in exploiting speculative parallelism.

Contents

1 Introduction **1**

1.1 Related Work 2

1.2 Background on TLS 6

1.3 Dissertation Contributions 8

2 TLS Compilation Framework **11**

2.1 Thread Extraction 12

2.2 Thread Optimization 15

2.3 Profiling Support 16

2.3.1 Loop Nesting Profiling 16

2.3.2 Data Dependence Profiling 17

2.3.3 Edge Profiling 19

3 Loop Selection **20**

3.1 Related Work 21

3.2 Loop Selection Algorithm 23

3.2.1 Loop Graph 23

3.2.2 Selection Criterion 25

3.2.3 Loop Selection Problem 26

3.2.4 Graph Pruning 27

3.2.5 Exhaustive Searching Algorithm 29

3.2.6 Heuristic-based Searching Algorithm 30

3.3 Loop Speedup Estimation 31

3.3.1 $T_{misspec}$ Estimation 33

3.3.2 T_{syn} Estimation I 35

3.3.3 T_{syn} Estimation II 36

3.3.4 T_{syn} Estimation III 38

4 Non-Loop Partitioning **39**

4.1 Related Work 40

4.2 Algorithm Features 41

4.3 Partitioning Algorithm 44

4.3.1 Compact CFG 44

4.3.2 Frequent Execution Path 46

4.3.3 Recursive Partitioning 47

4.3.4 Performance Estimation 48

4.3.5 Sub-Path Expansion 51

5 Thread Optimizations **52**

5.1 Related Work 53

5.2 Speculative Scheduling for Memory Dependence 54

5.2.1 Intra-Thread Speculation 56

5.2.2 Working with TLS 62

5.3 Reduction Transformation 64

5.3.1 Reduction Elimination 65

5.3.2 Partial Reduction Elimination 66

5.3.3 Aggressive Reduction Transformation I 68

5.3.4 Aggressive Reduction Transformation II 69

5.3.5 Transformation Algorithm 70

5.4 Iteration Merging for Load Balancing 74

5.4.1 Iteration Merging 76

5.4.2 Transformation Algorithm 77

5.5 Case Study with Compression Applications 79

6 Performance Evaluation **83**

6.1 Simulation Framework 83

6.2 Benchmark Description 84

6.3 Loop Selection 85

6.3.1 Statistics for the Selected Loops 86

6.3.2 Performance Impact of Loop Speedup Estimation 90

6.4 Non-loop Partitioning 92

6.5 Impact of Instruction Scheduling for Memory Dependence 96

6.6 Impact of Reduction Transformation 101

6.7 Impact of Iteration Merging 103

6.8 Program Speedup 104

7 Conclusions **106**

List of Tables

6.1 Simulation parameters. 84

6.2 Loop statistics. 85

6.3 Loop statistics of estimation I. 87

6.4 Loop statistics of estimation II. 88

6.5 Loop statistics of estimation III. 89

6.6 Loop statistics of perfect estimation. 90

6.7 Non-loop thread statistics. 93

6.8 Loops for reduction transformation. 102

List of Figures

2.1 A TLS compiler framework. 12

2.2 Examples of loop threads and non-loop threads. 13

3.1 Example of loop graph. 23

3.2 Example of loop graph with recursion. 24

3.3 Loop graph before pruning. 27

3.4 Loop graph after pruning. 29

3.5 Impace of delay D assuming 4 processors. 31

3.6 Cost of mis-speculation. 34

3.7 Cost of synchronization. 35

4.1 Non-loop partitioning algorithm applied to each procedure. 42

4.2 Bipartition algorithm. 43

4.3 Examples of compact CFG. 45

5.1 Impact of instruction scheduling. 56

5.2 Data speculation used in scheduling for memory dependence. 57

5.3 Control speculation used in scheduling for memory dependence. . . 60
5.4 Cascaded speculation with recovery support. 61
5.5 Speculative forwarding. 63
5.6 Reduction elimination. 65
5.7 Reduction variable with an additional use. 66
5.8 Using the intermediate result of a reduction variable to determine a branch outcome. 69
5.9 Algorithm for reduction transformation. 72
5.10 Load imbalance. 75
5.11 Iteration merging. 76
5.12 Algorithm for iteration merging 78

6.1 The program speedup of loop threads. 91
6.2 The coverage of loop threads. 92
6.3 The program speedup of non-loop threads. 93
6.4 The coverage of non-loop threads. 94
6.5 The prediction accuracy of non-loop thread. 94
6.6 The performance impact of thread prediction. 95
6.7 The performance impact of instruction scheduling for memory dependence on the loop threads. 97
6.8 The performance impact of instruction scheduling for memory dependence on the non-loop threads. 98

6.9 The performance impact of different instruction scheduling strategies on the loop threads. 99

6.10 The performance impact of recovery code on the loop threads optimized by using the most aggressive instruction scheduling strategy. 100

6.11 The performance impact of reduction transformation on the loop threads. 103

6.12 The performance impact of iteration merging on the loop threads. . 104

6.13 The program speedup achieved from both loop and non-loop threads. 104

Chapter 1

Introduction

Due to the increasing difficulty of improving microprocessor's performance by exploiting Instruction-Level Parallelism (ILP) within a single thread of execution, microprocessors that support multiple threads of execution are becoming increasingly common [34, 37, 32, 31, 4, 24, 30]. By allowing multiple threads to run in parallel, such multithreaded processors are effective in exploiting more coarse-grained Thread-Level Parallelism (TLP). One attractive method for sequential programs to benefit from the increasing computing resources provided by multithreaded processors is to partition them into multiple threads of execution. However, the parallelization of sequential programs, either manually by a programmer or automatically by a compiler, is difficult because of pointer aliasing, irregular array accesses, and complex control flow. Thread-Level Speculation (TLS) facilitates the parallelization of sequential programs [36, 2, 22, 13, 19, 57, 25, 26, 39, 44, 47, 59, 62, 14]

by allowing potentially dependent threads to execute in parallel, while the sequential semantics of the original programs are preserved through runtime checking.

Although TLS greatly simplifies the automatic parallelization process and has shown good performance potential [60, 47], only moderate performance improvements have been achieved for general-purpose applications [18, 70, 42]. Thus adequate compiler techniques must be developed to fully utilize the underlying TLS support. This calls for a deep understanding of performance bottlenecks and a study of compiler techniques to fully explore the optimization opportunities exposed by TLS.

This dissertation addresses several important compilation issues when sequential programs are speculatively parallelized for TLS. It investigates the compiler techniques that can identify and extract speculative threads with good performance potentials, and the optimization techniques that can further enhance the performance of speculative threads. With these techniques, speculative parallelism can be effectively identified and extracted from those challenging general-purpose applications.

1.1 Related Work

Over the last decade, TLS has become an active research area, where numerous schemes have been proposed for exploiting such speculative parallelism in sequential programs.

Multiscalar processor [57] was the pioneer work that introduced the idea of TLS and provided complete hardware mechanism called Address Resolution Buffer (ARB) [21] for dynamically enforcing memory dependences.

TLS was then popularized and has been extensively studied on both Chip Multiprocessor (CMP) [23, 26, 60, 38, 44, 62, 59, 14, 12] and Simultaneous Multi-threaded Processor (SMT) [2, 49, 48]. Most of these follow-up studies, especially those on CMP processors, rely on the extended shared memory cache coherence schemes to support memory dependence speculation. Such cache-based schemes typically show better scalability compared to the centralized ARB, and are more appropriate for exploiting parallelism in larger threads.

Transactional memory [28, 53, 27, 5, 54, 45] follows the similar idea of speculative execution to optimistically execute parallel programs without explicitly inserted synchronizations. A transaction is an atomic unit of execution. It commits if it finishes without a data conflict. Otherwise, it aborts and re-executes.

Along with these hardware innovations in supporting efficient TLS, there are several research work done on compiler support for TLS [65, 6, 69, 70, 18, 33, 52, 42].

Multiscalar processor relies on the compiler to partition an entire sequential program into multiple tasks (threads), each of which corresponds to a sequence of continuous basic blocks in the Control Flow Graph (CFG) [1]. In order to maximize the performance of resulting threads, the partitioning algorithm [65] implemented

in the compiler uses a variety of heuristics based on the information such as data dependence, thread predictability, load imbalance, and so on.

This thread extraction method was later improved by Johnson et al. [33] who modeled thread partitioning as a min-cut problem. By applying a sequence of balanced min-cuts to the CFG annotated with weights, the improved method considers all performance-related factors simultaneously and generates threads with a better performance.

Also in the multiscalar compiler [65], several compiler-aided performance enhancement techniques were proposed. Among them, instruction scheduling for improving inter-thread register value communication was one important technique that inspired several other investigations including this thesis work.

The compiler work presented by Bhowmik et al. [6] extensively uses data and control dependence profiling during task selection. It supports flexible thread spawning policy such as out-of-order spawning, a feature that is later demonstrated to be very expensive to implement in the hardware [55].

Mitosis compiler [52] extracts threads with an emphasis on thread predictability by exploring control independence in the program. In order to deal with inefficiency caused by inter-thread data dependences, the compiler generates pre-computation slices that are executed ahead of speculative threads to speculatively compute live-in values.

Du et al. [18] present a cost-driven compilation framework to select loops for

speculative parallelization. They build a cost graph from the control flow and data dependence graphs to estimate the mis-speculation cost. The estimated mis-speculation cost, along with a set of other criteria such as the size of iteration and the number of iterations, are used to determine which loops in a program should be speculatively parallelized.

POSH compiler [42] is designed to exploit parallelism from both loops and procedures. A large set of loops and procedures are first selected by the compiler. A performance profiler is then invoked to execute the program. The performance information collected by the profiler is feed back to the compiler such that any initially selected parallel threads with poor performance will be discarded. Although some hard-to-predict runtime behaviors such as cache performance can now be estimated by the performance profiler, it also introduces significant overheads to thread extraction.

Zhai et al. [69, 70] showed the importance of improving inter-thread value communication. In their study, they found that for frequently occurring inter-thread data dependences, synchronizations have to be used in order to avoid excessive speculation failures. However, synchronization may cause significant stalls in the consumer thread for the dependent data to become available. To address this problem, instruction scheduling algorithm based on data-flow analysis was proposed to reduce the length of critical forwarding path introduced by synchronizing accesses to scalar variables.

Finally, automatic parallelization techniques have been extensively studied for over 20 years [68, 35, 29, 7, 61]. The main limitation of these traditional compilation techniques is that the compiler has always to be conservative in order to generate the correct code that works in all cases. That is, a compiler has to synchronize all possible inter-thread dependences, even though some of them do not or rarely exist. Speculation allows a compiler to make optimistic assumptions about the ambiguous information that cannot be statically determined. By speculating on those infrequently occurring data dependences, a TLS compiler can exploit more parallelism beyond the dependence limitation imposed on the traditional compilers. Consequently, the amount of parallelism that can be extracted is no longer limited by the worst-case assumption.

1.2 Background on TLS

In TLS, the compiler partitions a program into speculative threads without having to prove that they are independent, while at runtime the underlying speculation mechanism checks whether inter-thread data dependences are preserved and re-executes any thread that violates the dependence constraint. By speculating on ambiguous dependences between threads, this TLS execution model allows the parallelization of programs that were previously non-parallelizable.

Unlike traditional non-speculative parallelization, the threads running under TLS have to follow the sequential order imposed by the original program seman-

tics. During execution, the earliest thread in the program order is treated as non-speculative while the others are speculative.

Although a variety of TLS implementations exist, a typical speculative multi-threaded processor includes the following components:

(1) *Speculative buffering* Since a speculative thread may be squashed due to possible data dependence violations, all the data stored by a speculative thread are unsafe and need to be buffered before they are allowed to be committed to the safe storage.

(2) *Violation detection* In order to detect data dependence violation at runtime, all exposed loads from a speculative thread need to be tracked by predecessor threads. A violation is detected when a predecessor thread stores to a variable whose value has already been loaded by a following speculative thread.

(3) *Recovery support* After a mis-speculation is detected, the violating thread has to be squashed and re-executed, that is, all speculatively modified data are invalidated and the thread restarts execution from its beginning. All successors of the violating thread are squashed and re-executed as well, since they may also have inconsistent states.

(4) *Data commit* A speculative thread is allowed to commit its modified data to the safe storage only after the point at which all its speculatively loaded value are verified to be correct and all its predecessor threads have committed their data. Such in-order commit is necessary for maintaining the sequential program

semantics.

1.3 Dissertation Contributions

TLS makes it possible to parallelize hard-to-parallelize sequential programs, thus exposing new opportunities to the parallelizing compiler. In this dissertation, we identify the key issues involved when TLS support is available in parallelizing general-purpose applications and propose several compiler techniques to exploit the potential of TLS. To make the investigation concrete, the proposed compiler techniques are implemented on the Intel's Open Research Compiler (ORC) [20], and their impacts are evaluated using the SPEC2000 benchmark suite.

In the area of compiler support for TLS, this dissertation makes the following contributions:

1. We propose a loop selection algorithm that determines which loops to parallelize in a program that contains a larger number of nested loops. Based on a complete set of information to estimate expected performance, this selection algorithm can identify a set of loops with good performance potentials, and the parallelization of those selected loops could maximize the overall program performance.

2. We propose a program partitioning algorithm to extract non-loop threads from sequential codes. For general-purpose applications, extracting loop

threads may not achieve adequate amount of parallelism due to complex control flow and hard-to-optimize data dependences. It is desirable to search for parallelism beyond loop threads. This partitioning algorithm allows us to exploit parallelism from those applications that have limited loop-level parallelism.

3. We extend the instruction scheduling algorithm proposed by Zhai et al. [69] to improve the efficiency of synchronizing frequently occurring memory dependences. Both intra-thread data and control speculation are used to enable more aggressive instruction scheduling. The scheduling algorithm is further enhanced by generating intra-thread recovery code to reduce the mis-speculation cost. Our experiments show that instruction scheduling is effective in reducing the length of critical forwarding path introduced by the synchronization of memory dependences, and the parallel performance is greatly improved for most of benchmark programs.

4. We propose an aggressive reduction transformation algorithm to reduce the length of critical forwarding path caused by a class of reduction variables. Such variables are defined in the loop body through reduction operations, but there also exist uses of their intermediate results, thus it is impossible to apply traditional reduction optimization [35] to eliminate the inter-thread data dependences caused by those variables. Through aggressive transformation, our proposed algorithm is effective in reducing the impact of such reduction

variables and greatly improves speedup for the loops whose performances are limited by those reduction variables.

5. We propose a loop transformation technique called iteration merging to improve the load balancing for speculative threads. Workload imbalance could cause significant performance degradation in TLS, since a speculative thread cannot commit until all its predecessors finish, even if its execution has completed a long time back. The proposed technique solves the load imbalance problem by dynamically combining multiple short loop iterations with a long one to achieve more balanced workloads during parallel execution.

The rest of dissertation is organized as follows. In Chapter 2, we describe our TLS compilation framework. Chapter 3 and Chapter 4 discuss the loop selection algorithm and the non-loop partitioning algorithm respectively. Chapter 5 describes three thread optimization techniques. Chapter 6 presents the performance evaluation. The conclusions and possible future work are presented in Chapter 7.

Chapter 2

TLS Compilation Framework

In this chapter, we present the compiler framework that we have derived for parallelizing general-purpose applications under TLS. This framework is built on Intel's ORC compiler. Using the ORC compiler allows us to take advantage of the state-of-the-art compilation technology to generate high-quality multithreaded code. Most passes in our framework are implemented in the Code Generation (CG) phase, the last phase in the ORC compiler, to avoid the interference with other existing optimizations.

Figure 2.1 shows the overall structure of the framework. It is composed of three major parts: *Thread Extraction*, *Thread Optimization*, and *Profiling Support*. The thread extraction phase is responsible for identifying potential sources of parallelism in sequential programs and breaking them into multiple threads. The resulting threads are then optimized in the thread optimization phase to en-

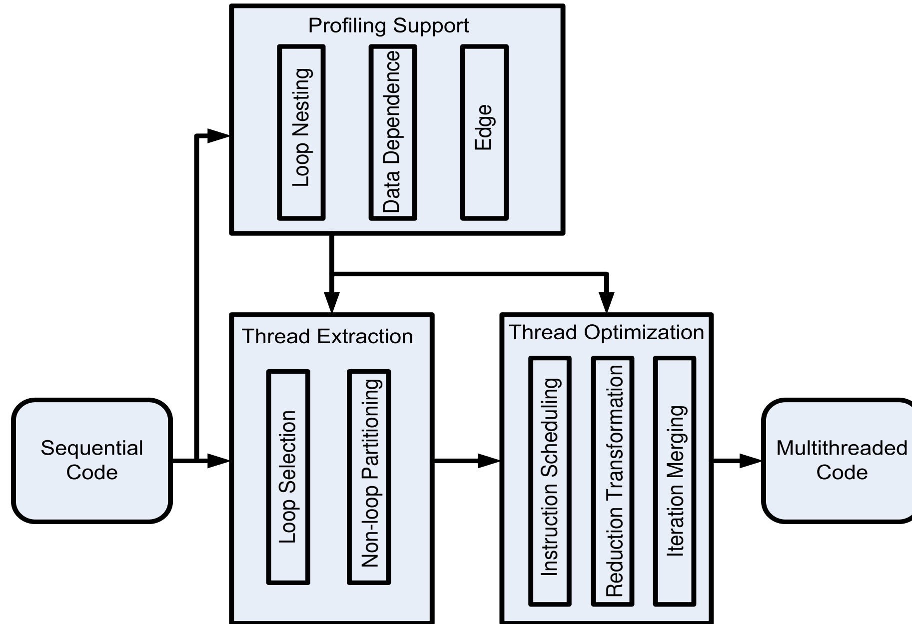
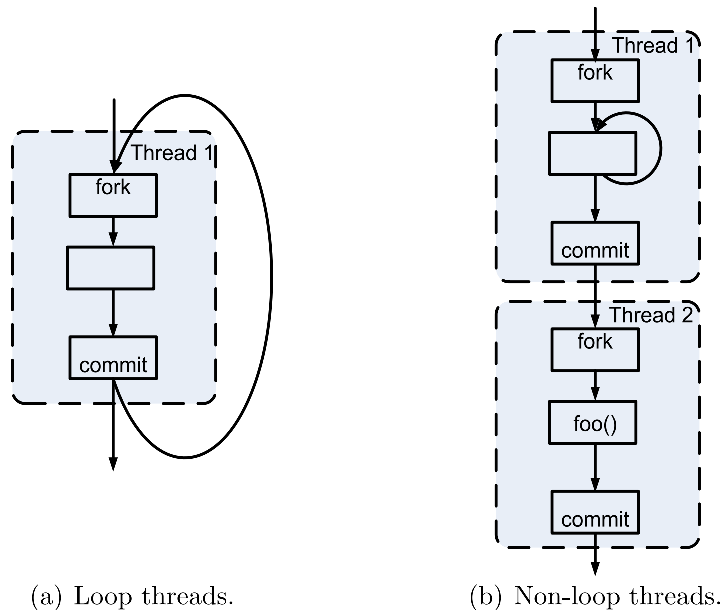


Figure 2.1: A TLS compiler framework.

hance the parallel performance. Both the thread extraction phase and the thread optimization phase rely on the profiling support to gather information that is hard to obtain from the static compiler analysis, such as data dependence probability.

2.1 Thread Extraction

One of the most important tasks of a TLS compiler is thread extraction, that is, to identify potential sources of speculative parallelism in programs and break them into multiple threads. The parallel performance is largely determined by how threads are extracted. For instance, extracting threads that have too many inter-thread data dependences may lead to significant performance degradation due to



(a) Loop threads.

(b) Non-loop threads.

Figure 2.2: Examples of loop threads and non-loop threads.

frequent inter-thread data communication.

Loops are the traditional targets for creating threads where each loop iteration is executed as a separate thread. We call such loop iteration-based threads *loop threads*. Figure 2.2(a) shows the CFG of a typical loop and the corresponding loop threads. A *fork* instruction is inserted at the beginning of the loop thread. When it executes, a new thread is spawned for the next loop iteration. A *commit* instruction is inserted at the end of the loop thread. Note that, since a speculative thread is not allowed to commit its modified data, the execution of a commit instruction by a speculative thread will stall the current thread until it becomes non-speculative.

Loop threads works well for those scientific applications where loops with sim-

ple control flow and regular data access have significant coverage. However, in general-purpose applications, the loops often have complex control flow and hard-to-optimize data dependences, so that extracting loop threads from those applications alone may not deliver sufficient amount of parallelism. To further exploit parallelism beyond loops, we examine the remaining parts of the program to identify good code portions that can be further partitioned into threads. Since those threads are not created from loop iterations, we call them *non-loop threads*. Figure 2.2(b) shows the CFG for a piece of sequential code that contains a loop followed by a procedure call *foo()*. Assume that parallelizing the loop does not achieves good performance. An alternative choice for parallelizing this piece of code is to partition it into two threads, one contains the loop and the other one contains the procedure call, as shown in Figure 2.2(b). If these two threads are highly independent, good performance can be achieved by running them in parallel.

Consequently, our thread extraction phase is divided into two parts: *Loop Selection* and *Non-loop Partitioning*. As shown in Figure 2.1, loop selection is invoked first to identify loops with good performance potential and decompose them into loop threads. Non-loop partitioning is then invoked to search for parallelism in the rest of the program and partition them into non-loop threads. We consider loop threads as the first priority during thread extraction since loops are the essential sources of parallelism and their iterative structures are amenable to parallel threads by nature. In contrast, extracting non-loop threads requires more sophis-

ticated partitioning algorithms and is applied when loop threads are not adequate to achieve good performance.

Loop selection and non-loop partitioning will be further discussed in Chapter 3 and Chapter 4 respectively.

2.2 Thread Optimization

The goal of the thread optimization phase is to improve the parallel performance of the extracted threads. It contains three optimizations: *Instruction Scheduling*, *Reduction Transformation*, and *Iteration Merging*, as shown in Figure 2.1.

The first two techniques, instruction scheduling and reduction transformation, are used to improve inter-thread communication caused by data dependences. Instruction scheduling is a general technique targeting those data dependences that limit the parallel performance, while reduction transformation is a more specific technique targeting those data dependences caused by reduction variables.

The last technique, iteration merging, is aimed to improve load balance among loop threads. Due to complex control flow in general-purpose applications, different loop threads extracted from the same loop may take different execution paths at runtime. This may result in dramatic variation in the thread size. Those loop threads with severe load imbalance problem are further transformed during iteration merging in order to achieve more balanced workloads.

The details about these three optimization techniques will be discussed in Chap-

ter 5.

2.3 Profiling Support

TLS, like other speculative mechanism, is only beneficial only when it is highly likely to succeed. This requires the compiler to have precise knowledge about programs to apply speculation wisely. Such knowledge, however, is often difficult to obtain at compiler time. Although it is possible to design complicated compiler analysis to obtain more precise information, profiling is a more efficient approach to collect those information at runtime. Therefore, it is extensively used in our compiler. As shown in Figure 2.1, there are three types of profiling supports: *Loop Nesting Profiling*, *Data Dependence Profiling*, and *Edge Profiling*. The collected profiles are used in both thread extraction phase and thread optimization phase.

2.3.1 Loop Nesting Profiling

Loop nesting profiling is used to aid loop selection. In order to avoid selecting loops that are nested, our loop selection needs to know whether any two loops are nested or not. While the loop nesting relation can be easily determined locally within a procedure, it is difficult to determine nesting relation globally at the program level since two loops may become nested through procedure call.

Loop nesting profiling provides precise loop nesting information by dynamically tracking the execution sequence of all loops in a program. Anytime a loop is invoked

while another loop is still active, a nesting relation is detected. The profiling results are summarized in a graph called *loop graph*, where each static loop is represented as a node, and the nesting relation between two loops is represented as an edge. Two loops are directly nested if there is no other loop levels in between. For simplicity, only direct nesting relations are represented in the loop graph. Any indirect nesting relation can be identified by computing the transitive closure of the loop graph. More details about loop graph will be discussed when the loop selection algorithm is presented in Chapter 3.

2.3.2 Data Dependence Profiling

Data dependence profiling is used to collect more precise dependence information. Compiler static analysis usually only tells us whether a data dependence between two memory operations exists or not. Without knowing how frequently a dependence will occur at runtime, the compiler may speculate on a frequent data dependence resulting in excessive mis-speculations. Such mis-speculation is very costly in TLS because the violating thread and all the following threads need to be squashed and re-executed.

As a result, the most important information gathered by dependence profiling is dependence probability. There are two types of dependence profilings supported in our framework. One is *procedure-level* dependence profiling, that is, given a specific procedure is executed, we want to know how likely a dependence between two

memory operations will occur. Here, the probability is approximated as the number of procedure invocations in which the dependence occurs divided by the total number of invocations of that procedure. The other one is *loop-level* dependence profiling, that is, given a loop iteration is executed, we want to know how likely a dependence between two memory operations will occur. In this case, the probability is approximated as the number of loop iterations in which the dependence occurs divided by the total number of iterations of that loop.

Along with dependence probability, we also collect dependence distance information in the loop-level dependence profiling. The dependence distance is the number of iterations between two dependent memory operations. The dependences with distance greater than the number of targeting processors are discarded since threads with such distance will not be executed simultaneously due to resource constraints and those dependences are always satisfied during execution.

Since collecting profiles for each procedure and loop is a very time-consuming process, we only focus on *true* dependence during profiling. Both *anti* and *output* dependences are ignored because they can often be removed by various renaming techniques. Moreover, sampling techniques are used to further reduce the profiling cost [11]. The details of these techniques are beyond the scope of this study and will not be discussed in this thesis.

2.3.3 Edge Profiling

Edge profiling provides detailed information about how often an control flow edge in CFG is executed. These information is used frequently in almost all compilation phases. For instance, our non-loop partitioning is applied to the frequently executed paths that are identified by using edge profile. Our instruction scheduling also relies on edge profile to speculatively schedule instructions along frequent paths.

Edge profiling has already been supported by the ORC compiler for aggressive optimizations. It can be easily extended to our TLS compiler framework.

Chapter 3

Loop Selection

Loops are attractive candidates for extracting parallel threads, as programs spend a significant amount of time executing instructions within loops, and the regular structure of loops makes it relatively easy to determine (i) the beginning and the end of a thread (i.e., each iteration corresponds to a single thread of execution) and (ii) the inter-thread data dependences. Thus it is not surprising that most previous research on TLS has focused on exploiting loop-level parallelism [63, 44, 10, 18, 69, 70, 66, 42].

However, general-purpose applications typically contain a large number of potentially nested loops, and thus deciding which loops should be parallelized for the best program performance is not always clear. We have found $10,612$ loops from 15 benchmarks in the SPEC2000 benchmark suite; among these, *gcc* contains more than $2,500$ loops. Thus it is necessary to derive a systematic approach to

automatically select loops from these applications for parallelization.

It is difficult for a compiler to determine whether a loop can speed up under TLS, as the performance of the loop depends on (i) the characteristics of the underlying hardware, such as thread creation overhead, inter-thread value communication latency, and mis-speculation penalties, and (ii) the characteristics of the parallelized loops, such as the size of iterations, the number of iterations, and the inter-thread data dependences. While detailed profiling information and complex estimations can potentially improve the accuracy of estimation, it is not clear whether these techniques will lead to an overall better selection of loops.

Furthermore, loop selection is limited by the constraints that two loops cannot be parallelized simultaneously if they become nested during execution. We say that loop B is nested within loop A when loop B is syntactically nested within loop A, or when A invokes a procedure that contains loop B. On average, we observe that the SPEC2000 benchmarks have a nesting depth of 7.5. Therefore a judicious decision must be made in order to select the proper loops for parallelization.

3.1 Related Work

A number of studies on TLS use loops as the primary source of speculative TLP. Some of them extract loop threads dynamically at runtime [63, 44, 10], while others rely on the compiler to statically extract loop threads [18, 69, 70, 42].

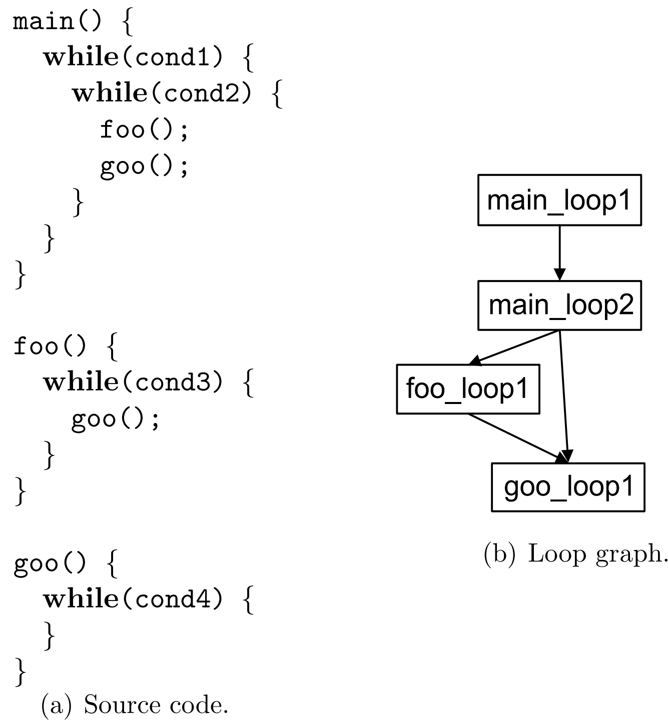
Due to the high cost at runtime, some studies of dynamic thread extrac-

tion [63, 44] only focued on inner loops, which typically have low coverage for general purpose applications.

Chen et al. [10] have proposed a dynamic loop selection framework for the Java program. They use hardware to extract useful information such as dependence timing and speculative buffering requirements and then estimate the speedup for loops based on these dynamically collected information. The main limitation of dynamic thread extraction is the lack of high level knowledge of programs.

Liu et al. [42] propose a selection method that relies on a performance profiler to estimate the parallelism of pre-selected loops, and the loops with poor performance are discarded by the compiler during final thread generation. The performance profiler is mainly used for estimating the impact of cache performance that is hard to predict at compiler time. However, the cost associated with such profiler is still the major concern during loop selection.

Du et al. [18] use a cost graph to estimated the impact of mis-speculation, and the loops are statically selected based on the estimated mis-speculation cost along with other simple selection criteria such as the size of iterations and the number of iterations. Those simple selection criteria are also commonly used by other studies [69, 70].



(a) Source code.

(b) Loop graph.

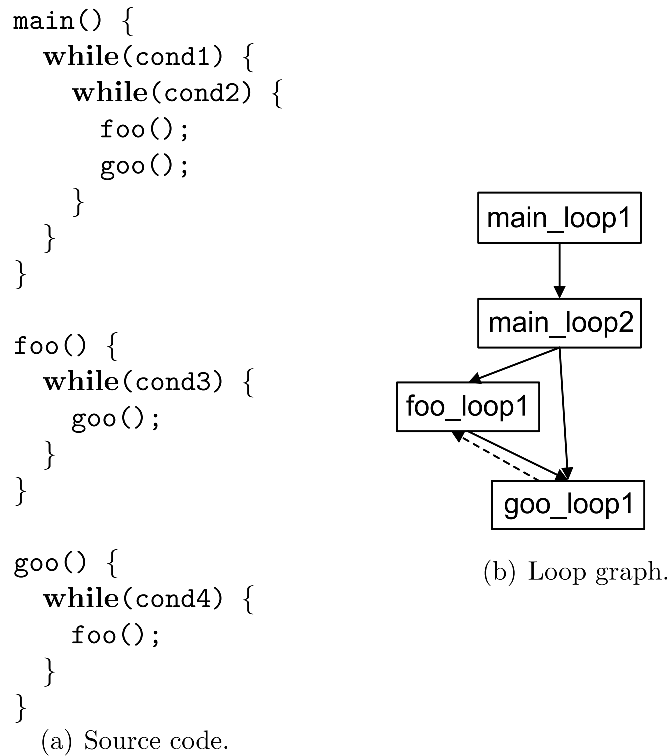
Figure 3.1: Example of loop graph.

3.2 Loop Selection Algorithm

In this section, we present a loop selection algorithm that chooses a set of loops to parallelize while maximizing overall program performance. The algorithm takes as input the speedup and coverage of all the loops in a program and outputs an optimal set of loops for parallelization.

3.2.1 Loop Graph

The main constraint in loop selection is that there should be no nesting relation between any two selected loops. To capture the nesting relation between loops, we



(a) Source code.

(b) Loop graph.

Figure 3.2: Example of loop graph with recursion.

construct a Directed Acyclic Graph (DAG) called a *loop graph*. Each node in the loop graph represents a static loop in the original program, and a directed edge represents the nesting relation between two loops.

Figure 3.1(a) shows an example source code and its corresponding loop graph is shown in Figure 3.1(b). Two loops in *main* procedure are represented by nodes *main_loop1* (for outer loop) and *main_loop2* (for inner loop) respectively, while the loops in procedure *foo* and *goo* are represented by nodes *foo_loop1* and *goo_loop1*. The edge from *main_loop1* to *main_loop2* indicates the syntactical nesting, while the edge from *main_loop2* to *foo_loop1* indicates nesting through procedure call

foo().

A recursive call introduces a cycle in the loop graph that violates the acyclic property. Figure 3.2 shows a similar example as the previous one except that procedure *goo* also calls *foo*. The dash edge from *goo_loop1* to *foo_loop1* represents this nesting relation and causes cycle in the loop graph. A cycle can usually be broken if we can identify back edge [1]. An edge from node s to node t is a back edge if every path that reaches s from the root passes through t. However, in Figure 3.2(b), no back edge is detected even though there is a cycle. In such case, we favor the nesting relation from *foo_loop1* to *goo_loop1* since *foo_loop1* is invoked before *goo_loop1* in terms of the time when they are first invoked. Consequently the edge from *goo_loop1* to *goo_loop1* is deleted.

A loop graph, like a call graph [46], can be constructed through runtime profiling or compiler static analysis. In our current implementation, it is built upon efficiently collected runtime profiles.

3.2.2 Selection Criterion

Since speculatively parallelizing nested loops usually requires complicated hardware support [55], our target thread execution model allows at most one active loop at any time during execution. As a result, we do not simultaneously select any two loops that have nesting relation. In the case of nested loops, we have to rely on some criterion to decide which loops we want to parallelize. Both speedup

and coverage are commonly used ones. However, only considering either one of them may not achieve desirable performance. For example, a loop with high coverage may not have good performance, and a loop with good performance may not have high coverage.

Therefore, we use a criterion called *benefit* that considers both speedup and coverage of a loop. It is defined as follows:

$$benefit = coverage \times (1 - \frac{1}{speedup}) \tag{3.1}$$

The benefit value represents the percentage of overall program execution time that can be reduced by parallelizing that loop. Thus a loop with a larger benefit value is more likely to be selected. The benfit values of two loops are additive if they are not nested. For example, *loop 1* has a benefit value of 0.15, and *loop 2* has a benefit value of 0.2. If they are not nested, their execution will not be overlapped. So overall we can save 0.35 percent of total execution time by parallelizing both of them. The speedup for the whole program can be computed directly from the benefit value as follows:

$$program\ speedup = \frac{1}{1 - benefit} \tag{3.2}$$

3.2.3 Loop Selection Problem

The general loop selection problem can be stated as follows: given a loop graph with benefit value attached to each node, find a set of nodes that maximizes the

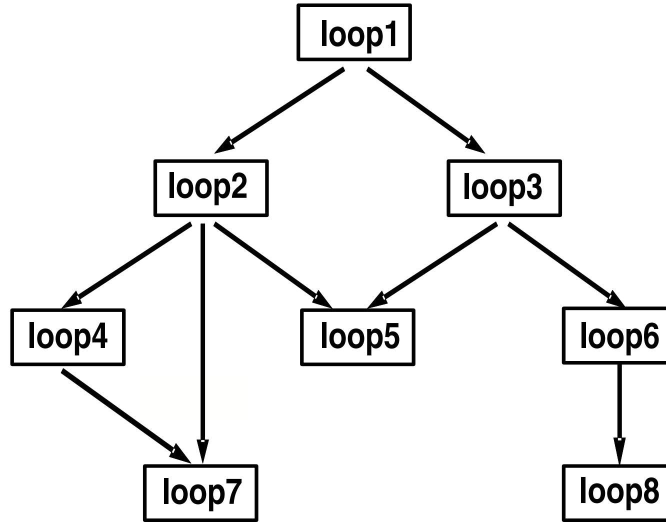


Figure 3.3: Loop graph before pruning.

overall benefits such that there is no path between any two selected nodes. We transform this loop selection problem into a well-known NP-complete problem, *weighted maximum independent set problem* [17], by computing the transitive closure of the loop graph. A set of nodes is called an *independent set* if there is no edge between any two of them.

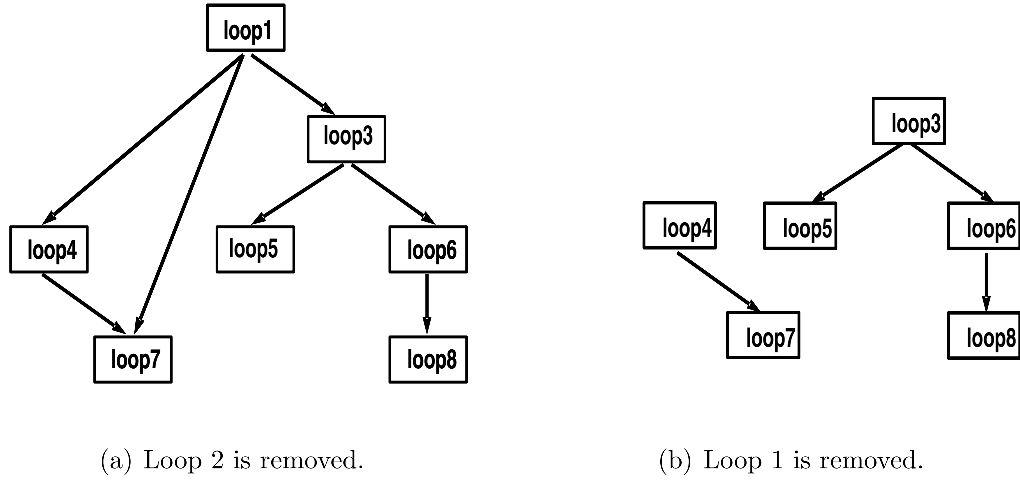
3.2.4 Graph Pruning

The general loop selection problem is NP-complete, so that an exhaustive search algorithm only works for a graph with few nodes. For a graph with hundreds or thousands of nodes, which is common for most of the benchmarks that we are studying, a more efficient heuristic has to be used. Because a heuristic-based algorithm only gives a sub-optimal solution, we must use it wisely.

By applying a technique called *graph pruning*, we can find a reasonable approximation more efficiently. Graph pruning simplifies the loop graph by eliminating those loops that will not be selected as speculative threads. These would include such loops as: (i) loops that have less than 100 dynamic instructions on average, as they are more appropriate for ILP; (ii) loops that have no more than 2 iterations on average, as they are more likely to underutilize multiple processor resources; and (iii) other loops that are predicted to slow down the program execution if parallelized. While the first two types of loops can be easily identifid by using profiles, the third one requires accurate performance estimation, which will be discussed in Section 3.3.

In the loop graph shown in Figure 3.3, assume loop 2 has poor parallel performance, we can remove this loop from the graph since it will not be considered for selection. After it is removed, we have to maintain the nesting relation among the remaining loops. The edges from *loop 1* to *loop 4*, *loop 5*, and *loop 7* are added, and the resulting loop graph is shown in Figure 3.4(a). Moreover, if *loop 1* is also removed due to its poor performance, the single connected graph is split into two sub-graphs, and each of them has smaller size than the original one, as shown in Figure 3.4(b).

Graph pruning reduces the size of a loop graph by eliminating unsuitable loops. After we delete those unnecessary nodes, one single connected graph is split into multiple smaller disjointed sub-graphs. Then we can apply selection algorithm to



(a) Loop 2 is removed.

(b) Loop 1 is removed.

Figure 3.4: Loop graph after pruning.

each sub-graph independently. It is efficient to use exhaustive searching algorithm for small sub-graphs. For larger sub-graphs, heuristic-based searching algorithm usually gives a reasonable approximation.

3.2.5 Exhaustive Searching Algorithm

In this simple algorithm, we exhaustively try every set of independent loops to find the one that provides the maximum benefit. For each computed independent loop set, we track all loops that have nesting relations to any loop in this independent set and record them in a bit vector called a *conflict vector*, where each bit represents a static loop. A bit is set if the corresponding loop has nesting relation to any loop that has been selected. By using a conflict vector, it is easy to find a new independent loop to add into the current set. After a new loop is added, the

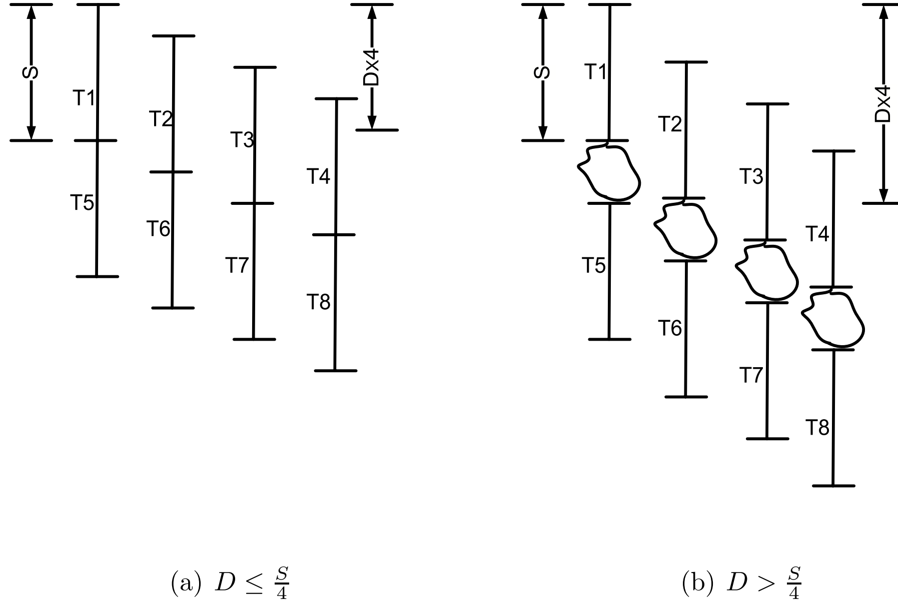
conflict vector is updated as well.

An exhaustive searching algorithm gives an accurate solution for the loop selection problem, but is very inefficient. Graph pruning creates smaller sub-graphs that are suitable for exhaustive searching. Exhaustive searching works efficiently for sub-graphs with less than 50 nodes in our experiments.

3.2.6 Heuristic-based Searching Algorithm

Even after graph pruning, some sub-graphs are still very big. For those, we use a heuristic-based algorithm. Here is how a simple heuristic works. We first sort all the nodes in a sub-graph according to their benefit values. Then we pick one node at a time and add it into the independent set such that this node has the maximal benefit value among the unselected nodes and it does not conflict with already selected nodes. Similarly to the exhaustive searching algorithm, we maintain a conflict vector for the selected independent set and update it whenever a new node is added.

Although this simple greedy algorithm gives a sub-optimal solution, it can select a set of independent loops from a large graph in polynomial time. In our experiments, the sizes of most sub-graphs are less than 200 nodes after graph pruning, so the inaccuracy introduced by this algorithm is negligible.



(a) $D \leq \frac{S}{4}$ (b) $D > \frac{S}{4}$

Figure 3.5: Impace of delay D assuming 4 processors.

3.3 Loop Speedup Estimation

Our goal in loop selection is to maximize the overall program performance, which is represented as the benefit value of the selected loops. In order to calculate the benefit value for each loop, we have to estimate both the coverage and speedup of each loop. Coverage can be estimated using a runtime profile. To estimate speedup, we have to estimate both sequential and parallel execution time.

We assume that each processor executes one instruction per cycle, i.e., each instruction takes one cycle to finish. It is relatively easy to estimate sequential execution time T_{seq} of a loop. We can determine the average size of a thread (average number of instructions executed per iteration) and the average number of parallel threads (average number of times a loop iterates) by using a profile. T_{seq}

can be approximated by using equation (3.3), where S is the average thread size and N is the average number of threads.

$$T_{seq} = S \times N \tag{3.3}$$

On the other hand, the parallel execution time depends on other factors such as the thread creation overhead, the cost of inter-thread value communication, and the cost of mis-speculation. We simplify the calculation by dividing the total parallel execution time T_{par} into two parts: perfect execution time $T_{perfect}$ and mis-speculation time $T_{misspec}$. $T_{perfect}$ is the parallel execution time on p processors assuming that there is no mis-speculation, while $T_{misspec}$ is the wasted execution time due to mis-speculation.

$$T_{par} = T_{perfect} + T_{misspec} \tag{3.4}$$

In the perfect execution model, the main overheads for the parallel execution are caused by the synchronization of frequently occurring dependences and the thread creation overhead. We model these overheads as the delay D between two consecutive threads. It is estimated by equation (3.5).

$$D = max(T_{syn}, O_{create}) \tag{3.5}$$

where T_{syn} is the synchronization cost and O_{create} is the thread creation overhead.

Depending on the delay D, we use different equations to estimate $T_{perfect}$. If $D \leq \frac{S}{p}$, we can have a perfect pipelined execution of threads, as shown in Figure

3.5(a), and use equation (3.6) for estimation.

$$T_{perfect} = (\frac{N-1}{p} + 1) \times S + ((N-1) \mod p) \times D \tag{3.6}$$

If $D > \frac{S}{p}$, delay D causes bubbles in the pipelined execution of threads and has a higher impact on the overall execution time, as shown in Figure 3.5(b). In this case, we use equation (3.7) for estimation.

$$T_{perfect} = (N-1) \times D + S \tag{3.7}$$

The key to accurately predicting speedup is how to estimate T_{syn} and $T_{misspec}$. T_{syn} is caused by the synchronization of frequently occurring data dependences, while $T_{misspec}$ is caused by the mis-speculation of unlikely occurring data dependences. We describe techniques to estimate T_{syn} and $T_{misspec}$ in the following sections.

3.3.1 $T_{misspec}$ Estimation

When a mis-speculation is detected, the violating thread will be squashed and all the work done by this thread becomes useless. We use the amount of work thrown away in a mis-speculation to quantify the impact of the mis-speculation on the overall parallel execution. The amount of work wasted depends on when a mis-speculation is detected. For instance, if a speculative thread starts at cycle c_{start} and mis-speculation is detected at cycle c_{detect}, we have $(c_{detect} - c_{start})$ wasted cycles. This is illustrated in Figure 3.3.1, where a mis-speculation caused by $load1$ in thread T_2 is detected by $store1$ in thread T_1.

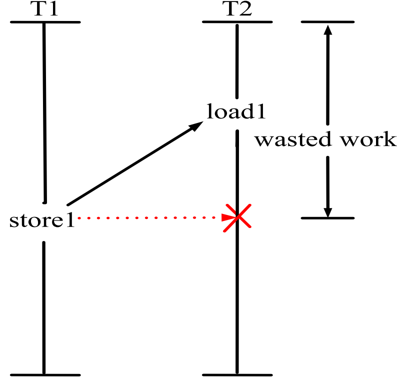
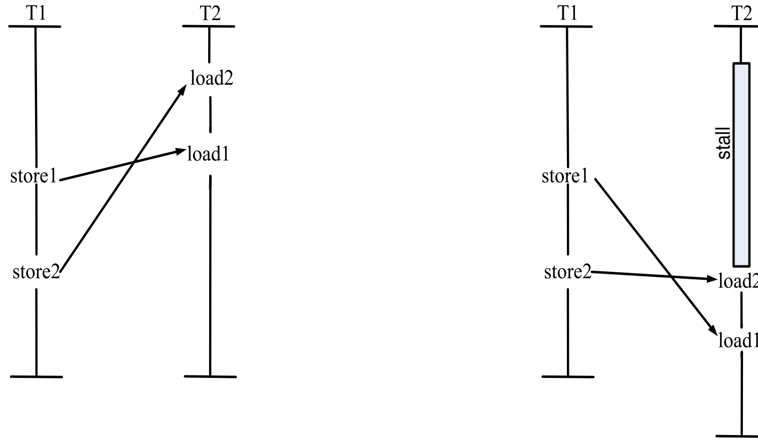


Figure 3.6: Cost of mis-speculation.

In our thread execution model, a mis-speculation is detected only when the consumer load is executed before the producer store. Assume that we speculate on a data dependence from instruction i_1 to i_2. Also assume c_1 and c_2 are the cycles at which i_1 and i_2 are executed within a thread. A mis-speculation only occurs when $D \times d + c2 < c1$, where D is the delay between consecutive threads defined in equation (3.5), and d is the dependence distance. The amount of wasted work due to a mis-speculation is $c_1 - D \times d$. Finally, the dependence probability $prob(i_1, i_2)$ between i_1 and i_2 is considered, and the impact of this mis-speculation is estimated by:

$$T_{misspec} = (c_1 - D \times d + O_{squash}) \times prob(i_1, i_2) \tag{3.8}$$

where O_{squash} is the overhead of thread squashing. If there are more than one data dependence that we decide to speculate, the one with the maximal $T_{misspec}$ is selected to estimate the overall impact of mis-speculation. Also, how to estimate



(a) Two inter-thread data dependences.

(b) The stall caused by synchronizing load2 and store2.

Figure 3.7: Cost of synchronization.

the cycle at which a particular instruction is executed within a thread will be discussed in Section 3.3.3.

3.3.2 T_{syn} Estimation I

One way to estimate the amount of time that parallel threads spend on synchronization is to identify all the instructions that are either the producers or the consumers of frequent inter-thread data dependences, and estimate the cost of synchronization as the total cost of executing all such instructions. In Figure 3.7(a), there are two data dependences between thread $T1$ and $T2$ and four load/store instructions are involved, so the overall synchronization cost is estimated as $(4 + O_{comm})$ cycles, where O_{comm} is the communication cost between threads.

The intuition behind this simple estimation is that the more instructions are involved in the inter-thread data dependences, the worse the parallel performance is. One drawback of this approach is that we treat all dependences equally during estimation. In fact, some data dependences could have higher impact on the performance than others. For the example in Figure 3.7(a), the dependence between $store2$ and $load2$ results in a higher synchronization cost than the dependence between $store1$ and $load1$ since it causes the longer stall in the consumer thread $T2$. In order to quantify the impact of each dependence, we have to consider the timing of executing each producer and consumer instruction. This leads to our next estimation technique described in Section 3.3.3.

3.3.3 T_{syn} Estimation II

To take into consideration the impact of synchronizing each dependence, we propose estimation technique II. Assuming that $load1$, the consumer instruction in thread T_2, is executed at cycle c_2 and that $store1$, the producer instruction in thread T_1, is executed at cycle c_1, the cost of synchronization between these two instructions is estimated as $(c_1 - c_2)$.

If the data dependence does not occur between two consecutive threads but rather has a dependence distance of d, the impact on the execution time of a particular thread should be averaged out over the dependence distance. Thus the impact of synchronizing the dependence between two threads is estimated as

follows:

$$T_{syn} = \frac{c_1 - c_2 + O_{comm}}{d} \tag{3.9}$$

There is one more missing piece if this estimation technique is to be successful, which is how to determine which cycle of a particular instruction should be executed. Since it is not possible to perfectly predict the dynamic execution of a thread, we made a simplification assuming each instruction will take one cycle to execute; thus the start cycle is simply an instruction count of the total number of instructions between the beginning of the thread and the instruction in question. However, due to complex control flows that are inherent to general-purpose applications, there can be multiple execution paths, each with different path length, that reach the same instruction. Thus the start time of a particular instruction is the average path length weighted by path taken probability, as shown in equation (3.10).

$$c = \sum_{p_i \in \{all\ paths\}} length(p_i) \times prob(p_i) \tag{3.10}$$

After we compute the cost for each dependence, the overall cost of synchronization is determined by the most costly one, since the cost of others can be hidden. As shown in Figure 3.7(b), $load1$ does not have to wait since the value has already been generated by $store1$.

3.3.4 T_{syn} Estimation III

Previous work has shown that the compiler can effectively reduce the cost of synchronization through instruction scheduling and that such optimizations are particularly useful for improving the efficiency of communicating register-resident scalars [69]. Unfortunately, the estimation technique described in Section 3.3.3 does not take such optimization into consideration and tends to over-estimate the cost of synchronization. It is desirable to find an estimation technique that considers the impact of instruction scheduling on reducing the synchronization cost. Thus, we use a third technique, in which the start time of an instruction is computed from the data dependence graph. When there are multiple paths that can reach an instruction in the data dependence graph, the average start time of this instruction can be measured by equation (3.11), assuming that the average length of a path p_i that reaches this instruction in the data dependence graph is $length(p_i)$.

$$c = max(length(p_i)) \tag{3.11}$$

Chapter 4

Non-Loop Partitioning

For general-purpose applications, extracting loop threads may not achieve adequate amount of parallelism due to complex control flow and hard-to-optimize data dependences. In such cases, it is desirable to search for TLP beyond loop threads. We call the process of extracting non-loop threads from a sequential program *non-loop partitioning*. In this chapter, we will describe an efficient non-loop partitioning algorithm. By using both loop threads and non-loop threads, we exploit as much parallelism as possible from those hard-to-paralelize applications.

Generally speaking, non-loop threads refer to any threads that do not correspond to loop iterations. Although any set of connected basic blocks in CFG can be treated as a non-loop thread, a non-loop thread typically has a unique entry block that can reach all other blocks in this set. A non-loop thread could have multiple exit blocks. However, the more exits it has, the more difficult to predict

its successor thread. To exploit TLP in an even large scope, a non-loop thread is allowed to contain nested loops or procedure calls.

4.1 Related Work

Realizing that loop-level parallelism is limited in general-purpose applications, researchers have been searching for parallelism in non-loop threads over the years.

The Multiscalar compiler [64] extracts non-loop threads by walking through the CFG of a procedure and forming threads from consecutive basic blocks. Since partitioning a program into multiple threads to achieve optimal parallel performance in general is NP-complete [56], a variety of heuristics that consider key performance factors are used to approximate the optimal solution. Since a thread typically terminates at the loop boundary or procedure call, the average size of threads extracted by this compiler is only around 20 instructions. The parallelism within such small scope can usually be exploited by superscalar processors.

Johnson et al. [33] later improve this partitioning algorithm by modeling program partitioning as a min-cut problem. They consider several key performance factors simultaneously during partitioning, so that better performance trade-off can be achieved. However, due to the complexity of their graph-based partitioning algorithm, it is inefficient to handle large CFG with thousands of basic blocks.

The Mitosis compiler [52] uses a spawning pair to describe the creation point and execution point of a speculative thread. So the goal of their algorithm is

to find the best set of spawning pairs to maximize the performance. The main feature of their thread execution model is that they use a pre-compuatation slice to compute live-in values for a speculative thread. Since they are primarily interested in understanding the performance limits, complicated performance analysis and exhaustive search are used in their partitioning algorithm.

Since program partitioning in general is a complicated process, some studies focus on more specific types of non-loop threads. Oplinger et al. [47] study the potential of exploiting procedure-level parallelism. By executing a procedure with the continuation code in its caller procedure, their study shows significant performance improvement. Since procedures are well-defined program structures, extracting non-loop threads targeting procedure-level parallelism is relatively straightforward. So that such non-loop threads is also exploited by POSH compiler [42].

4.2 Algorithm Features

Compared to the previous partitioning algorithms, our algorithm has the following features:

First, when partitioning sequential code, we rely on the loop structure in a program and partition each individual loop one at a time in an inner-loop-first order. As a result, before an outer loop is partitioned, all its inner loops have already been considered for partitioning. If an inner loop has been partitioned into non-loop threads, it will be excluded when the outer loop is partitioned.

```
Nonloop_Partition(Procedure P) {
    process each loop in inner-loop-first order {
        if (loop is not parallelized) {
            contruct compact CFG;
            find frequent execution path;
            initialize path set S;
            while (!Empty(S)) {
                P = Pop(S);
                if (BiPartition(P, P1, P2)) {
                    Push(S, P1);
                    Push(S, P2);
                }
            }
            extend partitioned sub-paths;
        }
    }
}
```

Figure 4.1: Non-loop partitioning algorithm applied to each procedure.

Otherwise, it will be considered as part of a non-loop thread in the outer loop. Similarly, if an inner loop has been partitioned into loop threads, it will be excluded as well. In the end, the procedure itself is partitioned, and it is handled in the same way as we partition a loop. Leveraging loop structure and partitioning each loop separately greatly improve efficiency of the partitioning algorithm, especially when a procedure has complex nested loops.

Second, our partitioning algorithm is based on frequent execution path. Instead

```
BiPartition(P, P1, P2) {
    for each BB b on the path P {
        weight = Performance_Estimation(P, b);
        if (weight > max_weight) {
            max_weight = weight;
            cut_point = b;
        }
    }
    if (max_weight > PERFORMANCE_THRESHOLD)
        P1 = all BBs from the head of P to the cut_point;
        P2 = all remainign BBs on P;
        return TRUE;
    }
    else
        return FALSE;
}
```

Figure 4.2: Bipartition algorithm.

of partitioning a CFG directly, we first identify frequent execution path in the CFG. This path is then partitioned into multiple sub-paths. The partitioned sub-paths are finally expanded to include other basic blocks in the CFG to form non-loop threads. This path-based partitioning algorithm is much simpler to implement than a graph-based algorithm while still achieving comparable performance along the frequent execution path.

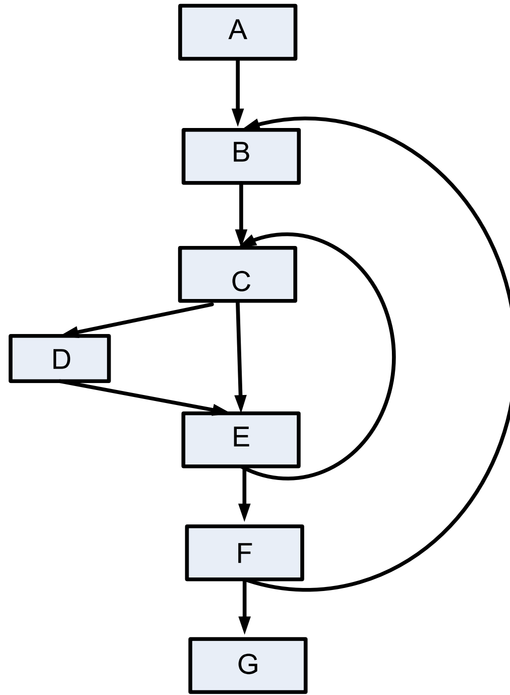
4.3 Partitioning Algorithm

Figure 4.1 illustrates our algorithm that partitions a procedure into non-loop threads. It iterates through all loops in that procedure from the inner most one to the outer most one. Each time an un-parallelized loop is processed, its body will be partitioned into multiple disjointed non-loop threads only if executing them in parallel can improve performance. Note that the procedure is treated as a special outer most loop (i.e., a loop that iterates only once when invoked) and is handled uniformly with other loops. The following discussion on loops is applicable to procedures as well.

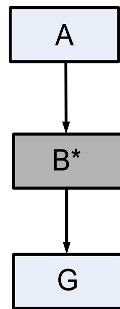
4.3.1 Compact CFG

For each loop under consideration, a compact CFG is first created where all inner loops are represented as super nodes. If any inner loop has been parallelized by loop threads or non-loop threads, it is excluded from current partitioning, otherwise, it is included in a thread created by current partitioning. All back edges of the current loop under partitioning are ignored since they will not be included by any non-loop threads extracted from the loop body. As a result, a compact CFG is a DAG, and this simplifies the partitioning process.

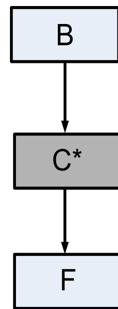
Figure 4.3 shows a CFG and how it is compacted. The original CFG of a procedure is shown in Figure 4.3(a). It contains a double-nested loop. The compact CFG for this procedure is shown in Figure 4.3(b), where the nested loop is



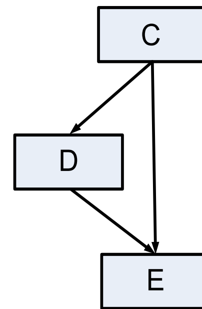
(a) The CFG of a procedure.



(b) The compact CFG of the procedure.



(c) The compact CFG of the outer loop.



(d) The compact CFG of the inner loop.

Figure 4.3: Examples of compact CFG.

represented as super node B*. The compact CFG for the outer loop is shown in Figure 4.3(c), where the inner loop is represented as super node C*. The compact CFG for the inner loop is shown in Figure 4.3(d). For both the inner loop and the outer loop, the back edges are not represented in their compact CFGs.

4.3.2 Frequent Execution Path

Before we partition a loop into non-loop threads, we identify the frequent execution path in its compact CFG. Due to complex control flow, there are usually multiple paths from the entry to the exit in compact CFG. The frequent execution path is the one that a loop spends most of its execution time. A path p_i is identified as a frequent execution path if it has the maximal value of $prob(p_i) \times length(p_i)$, where $prob(p_i)$ is path probability, and $length(p_i)$ is the path length. In our implementation, path probability $prob(p_i)$ is estimated by using edge profiles, and path length $length(p_i)$ is approximated by the number of instructions along path p_i.

A frequent execution path is computed by topologically traversing the compact CFG. For each node n_i, the frequent execution path, $freq_path(n_i)$ is the most time consuming path that reaches this node from the entry. It is calculated by examining the frequent path of each predecessor n_p. The one with maximal value of $prob(freq_path(n_p)) \times length(freq_path(n_p))$ is selected and extended to include current node n_i. The probability of $freq_path(n_i)$ is computed by:

$$prob(freq_path(n_i)) = prob(freq_path(n_p)) \times prob(e_{p,i})$$

where $prob(e_{p,i})$ is the probability of edge $e_{p,i}$ from n_p to n_i. And the length of $freq_path(n_i)$ is computed by:

$$length(freq_path(n_i)) = length(freq_path(n_p)) + length(n_i)$$

where length(n_i) is the estimated number of instructions in node n_i.

The frequent execution path for a compact CFG is the one computed for its exit node. If there are multiple exit nodes, the maximal one is selected.

The identified frequent execution path is then used to initialize the *path set*, a set that contains all paths that need to be further partitioned. Note that, there could be some inner loops on this path that have already been parallelized. In such cases, their corresponding supper nodes are excluded from the frequent execution path. Consequently, the path is divided into multiple sub-paths, and those sub-paths are added into the path set for further partitioning.

4.3.3 Recursive Partitioning

After the path set is initialized, the main partitioning process starts. When a path is chosen for partitioning, it will be bi-partitioned into two sub-paths if the overall performance can be further improved based on the performance estimation, which will be discussed in Section 4.3.4. These two sub-paths are then added into the path set for further partitioning. If partitioning a path does not improve performance, this path will not be considered by future partitioning and is simply discarded from the path set. This recursive partitioning process continues until the path set

is empty.

4.3.4 Performance Estimation

When partitioning a particular path p into two sub-paths, the most difficult task is to find the right partitioning point to maximize performance. In our algorithm, we try every basic block on path p as the potential partitioning point and estimate the performance of resulting sub-paths. The basic block with the best estimated performance is selected as the final partitioning point for path p.

During performance estimation of sub-paths, the impacts of several key performance factors such as data dependence, control predictability, and load imbalance have to be taken into consideration. To simplify discussion, other machine-dependent parameters such as thread creation overhead, squash overhead, and communication delay are ignored. It is straightforward to include them in the final performance estimation model.

Control Predictability

Assume path p is partitioned into p_1 and p_2. We first consider the impact of control predictability. Assume the probability of executing p_2 given p_1 is executed is P, the overall parallel execution time T_{par} is approximated by:

$$T_{par} = T_{succ} \times P + T_{fail} \times (1 - P)$$

where T_{succ} is the execution time when p_2 is correctly predicted, and T_{fail} is the execution time when p_2 is mis_predicted.

If p_2 is mis-predicted, another path should be taken during execution. However, it is difficult to know which path is really taken during estimation, we assume that the same path p_2 is re-executed. Since the mis-prediction is usually detected at the end of p_1, T_{fail} is approximated by:

$$T_{fail} = length(p_1) + length(p_2)$$

If p_2 is correctly predicted, both data dependence and load imbalance are the primary factors determining the parallel performance. Their impacts are considered when we estimate T_{succ}.

Load Imbalance

When estimating T_{succ}, we first consider the impact of load imbalance. Let T_1 be the time taken for executing all instructions in p_1, and T_2 be the time taken for executing all instructions in p_2. When load imbalance is considered, the parallel execution time is determined by the longer one:

$$T_{succ} = max(T_1, T_2)$$

For simplicity, we assume the execution of p_1 is not affected by its predecessor paths, so that T_1 is approximated by:

$$T_1 = length(p_1)$$

although some instructions on p_1 may depend on instructions on the predecessor paths and cause stalls when p_1 is executed.

Now, T_2 is the last parameter we need to estimate. The impact of data dependences between p_1 and p_2 is considered when we estimate T_2.

Data Dependence

As we discussed in in Section 3, there are two ways to handle an inter-thread data dependence. One is synchronization, which is used for frequently occurring dependences to avoid excessive mis-speculations. The other one is speculation, which is used for infrequently occurring dependences to exploit speculative parallelism. The impact of both synchronization and speculation is modeled as delay between two threads.

Assume there is a data dependence between instruction i_1 on path p_1 and instruction i_2 on path p_2, and c_1 and c_2 are the cycles at which i_1 and i_1 are executed in p_1 and p_2. If it is a frequent dependence, synchronization is used, and the delay caused by this synchronization is estimated by:

$$d = c_1 - c_2$$

If it is a infrequent dependence, speculation is used, and the delay caused by a mis-speculation is estimated by:

$$d = c_1 \times prob(i1, i2)$$

where $prob(i1, i2)$ is the dependence probability between i_1 and i_2, and it equals to the probability that a mis-speculation occurs.

We examine all dependences between p_1 and p_2, and select the one with maximal delay d to approximate the overall impact of data dependence. The execution time of p_2 is estimated by:

$$T_2 = length(p_2) + d$$

4.3.5 Sub-Path Expansion

The partitioned sub-paths are expanded to cover all remaining basic blocks in the compact CFG. Finally dis-jointed sub-graphs are formed, and each of them corresponds to a non-loop thread. A basic block can be included in a sub-path if it can be reached from that sub-path. However, a basic block may be reached by multiple sub-paths. In such case, the probability of reachability needs to be considered, and the basic block is included in the sub-path from which it is most likely to be reached.

Chapter 5

Thread Optimizations

In this chapter, three thread optimization techniques are presented to improve the performance of parallel threads extracted using the techniques decribed in Chapter 3 and Chapter 4.

For general-purpose applications, one of the main constraints that limit the parallel performance is inter-thread data dependence. While infrequently occurring dependences can be optimistically handled by TLS, frequently occurring dependences often serialize the parallel execution and become the performance bottleneck. The first two techniques, *instruction scheduling* and *reduction transformation* are proposed to improve the inter-thread communication due to frequently occurring data dependences.

Load imbalance is another performance bottleneck in the parallel execution. The complex control flow in general-purpose applications often causes dramatic

variations in the thread size. Our third technique, *iteration merging*, is proposed to achieve more balance workloads across loop threads.

5.1 Related Work

Instruction scheduling is an effective technique to reduce the impact of inter-thread data dependence. It has been exploited in both the traditional parallelizing compilers [15, 9] and TLS compilers [65]. It also has been extensively used in software pipelining to minimize the the delay of executing consecutive loop iterations [40, 3]. All these scheduling techniques are conservative in the sense that they have to follow all possible intra-thread dependences during scheduling. In contrast, our scheduling technique, based on the algorithm developed by Zhai et al. [69], performs more aggressive scheduling by using both intra-thread data and control speculation. We extend their algorithm to do aggressive scheduling for memory dependence. Furthermore, recovery code is used to reduce the mis-speculation cost.

The Mitosis compiler [52] uses a pre-computation slice to speculatively compute live-in values for a thread, so it can be regarded as a complex value prediction. There are two major differences between that approach and our instruction scheduling: i) extra cost is introduced due to the execution of the pre-computation slice, while instruction scheduling does not introduce so many additional instructions (control speculation may introduce more dynamically executed instructions, but such overhead is often negligible); ii) a mis-prediction caused by the pre-

computation slice can be detected only when the producer thread finishes, while in our scheme, a mis-prediction is detected much earlier by the producer thread through the intra-thread speculation mechanism.

In traditional compiler [35], the dependence caused by a reduction variable is removed by a transformation called *reduction elimination*. However, the application of reduction elimination requires that there is no use of intermediate results of reduction variable in the parallelized loop. Our proposed reduction transformation is not limited by this constraint, and it also takes advantage of the usage pattern of the reduction variable to perform aggressive transformation.

Prabhu et al. [50, 51] propose a complex manual transformation technique called *speculative pipelining* to transform the loop to achieve balanced workloads. However, our iteration merging technique is proposed in the context of automatic compilation and thus can be integrated into an optimizing compiler.

5.2 Speculative Scheduling for Memory Dependence

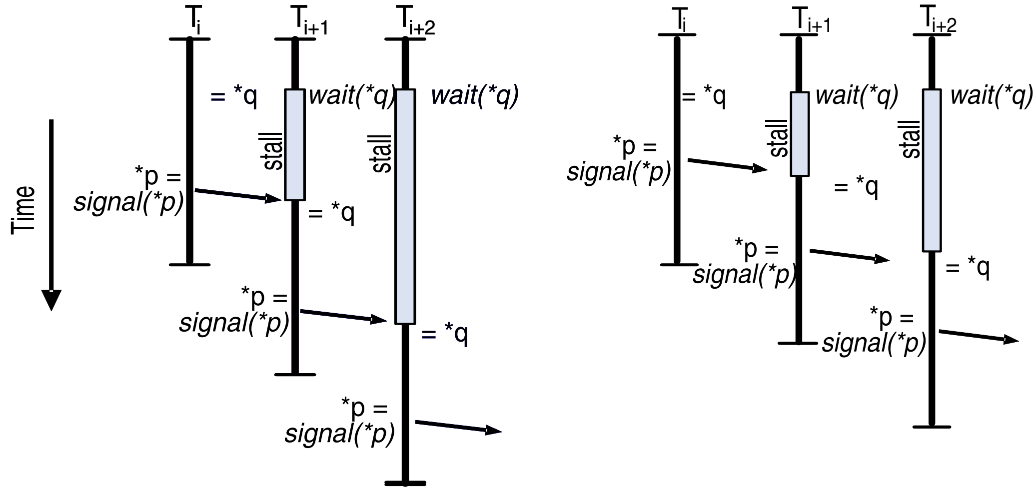
In order to avoid excessive failures under TLS, synchronizations are required for frequently occurring *memory* dependences [70]. In Figure 5.1(a), there is a frequent data dependence between *store *p* and *load *q* in consecutive threads. To avoid mis-speculation, synchronization is used to delay *load *q* until *store *p* finishes

its execution. Here the synchronization is implemented by a pair of *signal* and *wait* instructions. *signal* is used by the producer thread to forward the data, while *wait* is used by the consumer thread to wait for the expected data. A pair of communicating *signal* and *wait* are assigned the same synchronization id (*sid*). *sid* is forwarded by *signal* along with data, while *wait* only receives data with the matching *sid*.

Although synchronization avoids frequent failures, it may cause stalls in the consumer thread if the expected data has not been produced. In Figure 5.1(a), such stalls causes severe serialization in parallel execution. The execution path between *store *p* and *load *q* is referred as *critical forwarding path* [69]. The impact of synchronization can be measured by the length of critical forwarding path. The longer the path is, the more impact the synchronization has.

Instruction scheduling can potentially reduce the stall time caused by synchronization. As shown in Figure 5.1(b), *store *p* is scheduled to be executed earlier so that *load *q* can access the produced data earlier. Therefore, the stall time of the consumer thread is reduced, and increased overlap between threads leads to a better parallel performance.

While instruction scheduling for *register* dependence has been shown effective for many applications [69], the benefit of scheduling for *memory* dependence is still unknown. Due to the facts that memory dependences are prevalent in general-purpose applications, it is important to investigate the performance impact of



(a) Before scheduling.

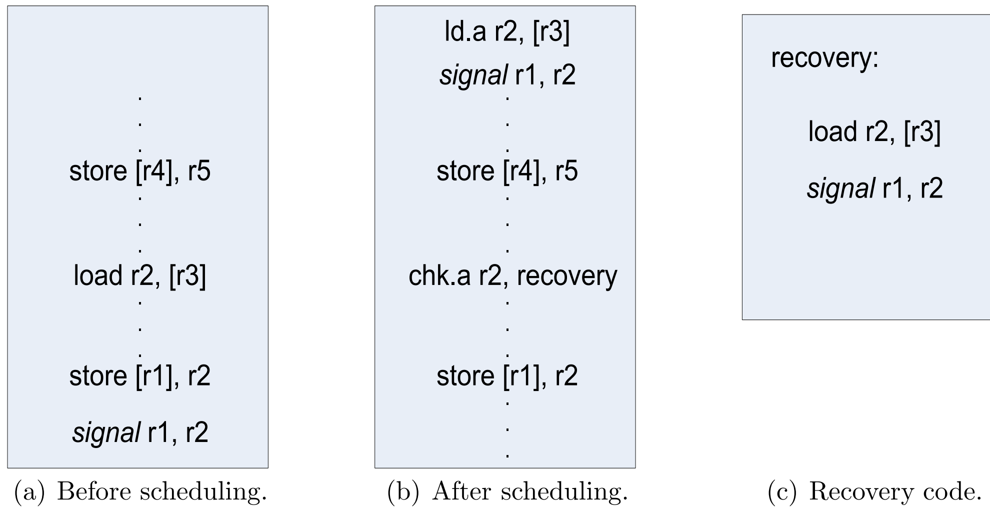
(b) After scheduling.

Figure 5.1: Impact of instruction scheduling.

scheduling techniques for memory dependence.

5.2.1 Intra-Thread Speculation

Our thread execution model supports a synchronization mechanism similar to the one described by Zhai et al. [70]. When a frequently occurring memory dependence is synchronized, both the *address* and the *value* of the producer store instruction are forwarded by the signal instruction. Therefore, the purpose of instruction scheduling is to hoist instructions such that both the store address and the store value can be computed and forwarded as early as possible. Note that, the store instruction itself does not need to be scheduled (the scheduling of **p* shown in Figure 5.1(b) is only for illustration purpose). This greatly simplify the scheduling



(a) Before scheduling. (b) After scheduling. (c) Recovery code.

Figure 5.2: Data speculation used in scheduling for memory dependence.

process since the possible *write-after-write* and *write-after-read* hazards due to the scheduling of a store instruction are avoided.

The main constraints during instruction scheduling are the data and control dependences. Data and control speculation are commonly used methods to overcome the dependence limitation during aggressive scheduling [41, 16]. In our thread execution model, both *intra-thread* data and control speculation similar to those on IA-64 architecture [32] are supported. However, in our scheme, *intra-thread* speculation is mainly used for improving the efficiency of *inter-thread* data communication.

Data Speculation

Data speculation allows a load instruction to be aggressively scheduled across a possible aliasing store instruction. Not only can it be used to improve the

single thread performance [16], it can also be used to improve the inter-thread communication.

In Figure 5.2(a), there is a store instruction *store [r1], r2* that causes frequent inter-thread dependence. The store address is kept in register *r1*, and the value to be stored is kept in register *r2*. Synchronization is used and a signal instruction *signal r1, r2* is inserted after this store. This signal instruction will forward both the address and the value of store to the consumer thread. In order to forward the data earlier, instruction scheduling is used to hoist this signal instruction. Since the value of *r2* used by *signal r1, r2* is loaded by *load r2, [r3]*, this load is also hoisted along with the signal. Now assume there is a store instruction *store [r4], r5* that *load r2, [r3]* may depend on along the scheduling path. If the probability of this dependence is low, it is beneficial to schedule the load across the store so that the signal can be executed much earlier, as shown in Figure 5.2(b). Different from an ordinary load, this load may cause potential dependence violation if it accesses the same memory location as the store. Consequently, it is changed into a data speculative load *ld.a*. The potential dependence violation is checked by a *chk.a* instruction inserted to the home location of the speculative load. If a violation is detected, a piece of recovery code shown in Figure 5.2(c) is invoked. In that recovery code, the value of $r2$ is reloaded and re-forwarded to the consumer thread.

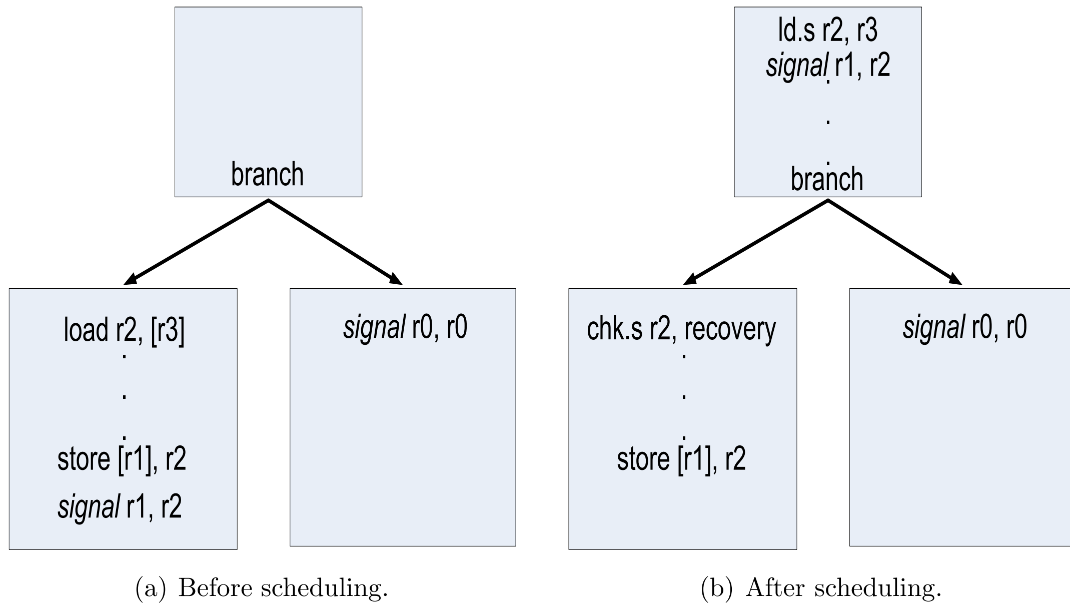
If the speculative load succeeds, the consumer thread receives the corresponding

signal only *once* during its execution. However, if the speculative load fails, the consumer thread will receive the signal *twice*. In our thread execution model, the consumer thread, upon receiving the same signal again (marked by the same *sid*), knows that a mis-speculation is detected by the producer thread and the previously received data (either address or value) is wrong. If the data has already been accessed, the consumer thread and all the following threads need to be squashed and re-executed.

With the recovery code support, the producer thread does not need to be squashed when an intra-thread mis-speculation is detected. The consumer thread and the following threads have to be squashed only when the speculatively forwarded value has been consumed. The presence of the recovery code minimizes the impact of an intra-thread mis-speculation, thus more aggressive instruction scheduling are allowed.

Control Speculation

Control speculation is used to speculatively move instructions across dependent branch instructions. An example of intra-thread control speculation is shown in Figure 5.3. The store instruction *store [r1], r2* in Figure 5.3(a) is synchronized and a signal instruction is inserted. Note that a signal instruction *signal r0, r0* is also inserted on the alternative path. Register *r0* has the value of zero, and the execution of this instruction simply informs the consumer thread that no data is



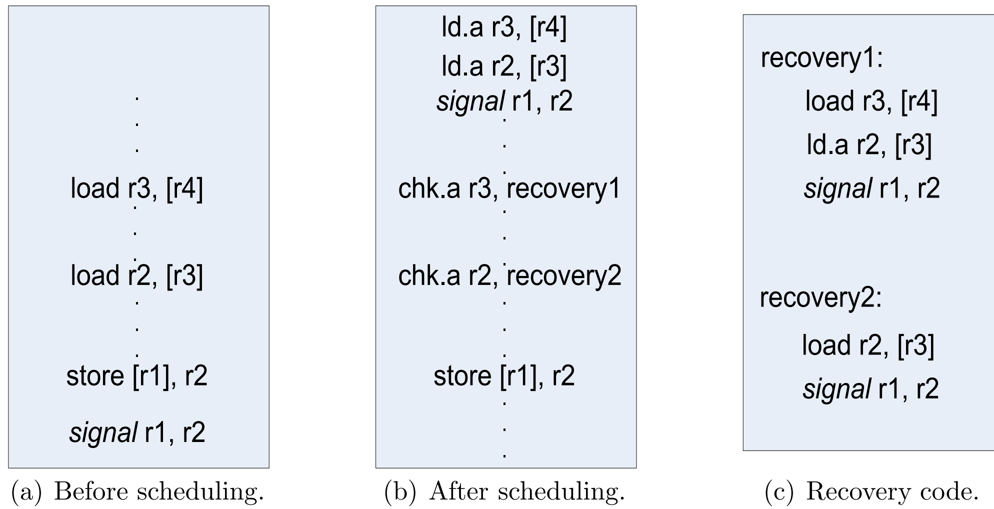
(a) Before scheduling.

(b) After scheduling.

Figure 5.3: Control speculation used in scheduling for memory dependence.

forwarded, so that the consumer thread no longer has to wait for it.

When we schedule the signal instruction across the branch instruction, the load instruction *load r2, [r3]* it depends on also needs to be scheduled. In such case, the load is changed into *ld.s*, and a check instruction *chk.s* is inserted accordingly, as shown in Figure 5.3(b). The similar recovery code as the one used in data speculation (in Figure 5.2(c)) is invoked when the speculative load causes an exception. If the alternative path is taken, *signal r0, r0* will be executed, so that the consumer thread will receive the same signal again and know that the data received previously is wrong.



(a) Before scheduling. (b) After scheduling. (c) Recovery code.

Figure 5.4: Cascaded speculation with recovery support.

Cascaded Speculation

During aggressive instruction scheduling, it is common that a signal depends on multiple loads. So that multiple loads need to be scheduled along with the signal. When multiple loads are speculatively scheduled, it is possible that there are dependences among them. In Figure 5.4(a), there are two loads that signal depends on: *load r2 [r3]* and *load r3 [r4]*. The second load also depends on the first load since the first load needs the value of *r3* that is loaded by the first load. These two loads are scheduled along with signal and are changed into speculative loads due to possible dependence, as shown in Figure 5.4(b).

Two pieces of recovery codes are generated accordingly in Figure 5.4(c). *Recovery 1* is for the first speculative load, while *recovery 2* is for the second speculative load. Since the second load depends on the first load, it is executed speculatively

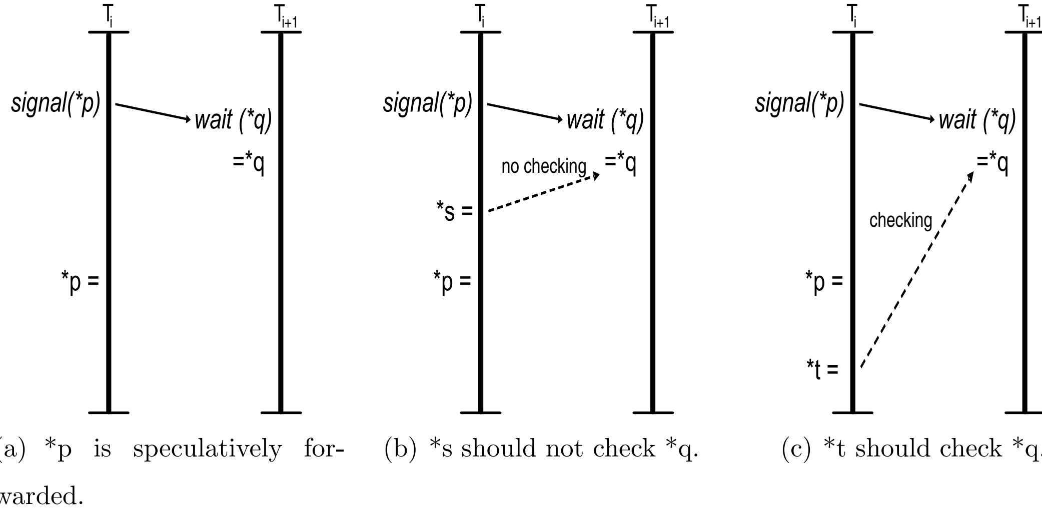
in *recovery 1*. If the second load fails, either in the place where it is scheduled to, or in *recovery 1*, *recovery 2* is invoked to reload the value of *r2*. Signals are inserted into both recovery codes. In the worst case, both speculative loads fail and both recovery codes are invoked. The data will be forwarded three times, and only the one forwarded by the last signal is correct.

As described previously, the consumer thread knows that a mis-speculation is detected by the producer thread each time it receives a signal with the same *sid* as the one received earlier. Squash is only required when the incorrectly forwarded data has been consumed.

5.2.2 Working with TLS

Unlike the scheduling for register dependence, the scheduling of memory dependence may interact with the underlying TLS support. The details of how scheduling and TLS work together need a closer examination.

We use the example in Figure 5.5 to demonstrate the potential problem. In Figure 5.5(a), there is a strong data dependence between the store of **p* and the load from **q*. The signal instruction used to synchronize this dependence is hoisted to an earlier point. The forwarded data by the signal is read by the load if *(p == q)*. Assume another store instruction, store of **s*, is executed between the signal and the store of **p*, and *(s == q)*, as shown in Figure 5.5(b). Under TLS, this store will check possible dependence violations caused by the following threads.



(a) *p is speculatively forwarded.

(b) *s should not check *q.

(c) *t should check *q.

Figure 5.5: Speculative forwarding.

However, in this case, the load of **q* should not cause a mis-speculation although it accesses the same memory location as the store of **s*. The reason is that the data loaded by **q* is newer than the one stored by **s*. On the other hand, assume there is another store, store of **t*, which is executed after the store of **p*, as shown in Figure 5.5(c). If $(t == q)$, this store should detect a mis-speculation since it provides a newer version of data than the one used by the load of **q*.

To summarize, the lifetime of a forwarded data is divided into two distinct stages after instruction scheduling. The first stage starts when the speculatively scheduled signal instruction is executed. It ends when the store instruction that needs to be synchronized is executed. In this stage, a forwarded data is only exposed for the *intra-thread* dependence checking performed by the producer thread. If an intra-thread mis-speculation is detected, either the forwarded value or address is wrong. If the forwarded data has already been accessed by the consumer thread,

an inter-thread mis-speculation is detected as well, and the consumer thread has to be squashed. The second stage starts right after the execution of the synchronized store instruction, and it ends when the consumer thread becomes non-speculative. In this stage, the forwarded data is exposed for the *inter-thread* dependence checking just as the same as an ordinary inter-thread data speculation. And this dependence checking is performed by the underlying TLS hardware.

Distinguishing these two stages in the lifetime of a forwarded data is the key for instruction scheduling of memory dependence to work properly under TLS. In our thread execution model, the compiler explicitly inserts an instruction called *expose* after the synchronized store. Before *expose* is executed, the forwarded data is kept in a buffer on the consumer thread side so that it will not be checked for inter-thread dependence violation. The execution of *expose* will inform the consumer thread to explicitly expose the forwarded data for inter-thread dependence checking.

5.3 Reduction Transformation

A reduction operation iteratively summarizes information into a single variable called the reduction variable [35]. The presence of reduction variables causes inter-thread dependences, and serializes parallel execution. Such serialization can become potential performance bottlenecks when nested loops are involved. The example in Figure 5.6(a) shows a reduction variable *sum* defined in a nested loop. During the parallel execution of the *outer loop*, in thread i, the definition in the

(a) A reduction variable	(b) Synchronizing the reduction variable	(c) Traditional reduction elimination

`while(cond1) {`
`   while(cond2) {`
`      sum++;`
`   }`
`}`
	`while(cond1) {`
	`   wait(sum);`
	`   while(cond2) {`
	`      sum++;`
	`   }`
	`   signal(sum);`
	`}`
		`while(cond1) {`
		`   sum[i]=0;`
		`   while(cond2) {`
		`      sum[i]++;`
		`   }`
		`}`
		`while(cond1) {`
		`   sum+=sum[i];`
		`}`

Figure 5.6: Reduction elimination.

last iteration of the inner loop is used by thread $i+1$ in the first iteration of the inner loop. This creates an inter-thread data dependence that must be synchronized as shown in Figure 5.6(b). However, such synchronization can potentially serialize parallel execution, since only after the inner loop in thread i finishes, can the inner loop in thread $i+1$ start.

5.3.1 Reduction Elimination

In traditional parallelizing compilers [35], reduction variables are eliminated through a process called *reduction elimination*, in which multiple independent variables are created and stored in an array as shown in Figure 5.6(c). Because each thread stores reduction variable in a private location *sum[i]*, inter-thread data dependences caused by *sum* are eliminated, thus the threads can be executed completely in parallel without any synchronization required. The final result of the reduction

(a)	(b)	(c)

`while(cond1) {`
`  while(cond2) {`
`    sum++;`
`  }`
`  ...`
`  =sum;`
`  ...`
`}`
	`while(cond1) {`
	`  while(cond2) {`
	`    sum[i]++;`
	`  }`
	`  ...`
	`  wait(sum);`
	`  =sum+sum[i];`
	`  ...`
	`  sum+=sum[i];`
	`  signal(sum);`
	`}`
		`while(cond1) {`
		`  while(cond2) {`
		`    sum0++;`
		`  }`
		`  ...`
		`  wait(sum);`
		`  =sum+sum0;`
		`  ...`
		`  sum+=sum0;`
		`  signal(sum);`
		`}`

(a) Using intermediate result of reduction variable	(b) Reduction transformation with explicit forwarding	(c) Replacing sum[] with sum0

Figure 5.7: Reduction variable with an additional use.

operation is computed after parallel execution ends.

In practice, it is not necessary that each thread needs to be assigned a distinct private variable. The number of private variables created usually equals to the number of active threads, so that each of them can access a different variable, and the same variable can be shared by multiple threads as long as they are not active simultaneously.

5.3.2 Partial Reduction Elimination

Although reduction elimination is effective in removing inter-thread dependences, the application of this technique is limited due to one important constraint—intermediate results of the reduction operation cannot be used anywhere in the parallelized loop.

In the example shown in Figure 5.7(a), the intermediate result of the reduction variable *sum* is used in the outer loop. In order to retrieve the intermediate result, the reduction variable must be communicated between the parallel threads. Fortunately, we can perform partial reduction elimination and update the value of *sum* only once in the outer loop, as shown in Figure 5.7(b), where all uses of the reduction variable *sum* in the outer loop is replaced with $sum + sum[i]$. Although the reduction variable is not eliminated, its impact on the parallel performance is greatly reduced, as we can see that the critical forwarding path between the update of *sum* in a thread and use of it in the successor thread becomes relatively small.

Another benefit of this transformation is that the summation step can be eliminated. Since the reduction variable is explicitly communicated between threads during the parallel execution, its final value is immediately available after the parallel execution.

Furthermore, because TLS provides implicitly renaming for the same variable used in different threads, we can avoid the creation of the array $sum[\,]$ and use a single scalar variable $sum0$ to hold the partial result of the reduction operation, as shown in Figure 5.7(c).

Just like any other variables that are communicated through synchronization, communication of such reduction variables can be further improved with instruction scheduling that is described in Section 5.2. For instance, $signal(sum)$ can

be hoisted to an earlier point, so that the critical path length due to *sum* can be further reduced.

5.3.3 Aggressive Reduction Transformation I

Reduction transformation described in Section 5.3.2 effectively reduces the length of critical forwarding path. However, not all usage patterns of reduction variables can be optimized with this transformation. In the example shown in Figure 5.8(a), the *signal* instruction cannot be scheduled before the inner loop because it depends on the value of $sum0$ that is computed by the inner loop, and *wait* instruction cannot be scheduled after the inner loop because it is used to guard a branch instruction. As a result, the critical forwarding path introduced by the communication of *sum* is still very long after reduction transformation.

However, the outcome of the branch instruction guarded by the reduction variable is often predictable, and we can exploit this predictability to postpone the use of the reduction variable till after the completion of the inner loop. In the example shown in Figure 5.8(b), the branch is predicted as not-taken, and it is moved across the inner loop and executed as a verification. In the original location of this branch, both $sum0$ and x are saved, so that they can be used later in the verification. The use of *sum* is delayed so that the critical forwarding path is reduced. When the value of *sum* becomes available, and the branch is proved to be mis-predicted, the thread must be squashed and an un-optimized version of code

while(cond1) {	while(cond1) {	while(cond1) {	while(cond1) {
wait(sum);	sum0'=sum0;	wait(sum);	while(cond2) {
if(sum+sum0>x)	x'=x;	while(cond2) {	sum0++;
work1;	work2;	sum0++;	work1;
else	...	if(sum+sum0>100)	}
work2;	while(cond2) {	return;	...
...	sum0++;	work1;	wait(sum);
while(cond2) {	}	}	if(sum+sum0>100)
sum0++;	...	...	recovery;
}	wait(sum);	sum+=sum0;	sum+=sum0;
...	if(sum+sum0'>x')	signal(sum);	signal(sum);
sum+=sum0;	recovery;	}	}
signal(sum);	sum+=sum0;		
}	signal(sum);		
	}		
(a) Used to determine a branch outcome	(b) Predicting branch outcome then verifying	(c) Used in the inner loop	(d) Predicting branch outcome then verifying in the outer loop

Figure 5.8: Using the intermediate result of a reduction variable to determine a branch outcome.

must be executed [58]. The squash/recovery mechanism that enables this aggressive optimization is already available in TLS, thus no extra hardware support is required.

5.3.4 Aggressive Reduction Transformation II

The aggressive transformation described in Section 5.3.3 does not handle all usage patterns of *sum* within the loop: the reduction variable can be used in the inner loop, as shown in Figure 5.8(c). In order to reduce the critical forwarding path introduced by such usage, the branch in the inner loop has to be moved to the

outer loop. Is it possible to make such a code transformation and to guarantee that all mis-predictions are detected? The answer is yes, and the key to this transformation is that most reduction operations are monotonic. If the reduction variable is monotonically increasing or decreasing and the branch is to test whether it is greater or less than a certain loop invariant, the verification can be delayed until the inner loop is complete. In our example, if the condition $sum+sum0 > 100$ is true in the inner loop, it must also be true for the test in the outer loop. So that mis-predictions can always be detected by the delayed verification in the outer loop, as shown in Figure 5.8(d).

5.3.5 Transformation Algorithm

Figure 5.9 shows the algorithm for the reduction transformation of a particular loop. An operation with a form of $var = var\ opr\ x$ is recognized as a reduction operation, where the operator *opr* is limited to be an *addition* or *subtraction* under current implementation, and the second operand x can be either a variable or a constant. The variable *var* used in the reduction operation is recognized as a reduction variable. All reduction variables in the loop are identified through pattern matching. In addition, each of identified reduction variable has to satisfy the following requirements:

1. There is no additional *definition* of a reduction variable in the loop besides the reduction operation.

2. The reduction variable is not alias with any other variables or procedure calls in the loop.

The partial reduction elimination is applied first. For each particular reduction variable *var*, a new variable *var0* is created to hold the partial result of *var* in each loop iteration. The result of *var* is accumulated at the end of each loop iteration, either before a loop continuation or a loop break. An additional use of the intermediate result of *var* is replaced with the use of *var+var0*, and *var* used and defined in a reduction operation is replaced with r0. Note that there could be multiple additional uses and reduction operations, and all of them are transformed uniformly with this algorithm.

After partial reduction elimination is applied, aggressive reduction transformation is performed. The targets of aggressive transformation are those highly predictable conditional branches that use the reduction variable *var*. Either aggressive transformation I or II is applied depending on whether a branch is in the current loop or in an inner loop.

If a branch is in the current loop, it is replaced with a *goto* to the predicted target, and all variables used to compute the branch outcome are saved to temporaries. The verification code is generated by copying the original branch and replacing all variables with the saved temporaries. A special *recovery* instruction is inserted into the verification code on the alternative path that is predicted not to taken. If the verification detects a mis-prediction, that is, the alternative path

```
R <- find all reduction variables in the loop;
for each reduction variable var in R {
    \* 1. partial reduction elimination *\
    create a new variable var0;
    for each reduction operation of var
        replace var with var0;
    for each additional use of var
        replace var with var + var0;
    insert var0 < −0 to the beginning of the loop;
    insert var < −var + var0 before any continuation/break in the loop;
    \* 2. aggressive reduction transformation *\
    for each highly predictable conditional branch that uses var {
        if (the branch is in the current loop nest) {
            insert var0' < −var0, x'1 < −x1, x'2 < −x2, ..., x'n < −xn before
              the branch, where x1, x2, ..., xn are the variables used
              to compute the branch outcome;
            insert a prediction verification before each var < −var + var0,
              where each verification test is a copy of the branch
              with all variables replaced with the saved ones;
            replace the branch with a goto to the predicted target;
        }
        else if (var is monotonically increased or decreased and
                it is compared with a loop invariant) {
            replace the branch with a goto to the predicted target;
            insert a prediction verification before each var < −var + var0,
              where each verification test is a copy of the branch;
        }
    }
}
```

Figure 5.9: Algorithm for reduction transformation.

is actually taken, this *recovery* instruction will be executed and causes the squash of the current thread. A verification code will be inserted before *var<-var+var0* (inserted by the partial reduction elimination) in each loop continuation/break point that can be *reached* by this branch.

If a branch is in an inner loop, the transformation is applied when the reduction variable is monotonically increasing or decreasing. However, there is one additional requirement for the branch to be correctly transformed, that is, a mis-prediction of that branch in the inner loop should always be detected by the verification code in the outer loop.

Let's first examine the case where the reduction variable is monotonically increasing. In such a case, the reduction variable has the following property: the value of *var* at the prediction point in the inner loop is always no greater than the value of *var* in the outer loop. Now assume that the prediction in the inner loop is *(var <= y)*, where *y* is the loop invariant. If a mis-prediction happens, that is, *(var > y)*, the outer loop will also detect that *var* is greater than *y* since the value of *var* observed by the outer loop is no smaller than the value of *var* in the prediction point. As a result, such a mis-prediction can be always detected in the outer loop. Now let's assume that the prediction in the inner loop is *(var >= y)*. A mis-prediction happens when *(var < y)*. However, such mis-prediction may not be detected in the outer loop since the value of *var* in the outer loop is no smaller than the value of *var* in the prediction point, and it is possible that *(var >= y)*

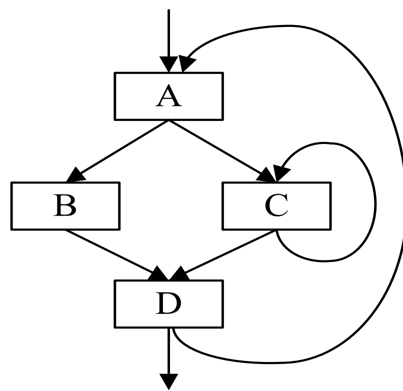
is true in the outer loop. In summary, when a reduction variable is monotonically increasing, a failure of the prediction of *(var <= y)* or *(var <= y)* in the inner loop is guaranteed to be detected in the outer loop. So the additional requirement for the branch is that it should has one of the following tests: *(var <= y)*, *(var < y)*, *(var >= y)*, or *(var > y)*, and the corresponding prediction should be: *(var <= y)*, *(var < y)*, *(var < y)*, or *(var <= y)*.

Similarly, when the reduction variable is monotonically decreasing, the branch should also has one of the following tests: *(var <= y)*, *(var < y)*, *(var >= y)*, or *(var > y)*. However, in contrast to the prediction that is made for a monotonically increased reduction variable, here the prediction should be: *(var > y)*, *(var >= y)*, *(var >= y)*, or *(var > y)* accordingly.

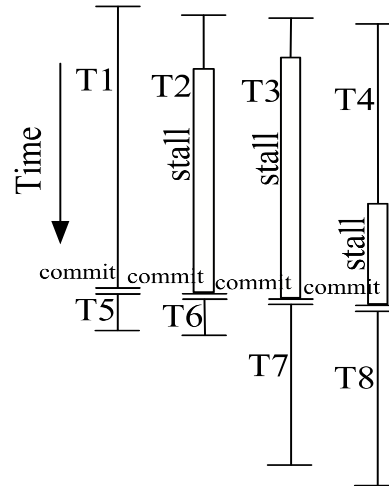
Such branch prediction is often highly accurate. The reason is that when a reduction variable is monotonically increasing, most of time it is less than a loop invariant, while when a reduction variable is monotonically decreasing, most of time it is greater than a loop invariant. Consequently, the branch outcome is highly predictable.

5.4 Iteration Merging for Load Balancing

In TLS, to preserve the sequential semantics, speculative threads must be committed in order. Thus, if a short thread that follows a long thread completes before the long thread, it must stall until the long thread completes. When the workload



(a) CFG of a nested loop.

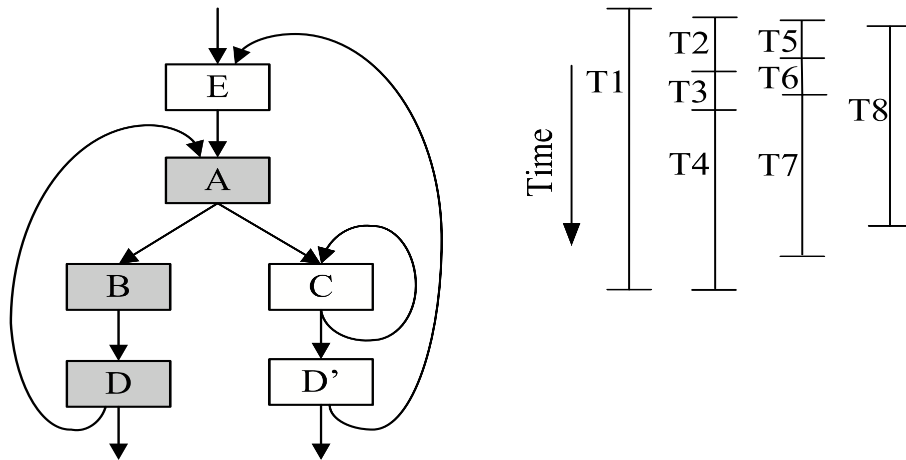


(b) Parallel execution.

Figure 5.10: Load imbalance.

is not balanced between parallel threads, such waiting time can be significant to cause the performance degradation.

Figure 5.10(a) shows the CFG of a doubly nested loop. Assume the outer loop is selected for parallelization. A thread can take two possible paths: the path $A \rightarrow B \rightarrow D$ on the left, and the path $A \rightarrow C \rightarrow ... \rightarrow C \rightarrow D$ on the right. However, a thread that takes the right path is much longer than a thread that takes the left path due to the execution of the inner loop on the right path. This causes the load imbalance problem during parallel execution as shown in Figure 5.10(b). Thread $T1$, $T4$, $T7$ and $T8$ take the long path, while thread $T2$, $T3$, $T5$, $T6$ take the short path. The short thread $T2$ and $T3$ finish their execution much earlier than the long thread $T1$, however, they have to wait until thread $T1$ commits its results in order to preserve the correct sequential semantics.



(a) CFG after iteration merging. (b) Parallel execution.

Figure 5.11: Iteration merging.

5.4.1 Iteration Merging

The load imbalance becomes a severe problem when there are variations in the thread size among *consecutively* executed threads. One possible way to solve this problem is to combine multiple consecutive short iterations with a long iteration to make the workload more balanced. However, it is often difficult to statically determine how many consecutive short iterations should be merged. Instead, we propose a compiler transformation technique called *iteration merging* to transform the loop in such a way that the number of iterations to be merged are determined *dynamically*.

The transformation is illustrated in Figure 5.11(a). In order to merge multiple short iterations with a long iteration shown in Figure 5.10(a), a new inner loop is

constructed for the path $A \rightarrow B \rightarrow D$, which is marked by the shadowed blocks. For all basic blocks that can be reached from the outside of this inner loop, tail duplications are needed in order to eliminate side entries. In this example, block D′ is tail duplicated and inserted in the outer loop. A new block E is also inserted to the beginning of the outer loop, and only contains a trivial unconditional branch that transfers the control flow to A. Later a $fork$ instruction will be inserted to E in order to spawn a new thread at runtime.

After iteration merging, the execution of a thread always starts with the short iteration, and multiple short iterations could be executed by the newly formed inner loop until the control transfers from block A to block C, that is, a long iteration is encountered. This long iteration is executed by the *same* thread right after the execution of multiple short iterations. As a result, more balanced workloads can be achieved after merging, as shown in Figure 5.11(b).

5.4.2 Transformation Algorithm

Figure 5.12 show the algorithm for iteration merging of a particular loop. Before iteration merging is applied, we have to determine whether a loop has unbalanced workloads. A typical loop that has load imbalance problem often has time consuming inner loops or procedure calls that leads to long iterations. The ratio between the size of long iteration and short iteration is estimated by the following equation:

$$ratio = \frac{coverage \times (1 - p)}{(1 - coverage) \times p} \tag{5.1}$$

```
L <- find the inner loop or procedure call with the maximal ratio;
bb_set <- find all basic blocks that can be reached by L;
for each basic block bb in bb_set that has a side entry {
    bb_set <- tail_duplicate(bb_set, bb);
}
create a new loop head H for the outer loop;
for each loop tail tb in bb_set {
    re-direct tb to H;
}
```

Figure 5.12: Algorithm for iteration merging

where *coverage* is the coverage of the inner loop or procedure call in the outer loop that needs to be transformed, and p is the probability that the inner loop or procedure call is executed in each outer loop iteration. Intuitively, an inner loop or procedure call with higher coverage and lower execution probability is more likely to cause the larger variations in the iteration size of the outer loop. The one that has the largest *ratio* is selected for transformation.

During transformation, a set of basic blocks that can be reached from the inner loop or procedure call are identified and stored into *bb_set*. For each basic block *bb* in *bb_set* that has a side entry, tail duplication is performed such that all basic blocks that can be reached by *bb* are duplicated. The *bb_set* is updated accordingly after each tail duplication. Finally, a new loop head H is created for the outer loop, and all loop tails in *bb_set* are re-directed to this new loop head.

5.5 Case Study with Compression Applications

To demonstrate the importance of the proposed optimization techniques, we conduct a case study of compression applications for which TLS typically achieves modest speedup. Two compression benchmarks BZIP2 and GZIP from SPEC2000 benchmark suite are selected for an extensive study [67].

BZIP2

BZIP2 [8] represents one class of compression applications that uses a block-based algorithm. It divides the input data into blocks of the size N ranging from 100k to 900k bytes, and processes the blocks sequentially. While it is possible to process different blocks in parallel, the huge size of the speculative data modified by each thread often exceeds the capacity of speculative buffer provided by TLS, which is typically from 16k to 32k bytes. The frequent stalls due to the buffer overflow inhibit most of the performance gains from TLS.

During the compression of each block S of size N, the most time consuming part is Burrows-Wheeler Transform (BWT). It forms N rotations of a block by cyclically shifting S, and sorts these rotations lexicographically. Bucket sort is used in the main sorting phase. The buckets are organized as a two-level hierarchical structure. The big bucket in the outer level contains all rotations starting with the same character, while the small bucket in the inner level contains all rotations starting with the same two characters.

Consequently, a two-level nested loop is used to traverse each bucket to sort all rotations inside. The outer loop seems an ideal target for parallel execution since sorting of big buckets can be done independently. However, in order to speedup the sequential algorithm, the information about the sorting of the current bucket is kept in the global data structures such as *quadrant* and used in the sorting of following buckets to avoid redundant computations. Also, the results of sorting the current big bucket are used to update other unsorted buckets. As a result, those optimizations for sequential algorithm introduces inter-thread dependences that are undesirable for parallel execution. On the other hand, the performance of the inner loop is mainly limited by the reduction-like variable *workDone*. Reduction elimination cannot be applied here since *workDone* is also used by non-reduction operations in the loop. The sorting of each small bucket is done by calling *qSort*. Since *qSort* is not always called in every inner loop iteration, it introduces unbalanced load among threads.

The compression algorithm also includes other phases such as run-length encoding, move-to-front encoding, and Huffman encoding. The performance of the main loops in those phases are typically limited by long critical forwarding paths that are hard to optimize.

The decompression phase in BZIP2 has much lower coverage than in the compression phase. Similar to compression, decompression is performed for one block at a time. Decompression of multiple blocks cannot run in parallel due to the

size limitation of the speculative buffer. Most loops in the decompression phase is sequential due to the fact that the decoding of a character is completely dependent on the previous characters.

GZIP

GZIP [71] represents another class of compression applications that uses a dictionary-based algorithm. The input data is scanned sequentially, once a repeated string is detected, it is replaced by a pointer to the previous string. A hash table is used for detecting a repeated string. All input strings of length three are inserted in the hash table.

Two versions of the algorithm are implemented. *Deflate_fast* is a simplified version, which is fast but with low compression ratio. The main loop iterates through all input characters. Each time a match is found, it is selected immediately. The main performance limitation is caused by the use of global variables such as *lookahead* and *strstart*. *Deflate*, a more complex and time consuming version, uses a technique called lazy evaluation in order to find a longer match. With lazy evaluation, the match is not selected immediately. Instead, it is kept and compared with the matches for the next input string for a better choice. However, the use of current match in the next matching step causes additional data dependences. Both *deflate* and *deflate_fast* call *longest_match* to find the longest match among all strings with the same hash index. The average iteration size of the main loop in

longest_match is typically small due to the facts that most of strings do not match with the current string and a fast check is used to avoid unnecessary comparison.

Similar to BZIP2, the decompression phase in GZIP has a much lower coverage than the compression phase. The decompression is performed sequentially since the decoding of the current character depends on the characters decoded previously. As a result, it is hard to extract parallel threads in the decompression phase.

Chapter 6

Performance Evaluation

6.1 Simulation Framework

The compiler techniques described in the previous chapters are evaluated on a simulator that is built upon Pin [43]. It models a CMP with four single-issue in-order processors. The configuration of our simulated machine model is listed in Table 6.1. Each processor has a private L1 data cache, a write buffer, an address buffer, and a communication buffer. The write buffer holds the data that are modified by a speculative thread [62]. The memory address accessed by any exposed load from a speculative thread is kept in the address buffer, which is checked by each store from previous threads for detecting possible inter-thread dependence violations. The communication buffer is used for storing the data forwarded by the immediate predecessor thread during synchronization. All four

Table 6.1: Simulation parameters.

Issue Width	1	Main Memory Latency	50 cycles
L1-Data Cache	32KB, 2-way, 1 cycle	Commu. Buffer	128 entries, 1 cycle
L2-Data Cache	2MB, 4-way, 10 cycle	Commu. Delay	10 cycles
Cache Line Size	32B	Thread Spawning	10 cycles
Write Buffer	32KB, 2-way, 1 cycle	Thread Squashing	10 cycles
Addr. Buffer	32KB, 2-way, 1 cycle		

processors share a L2 data cache, which is used as a *safe* storage to store the committed non-speculative data.

All simulations are performed using the *ref* input set. To reduce the simulation time, a simple sampling technique is used. For each parallelized loop, it is simulated up to 1 thousand invocations, and each invocation is simulated up to 0.1 million iterations. For each non-loop thread, it is simulated up to 0.1 million invocations. Overall, by using this sampling technique, we are able to simulate up to 10 billion instructions while covering all parallel threads in the program.

6.2 Benchmark Description

We select SPEC2000 benchmark suite for this evaluation. 11 integer benchmarks (*mcf*, *crafty*, *twolf*, *gzip*, *bzip2*, *vpr*, *vortex*, *parser*, *perl*, *gap*, and *gcc*) and 4 floating point benchmarks (*ammp*, *art*, *equake*, and *mesa*) are studied. All of them are written in C, and represent a wide class of general-purpose applications that are hard to parallelize by traditional parallelizing compiler.

Table 6.2: Loop statistics.

	Application Name	Number of Loops	Maximum Nest Level
CINT2000	mcf	47	4
	crafty	1,905	9
	twolf	839	7
	gzip	185	6
	bzip2	153	9
	vpr-place	387	5
	vpr-route	387	6
	vortex	181	7
	parser	498	10
	perl	748	8
	gap	1,680	10
	gcc	2,592	12
CFP2000	ammp	68	10
	art	373	6
	equake	88	5
	mesa	868	6
Average		687	7.5

6.3 Loop Selection

In this section, the effectiveness of loop selection is evaluated. We list the statistics for the loops in all benchmarks in Table 6.2. The level of loop nest is measured based on the *loop graph* described in Chapter 3. The loop graph is built dynamically by using *ref* input set, and all the recursive edges in the graph have been removed.

From Table 6.2, we can see that both integer and floating point benchmarks have complex loop nests. On average, each of them has 687 loops with an average nest level of 7.5. In the extreme case of *gcc*, it has 2592 loops with the maximal nest level of 12. Consequently, it is difficult to select loops from such complex loop nests without a systematic approach.

As described in Chapter 3, the key to the success of loop selection is to accurately predict the speedup for each individual loop. We have designed three loop speedup estimation techniques according to different strategies used in estimating the synchronization cost. Here is the short description of these three techniques. The first estimation technique (estimation I) uses a simple estimation strategy that counts the number of instructions involved in inter-thread data dependence to approximate the impact of synchronization. The second estimation technique (estimation II) takes into consideration the timing of executing the producer instruction and the consumer instruction of each dependence pair, and select the one that causes the longest stall of the consumer instruction to estimate the synchronization cost. The third technique (estimation III) takes one step further by considering the effects of instruction scheduling, so that the timing estimation of the producer instruction and the consumer instruction is based on the dependence graph.

6.3.1 Statistics for the Selected Loops

All three speedup estimation techniques are implemented in the loop selection phase, and three sets of loops are selected. The statistics for the selected loops are listed in Table 6.3, Table 6.4, and Table 6.5 respectively. For comparison, we also use the speedup obtained from simulation as the perfect estimation. The loops selected by using this perfect estimation are treated as the upper bound of loop

Table 6.3: Loop statistics of estimation I.

	Application Name	Number of Loops	Average Nest Level	Average Iteration Size
CINT2000	mcf	8	1.75	214
	crafty	6	4.16	265
	twolf	16	1.93	4,679
	gzip	7	2.85	307
	bzip2	18	3.77	346
	vpr-place	4	3	248
	vpr-route	18	2.77	370
	vortex	10	3.9	1,971
	parser	42	4.64	207
	perl	10	3.9	543
	gap	10	3.8	1,026
	gcc	81	3.53	281
CFP2000	ammp	25	4.28	334
	art	19	4.15	325
	equake	8	2	220
	mesa	4	3	3,530
Average		18	3.34	929

selection. The statistics for these perfectly selected loops are listed in Table 6.6. Note that, the nest level of a loop in the loop graph is defined as the length of the longest path from the outer most loop to that particular loop. So the *higher* nest level a loop has, the *more deeply* it is nested. The loop iteration size is measured by counting the number of dynamic instructions.

Based on the statistics shown in these tables, we make the following observations:

First, estimation I tends to select loops with lower nest level than other two techniques. Those loops have larger iteration size with 929 dynamic instructions on average. As we described previously, this simple estimation only considers the instructions directly involved in the inter-thread data dependence, and the number

Table 6.4: Loop statistics of estimation II.

	Application Name	Number of Loops	Average Nest Level	Average Iteration Size
CINT2000	mcf	4	4.5	235
	crafty	7	5.17	127
	twolf	4	2.25	209
	gzip	3	2.33	322
	bzip2	13	3.84	218
	vpr-place	2	3.5	154
	vpr-route	9	2.88	243
	vortex	6	3.83	221
	parser	10	4.8	149
	perl	5	4.8	257
	gap	3	4	119
	gcc	19	4.2	186
CFP2000	ammp	15	4.26	337
	art	17	3.94	343
	equake	4	2.5	233
	mesa	5	6	146
Average		8	3.9	219

of such instructions becomes relatively small when the iteration size becomes large. As a result, those loops are predicted to have high speedup and are more likely be selected by estimation I.

Second, estimation II usually selects loops with higher nest level. Those loops have much smaller iteration size with 219 dynamic instructions on average. The number of loops selected by this technique is much less than other two techniques. Since this technique ignores the effects of possible thread optimizations such as instruction scheduling, the small number of loops selected by this technique indicates that, without thread optimizations, few loops achieve good performance in these benchmarks.

Third, estimation III tends to select loops with the average nest level higher

Table 6.5: Loop statistics of estimation III.

	Application Name	Number of Loops	Average Nest Level	Average Iteration Size
CINT2000	mcf	13	2.38	174
	crafty	5	4.11	371
	twolf	19	2.26	244
	gzip	6	2.83	308
	bzip2	19	3.84	410
	vpr-place	3	3	250
	vpr-route	19	3	968
	vortex	9	4	647
	parser	40	4.35	297
	perl	9	4.77	372
	gap	7	3.8	213
	gcc	98	3.91	277
CFP2000	ammp	21	3.85	464
	art	25	3.84	327
	equake	9	1.77	242
	mesa	4	2.66	373
Average		19	3.4	371

than estimation II after the effects of thread optimization have been considered. The average iteration size with 371 dynamic instructions is also larger than estimation II, but is much smaller than estimation I. However, the number of loops selected by this technique is similar as estimation I.

Finally, for perfect estimation, it tends to select more loops, especially for *gcc* and *parser*, both of which have significant number of loops. We can also see that, to achieve the best performance, the average nest level of the selected loop is 3.36. Neither benchmark achieves this best performance by simply selecting either outer loops or inner loops.

Table 6.6: Loop statistics of perfect estimation.

	Application Name	Number of Loops	Average Nest Level	Average Iteration Size
CINT2000	mcf	12	2.21	132
	crafty	13	4.09	176
	twolf	14	2.17	644
	gzip	8	2.76	195
	bzip2	21	3.84	624
	vpr-place	5	2.8	257
	vpr-route	17	2.92	667
	vortex	11	3.97	389
	parser	54	4.41	812
	perl	13	4.84	324
	gap	11	3.78	356
	gcc	124	3.85	278
CFP2000	ammp	18	3.71	684
	art	22	3.97	426
	equake	8	1.76	478
	mesa	6	2.72	942
Average		22	3.36	462

6.3.2 Performance Impact of Loop Speedup Estimation

To study the performance impact of different speedup estimation techniques, we generate loop threads for the selected loops. Instruction scheduling is then applied on these loop threads to improve the synchronization due to *register* dependence [69]. We run the final parallelized code on our simulator. The program speedup is shown in Figure 6.1 and the loop coverage is shown in Figure 6.2. Note that, the coverage is measured by counting the number of cpu cycles when a fully optimized sequential version of code is simulated.

Estimation I, despite its simplicity, achieves comparable performance as estimation III for over half of benchmarks (especially for floating point benchmarks). For

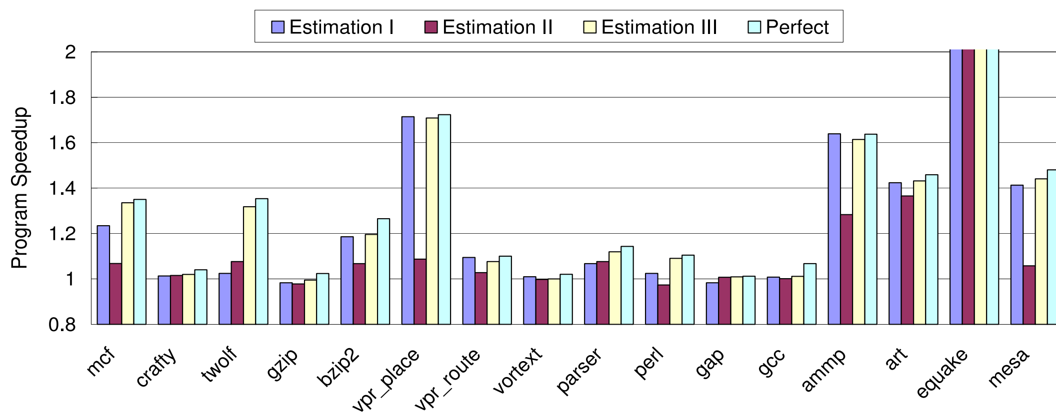


Figure 6.1: The program speedup of loop threads.

some benchmarks such as *mcf*, *twolf*, and *perl*, it tends to over-estimate the performance of those loops with high coverage, and cannot achieve similar performance improvements as estimation III.

Estimation II, without considering the effect of optimizations, usually selects loops with lower coverage. For some benchmarks such as *mcf*, *vpr_place*, and *mesa*, such low coverage of the selected loops results in poor program performance. Overall, this estimation technique is too conservative in selecting loops.

Estimation III achieves the best performance improvement across all benchmarks. Such improvement is close to the perfect estimation with a difference less than 5%.

Finally, for some benchmarks such as *crafty* and *gap*, the loop coverage is very low (below 20%) even under the perfect estimation. This strongly indicates that the loop-level parallelism is limited in such programs, and it is necessary to exploit parallelism beyond loops.

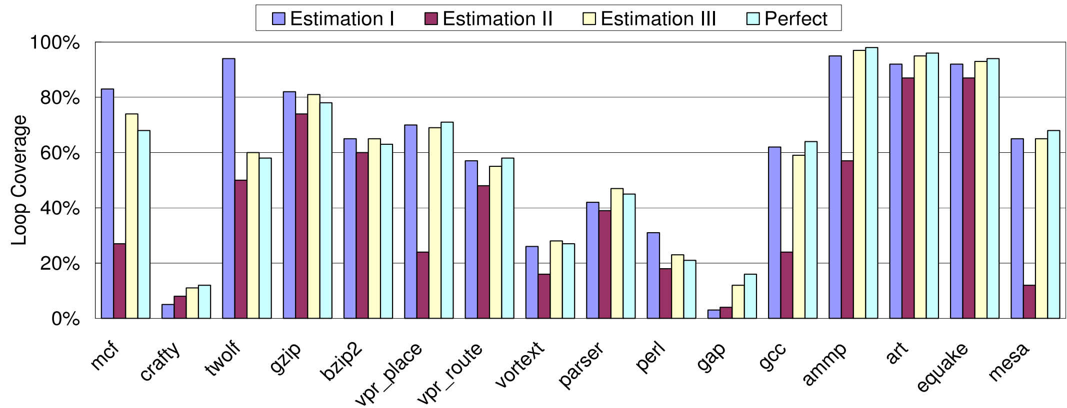


Figure 6.2: The coverage of loop threads.

6.4 Non-loop Partitioning

In this section, we evaluate the effectiveness of the non-loop partitioning algorithm described in Chapter 4. Non-loop partitioning is performed after loop threads are extracted in the loop selection phase. Table 6.7 lists the statistics of the non-loop threads extracted from each benchmark.

Overall, we extract 255 non-loop threads from 9 programs. For the remaining programs (*mcf*, *gzip*, *vpr-place*, *ammp*, *art*, *equake*, *mesa*), non-loop threads are not created since they have substantial amount of loop-level parallelism. The extracted non-loop threads have reasonable size with 250 dynamic instructions on average. The program speedup achieved by using non-loop threads is shown in Figure 6.3, and the coverage of non-loop threads is shown in Figure 6.4. Although the achieved speedup (1.04 on average) is not as impressive as loop threads, non-loop threads still helps especially for those programs that have limited loop-level parallelism. For instance, the program speedup for *crafty*, *gap* and *vortex* are all

Table 6.7: Non-loop thread statistics.

	Application Name	Number of Non-Loop Threads	Average Thread Size
CINT2000	mcf	0	0
	crafty	33	238
	twolf	19	372
	gzip	0	0
	bzip2	11	274
	vpr-place	0	0
	vpr-route	9	234
	vortex	22	278
	parser	39	209
	perl	21	267
	gap	34	221
	gcc	57	197
CFP2000	ammp	0	0
	art	0	0
	equake	0	0
	mesa	0	0

around 1.05, and non-loop threads in these programs account for more than 20% of total execution time.

One important factor that affects the performance of non-loops is thread predictability. In our thread execution model, compiler statically predicts the successor thread so that each fork instruction knows which thread needs to be spawned.

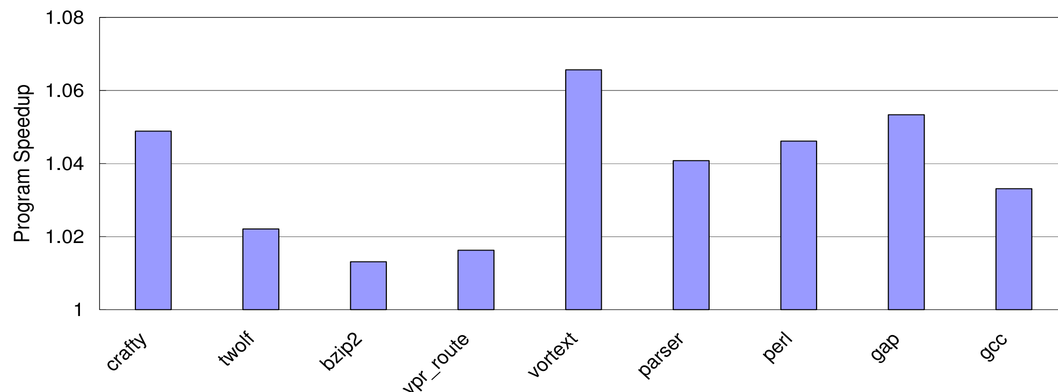


Figure 6.3: The program speedup of non-loop threads.

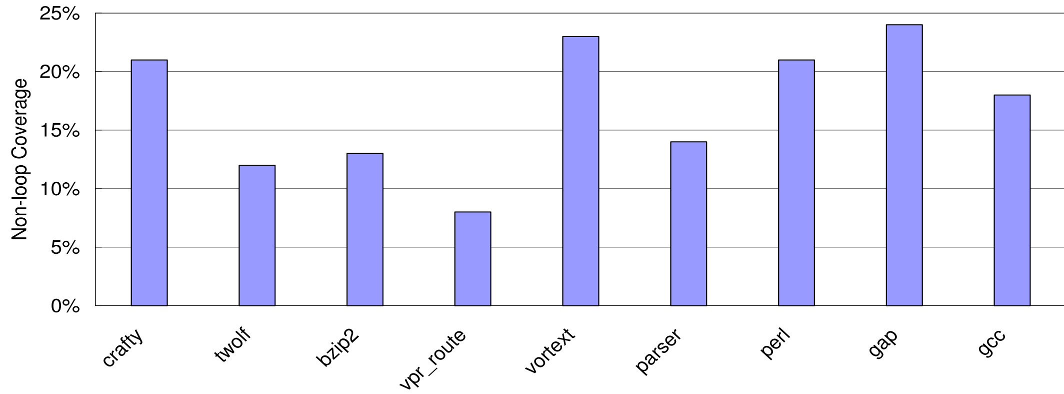


Figure 6.4: The coverage of non-loop threads.

The prediction is verified by the predecessor thread after it finishes execution. At this point, it knows exactly which thread needs to be executed next. If the successor thread is incorrectly predicted, it will be squashed along with all its successors. Since the overhead caused by a mis-prediction is so significant, it is important to know how well this static prediction scheme performs.

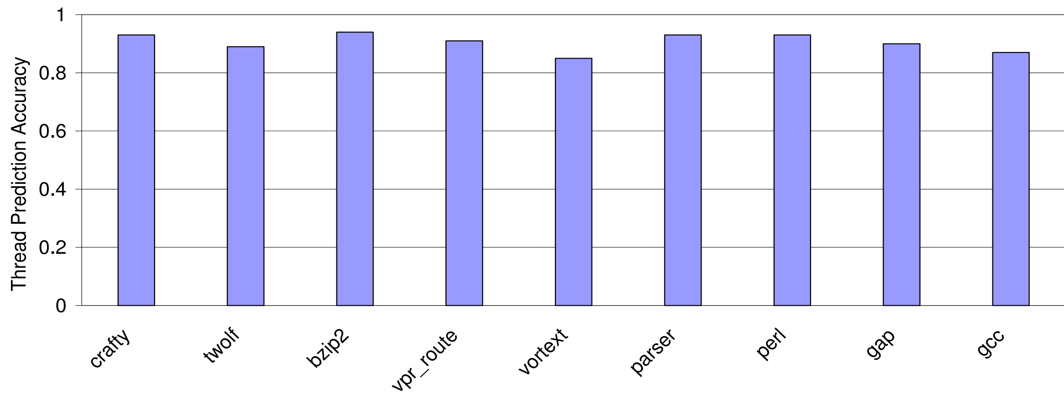


Figure 6.5: The prediction accuracy of non-loop thread.

Figure 6.5 shows the accuracy of static prediction. It is measured as the probability that a thread correctly predicts its successor thread. One means the successor thread is always correctly predicted, while zero means it is always incorrectly pre-

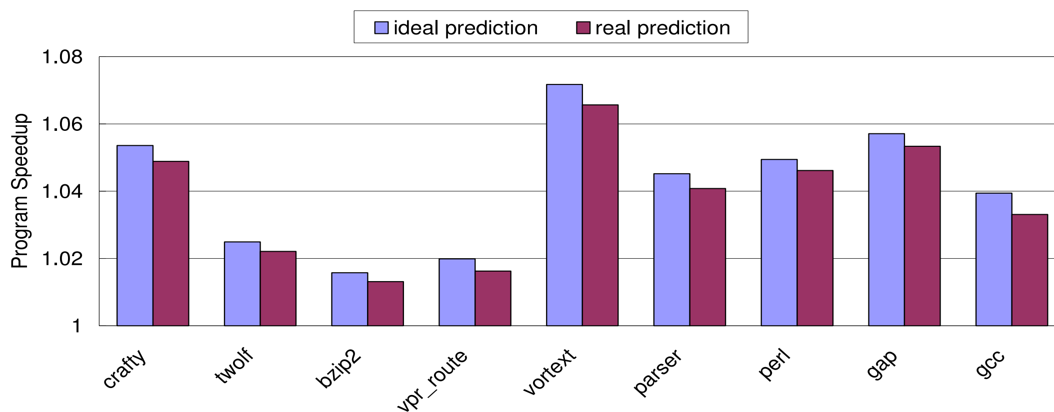


Figure 6.6: The performance impact of thread prediction.

dicted. For all the benchmarks, the static prediction achieves high accuracy with 90% on average. This is because our non-loop partitioning algorithm is based on the frequent execution path, which is highly predictable in most of cases.

The performance impact of static prediction is shown in Figure 6.6. Our static prediction is compared against an ideal scheme where each successor thread can always be correctly predicted. Our static prediction achieves the performance improvements close to this ideal scheme with the difference less than 1% for all the benchmarks. The high prediction accuracy minimizes the performance impact of mis-predictions.

6.5 Impact of Instruction Scheduling for Memory Dependence

In this section, the impact of instruction scheduling for memory dependence is evaluated. The instruction scheduling is applied on both loop threads and non-loop threads. The loop threads are created from the set of loops identified by estimation III described in Section 6.3, while the non-loop threads are the ones created by non-loop partitioning described in Section 6.4.

To gain more insights on the parallel execution, we break down the total execution time into 7 categories. *Busy* represents the amount of time spent in executing useful instructions. *Idle* represents the amount of time wasted due to the lack of parallel threads. *Syn* represents the amount of time spent in synchronizing frequently occurring memory dependences and all register dependences. Both the waiting time of the consumer thread and inter-thread communication delay are included in this category. *Cache* represents the amount of time spent on data cache and main memory. *Fail* represents the amount of time wasted due to the squashes of speculative threads. *Imbalance* indicates the amount of wasted time due to the imbalanced workloads. *Overflow* represents the amount of stall time due to the buffer overflow.

Figure 6.7 shows the performance impact of instruction scheduling on the loop threads. The execution time of the loop threads is normalized assuming the se-

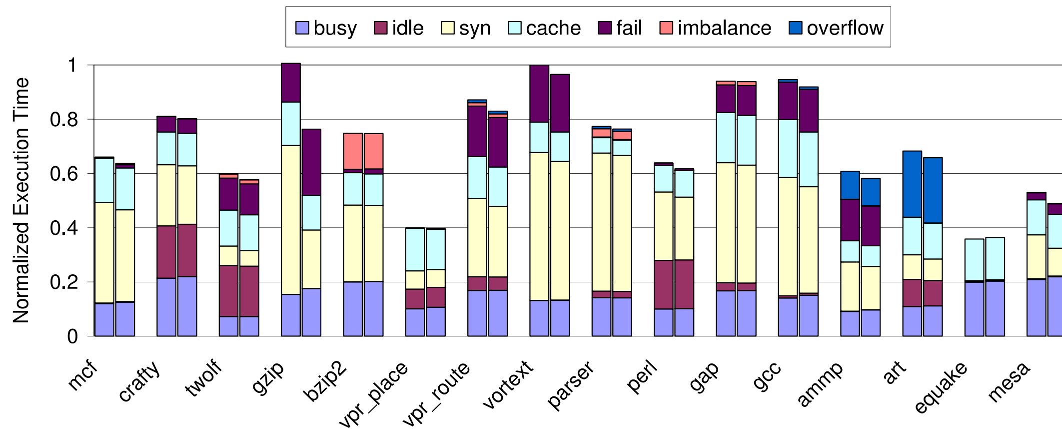


Figure 6.7: The performance impact of instruction scheduling for memory dependence on the loop threads.

quential execution time is 1. The first bar shows the normalized execution time of the loop threads before instruction scheduling for memory dependence is applied. The only thread optimization applied here is the instruction scheduling for register dependence. The second bar shows the normalized execution time of the loop threads after the instruction scheduling for memory dependence is applied.

Over half of benchmarks benefit from the instruction scheduling for memory dependence. The most significant one is *gzip*, whose speedup is improved from 0.99 to 1.31. Other benchmarks such as *mcf*, *twolf*, *vpr_route*, *vortex*, *perl*, *gcc*, *ammp*, *art*, and *mesa* also achieve 3% to 16% performance improvements. For these benchmarks, the improvements is mainly due to the reduced synchronization cost. Instruction scheduling also may have side effects, for instance, it increases the mis-speculation cost for *mcf*, *gzip*, *bzip2*, *gap*, *gcc* and *mesa*. There are two reasons: i) the aggressive instruction scheduling in the producer thread may fail and

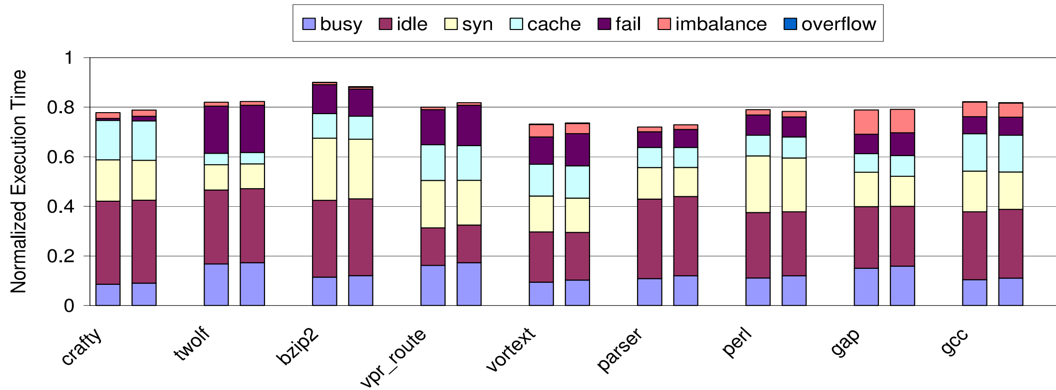


Figure 6.8: The performance impact of instruction scheduling for memory dependence on the non-loop threads.

cause the squash of the consumer thread and all following threads; and ii) the consumer thread is executed more aggressively since the data is available much earlier, and such aggressive execution may lead to more exposed loads that cause potential mis-speculations. For some benchmarks like *vpr_place* and *equake*, instruction scheduling has little impact on the performance, since memory dependence is no longer the bottleneck for these benchmarks.

Figure 6.8 shows the performance impact of instruction scheduling on the non-loop threads. The first bar shows the normalized execution time before instruction scheduling for memory dependence is applied, while the second bar shows the normalized execution time after the instruction scheduling for memory dependence is applied.

Compared to the loop threads, the non-loop threads benefit little from instruction scheduling for memory dependence. For several benchmarks like *crafty*, *vpr_route*, *vortex*, and *parser*, the situation is even worse, and they are actually

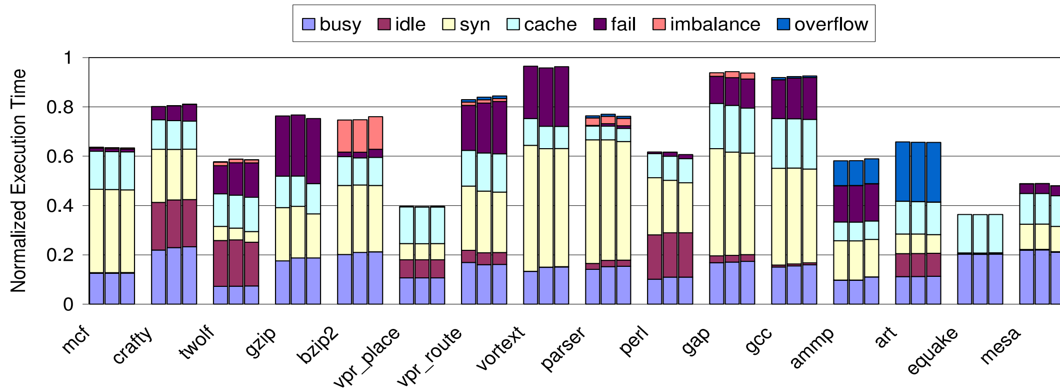


Figure 6.9: The performance impact of different instruction scheduling strategies on the loop threads.

slowed down due to the increased mis-speculation cost. For other benchmarks, only very little improvements have been achieved. As a result, instruction scheduling is not recommended for the non-loop threads.

During aggressive instruction scheduling, one of the main concern is that how aggressively the instructions should be scheduled. Data dependence probability is a commonly used hint for such decision. For instance, if the probability of a memory dependence is higher than some threshold, we will stop scheduling an load instruction across the store it depends on. However, it is often difficult to select such a threshold value to maximize the overall benefit of instruction scheduling. In order to gain insights on how aggressive instruction scheduling should be, we try different threshold values during scheduling. The higher the threshold is, the more aggressive the instruction scheduling is. Since the loop threads are more likely to benefit from instruction scheduling as it is shown previously, we use the loop threads for this study.

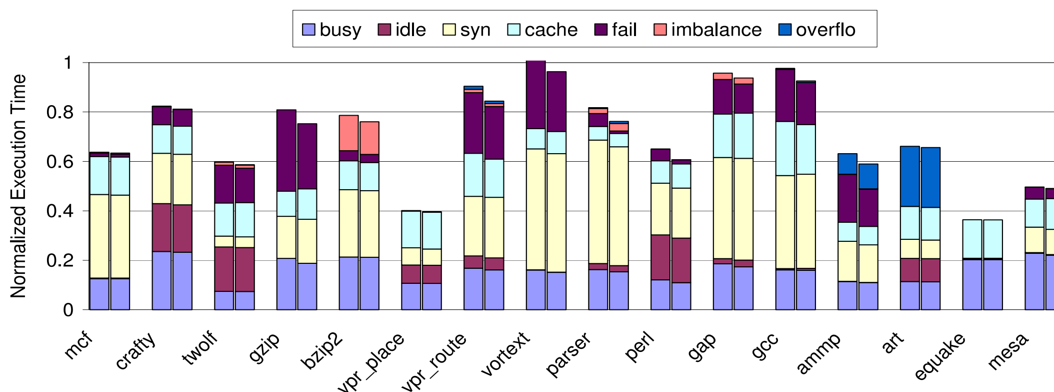


Figure 6.10: The performance impact of recovery code on the loop threads optimized by using the most aggressive instruction scheduling strategy.

Figure 6.9 shows the performance impact of different instruction scheduling strategies. The first bar shows the performance of the the most conservative scheduling for which we set the threshold value to 20%, that is, a load will not be scheduled across a store if the dependence probability is higher than 20%. The second bar shows the performance of a more aggressive scheduling with the threshold value of 50%. The third bar shows the performance of the most aggressive scheduling with the threshold value of 80%. Overall, these three scheduling strategies achieve similar performance improvements no matter how aggressive they are. This strongly indicates that the data dependence is usually biased. Most of dependences occur either very infrequently with a probability lower than 20% or very frequently with a probability higher than 80%. Such property of data dependence is also observed by Chen et al. [11].

From Figure 6.9, we can also see that the mis-speculation cost tends to increase as the scheduling becomes more aggressive. To understand how important the

recovery code is in reducing the mis-speculation cost, we select the loop threads optimized by using the most aggressive scheduling strategy to study the impact of recovery code. Figure 6.10 shows the experimental results. The first bar shows the performance of the loop threads without recovery code support, while the second bar shows the performance of the loop threads with recovery code support. Several benchmarks such as *gzip*, *vpr_route*, *vortex parser*, *perl*, *gcc* and *ammp* have significant performance degradation when the recovery code support is not available. The performance loss is mainly due to the increased mis-speculation cost. With recovery code support, such mis-speculation cost can be greatly reduced so that instruction scheduling can be performed more aggressively.

6.6 Impact of Reduction Transformation

In this section, the performance impact of reduction transformation is evaluated. Reduction transformation is only applied to the loop threads. Table 6.8 shows for each benchmark the number of loops with reduction variables transformed.

Figure 6.11 shows the performance impact of reduction transformation on the loop threads. As the same as Figure 6.7, The first two bars shows the normalized execution time before and after instruction scheduling for memory dependence is applied, while the third bar shows the normalized execution time after both instruction scheduling and reduction transformation are applied. Three benchmarks benefit significantly from this transformation: the speedup for *twolf* is improved

Table 6.8: Loops for reduction transformation.

	Application Name	Number of Loops Transformed
CINT2000	mcf	3
	crafty	2
	twolf	6
	gzip	4
	bzip2	5
	vpr-place	1
	vpr-route	3
	vortex	4
	parser	10
	perl	3
	gap	3
	gcc	19
CFP2000	ammp	1
	art	0
	equake	0
	mesa	2

from 1.73 to 1.85; the speedup for *bzip2* is improved from 1.33 to 1.49; and the speedup for *mesa* is improved from 2.04 to 2.59. Other two benchmarks *parser* and *gcc* also benefit from this transformation although the improvements are not as significant as the former three benchmarks. The speedup for *parser* is improved from 1.30 to 1.34, while the speedup for *gcc* is improved from 1.08 to 1.13. The improvements are mainly due to the reduced cost of synchronizing reduction variables. For both *bzip2* and *mesa*, such synchronization costs are greatly reduced after reduction transformation.

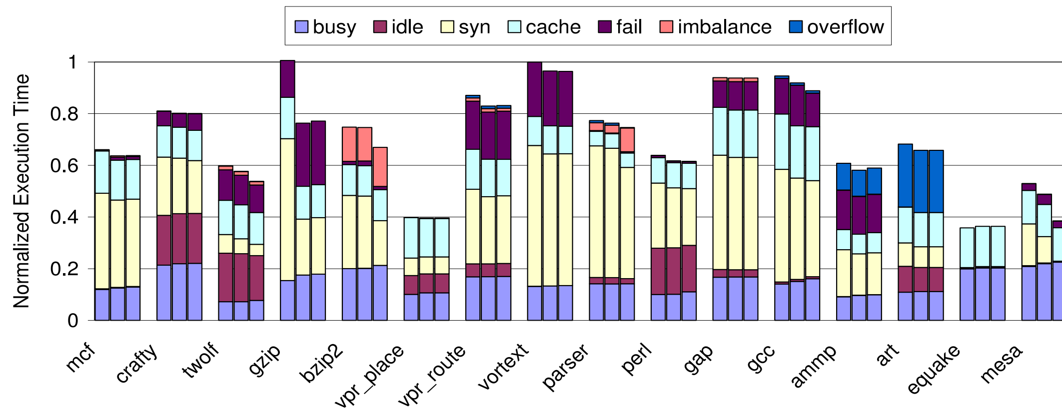


Figure 6.11: The performance impact of reduction transformation on the loop threads.

6.7 Impact of Iteration Merging

In this section, the impact of iteration merging is evaluated. It is the last optimization that is applied on the loop threads. Its performance impact is shown in Figure 6.12. The first three bars are the same as the ones shown in Figure 6.11, while the fourth bar shows the performance improvements after iteration merging is applied on top of previous two optimizations. Iteration merging is most effective for two benchmarks *bzip2* and *parser*. Both of them suffer the performance loss due to imbalanced workloads. Note that, for *parser*, the load imbalancing becomes a more severe problem after the reduction transformation is applied. The speedup for *bzip2* is improved from 1.49 to 1.69, while the speedup for *parser* is improved from 1.33 to 1.39. Some loop threads in other two benchmarks *twolf* and *gap* also benefit from this optimization, but since the coverages of those loops are low, no significant improvements are achieved.

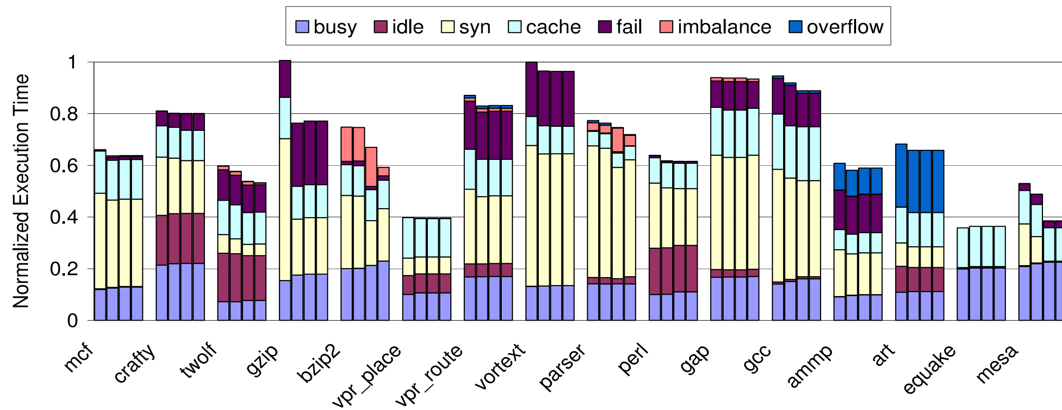


Figure 6.12: The performance impact of iteration merging on the loop threads.

6.8 Program Speedup

Figure 6.13 shows the overall program speedup achieved from both loop threads and non-loop threads after all optimizations described in the previous sections are applied.

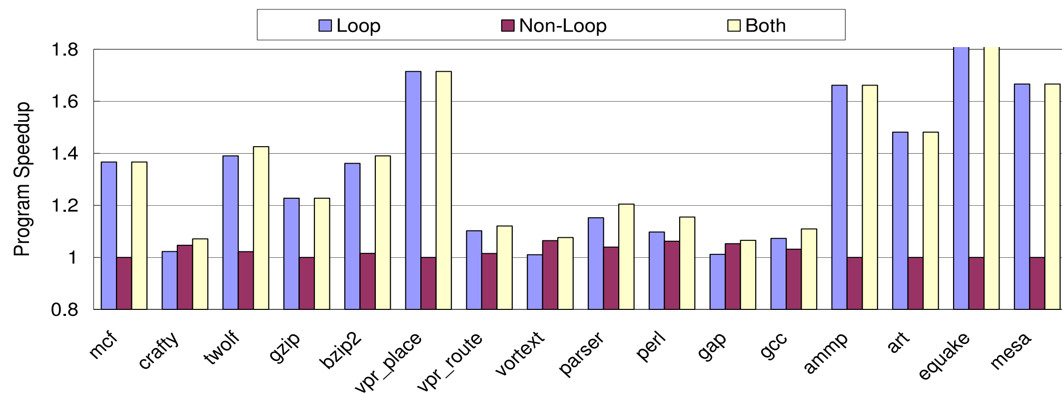


Figure 6.13: The program speedup achieved from both loop and non-loop threads.

On average, the loop threads achieve the speedup of 1.36, while the non-loop threads achieve the speedup of 1.02. Both of them achieve the speedup of 1.38. Loop threads are effective for all floating point benchmarks with the average

speedup of 1.81. Loop threads are also effective for several integer benchmarks such as *mcf*, *twolf*, *gzip*, *bzip2* and *vpr_route*, and all of them achieve the speedup higher than 1.2. For the benchmarks that have limited loop-level parallelism such as *crafty*, *vortex* and *gap*, non-loop threads can further deliver the speedup around 1.05.

Chapter 7

Conclusions

Microprocessors that support multiple threads of execution are becoming increasingly common. TLS allows a sequential program to be speculatively parallelized to utilize the underlying hardware resources. Under TLS, The potentially dependent threads can be executed in parallel while the sequential semantics of the program is maintained through runtime checking. Although TLS greatly simplifies the automatic parallelization process, the compiler is still crucial to deliver the desirable performance for those hard-to-parallelize applications.

In this thesis, we have studied several compiler techniques to explore the potential of TLS. One important task of a TLS compiler is to identify speculative threads with good performance potential. Loops are the primary candidates for extracting parallel threads due to their regular structures and significant coverage. We have proposed a loop selection algorithm that decides which loops should be

parallelized for a program with a large number of nested loops. Based on the accurate performance prediction of parallel execution, this loop selection algorithm is effective in selecting a set of loops to maximize the overall program performance. We have also observed that some general-purpose applications have limited loop-level parallelism. For such applications, we have proposed a partitioning algorithm to extract non-loop threads as the complementary to the loop threads.

The other important task of a TLS compiler is to optimize the performance of extracted threads. We have proposed three optimization techniques to improve the thread performance: instruction scheduling, reduction transformation and iteration merging. Instruction scheduling improves the efficiency of synchronizing frequently occurring memory dependences; reduction transformation reduces the impact of a class of reduction variables whose intermediate results are used by non-reduction operations; iteration merging improves the load balancing by dynamically combining multiple short loop iterations with a long iteration to achieve more balanced workloads.

All the proposed techniques are implemented in a TLS compiler framework built on the Intel's ORC compiler. On average, for 15 SPEC2000 benchmarks, we achieve 1.36 program speedup from loop threads. For some applications that have limited loop-level parallelism, we also achieve 1.05 program speedup from non-loop threads. Overall, we achieve 1.38 program speedup from both loop threads and non-loop threads. Our experimental results show that the compiler is crucial

in delivering the desirable performance for general-purpose applications, and our proposed techniques are effective in exploiting speculative parallelism under TLS.

The research work presented in this dissertation can be extended as follows. First, exploiting TLP at single loop level may not be sufficient to fully utilize the increasing computing resources on a modern multithreaded procssor, thus it is desirable to extend our loop selection algorithm to identify multiple levels of parallel loops and map them to the hardware hierarchically. Second, it is not clear whether the traditional compiler techniques, such as procedure inlining and loop unrolling, will deliver the desired performance when coupled with TLS. Furthmore, we need to investigate the application of such optimizations to enhance the performance of TLS. Our compiler framework provides a convenient way to study these problems. Finally, our proposed techniques can be potentially applied to optimize programs for transactional memory. However, since the sequential ordering of threads is not mandatory in transactional memory, our performance estimation techniques need to be reconsidered.

Bibliography

[1] Aho, Sethi, and Ullman. *Compilers, Principles, Techniques and Tools*. Addison, Wesley, 1986.

[2] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. In *31st Annual IEEE/ACM International Symposium on Microarchitecture (Micro-31)*, December 1998.

[3] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software Pipelining. *ACM Computing Surveys*, 27(3):367–432, Sept 1995.

[4] AMD Corporation. Leading the Industry: Multi-core Technology & Dual-Core Processors from AMD. http://multicore.amd.com/en/Technology/, 2005.

[5] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, February 2005.

[6] A. Bhowmik and M. Franklin. A General Compiler Framework for Speculative Multithreading. In *the 14th ACM Symposium on Parallel Algorithms and Architectures*, August 2002.

[7] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with polaris. *IEEE Computer*, 29(12):78–82, 1996.

[8] M. Burrow and D.J. Wheeler. A Block-sorting Lossless Data Compression Algorithm. Technical Report 124, Digital Systems Research Center, May 1994.

[9] D. K. Chen and P. C. Yew. Statement Re-ordering for DOACROSS Loops. In *International Conference on Parallel Processing*, pages 24–28, August 1994.

[10] Mike Chen and Kunle Olukotun. TEST: A Tracer for Extracting Speculative Threads. In *The 2003 International Symposium on Code Generation and Optimization*, March 2003.

[11] T. Chen, J. Lin, X.Dai, W.C. Hsu, and P.C. Yew. Data Dependence Profiling for Speculative Optimizations. In *Int'l Conf on Compiler Construction (CC)*, pages 57–62, March 2004.

[12] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *27th*

Annual International Symposium on Computer Architecture (ISCA '00), June 2000.

[13] M. Cintra and J. Torrellas. Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors. In *8th International Symposium on High-Performance Computer Architecture (HPCA-8)*, 2002.

[14] L. Codrescu, D. S. Wills, and J. Meindl. Architecture of the Atlas Chip-Multiprocessor: Dynamically Parallelizing Irregular Applications. *IEEE Transactions on Computers*, 50(1), Jan 2001.

[15] Ronald G. Cytron. *Compile-Time Scheduling and Optimization for Asynchronous Machines*. PhD thesis, University of Illinois at Urbana-Champaign, 1987.

[16] X. Dai, A. Zhai, W.C. Hsu, and P.C. Yew. A General Compiler Framework for Speculative Optimizations Using Data Speculative Code Motion. In *The 2005 International Symposium on Code Generation and Optimization*, March 2005.

[17] D.Z. Du and P.M. Pardalos. *Handbook of Combinatorial Optimization*. Kluwer Academic Publishers, 1999.

[18] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai. A Cost-Driven Compilation Framework for Speculative Parallelization of Sequential

Programs. In *ACM SIGPLAN 04 Conference on Programming Language Design and Implementation (PLDI'04)*, June 2004.

[19] Pradeep Dubey, Kelvin O'Brien, Kathryn O'Brien, and Charles Barton. Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-assisted Fine-Grained Multithreading. In *International Conference on Parallel Architectures and Compilation Techniques (PACT 1995)*, June 1995.

[20] Open Research Compiler for Itanium Processors, Jan 2003.

[21] M. Franklin and G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, 45(5), May 1996.

[22] Manoj Franklin and Gurindar S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelsim. In *19th Annual International Symposium on Computer Architecture (ISCA '92)*, pages 58–67, May 1992.

[23] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative Versioning Cache. In *4th International Symposium on High-Performance Computer Architecture (HPCA-4)*, February 1998.

[24] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, and T. Yamazaki Y. Watanabe. A novel SIMD architecture for the Cell heterogeneous chip-multiprocessor. In *Hot Chips 17*, August 2005.

[25] M. Gupta and R. Nim. Techniques for Speculative Run-Time Parallelization of Loops. In *Supercomputing '98*, November 1998.

[26] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of ASPLOS-VIII*, October 1998.

[27] 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 *31st Annual International Symposium on Computer Architecture (ISCA '04)*. Jun 2004.

[28] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20th Annual International Symposium on Computer Architecture (ISCA '93)*, pages 289–300, May 1993.

[29] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Preliminary Experiences with the Fortran D Compiler. In *Supercomputing '93*, 1993.

[30] IBM. IBM delivers Power-based chip for Microsoft Xbox 360 worldwide launch, 2005.

[31] Intel Corporation. Intel's Dual-Core Processor for Desktop PCs. http://www.intel.com/personal/desktopcomputer/dual_core/index.htm, 2005.

[32] Intel Corporation. Intel Itanium Architecture Software Developer's Manual, Revision 2.2. http://www.intel.com/design/itanium/manuals/iiasdmanual.htm, 2006.

[33] T.A. Johnson, R. Eigenmann, and T.N. Vijaykumar. Min-Cut Program Decomposition for Thread-Level Speculation. In *ACM SIGPLAN 04 Conference on Programming Language Design and Implementation (PLDI'04)*, June 2004.

[34] Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *Microprocessor Forum '99*, October 1999.

[35] Ken Kennedy and Randy Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Academic Press, 2002.

[36] T. Knight. An Architecture for Mostly Functional Languages. In *Proceedings of the ACM Lisp and Functional Programming Conference*, pages 500–519, August 1986.

[37] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2), March 2005.

[38] V. Krishnan and J. Torrellas. A Chip Multiprocessor Architecture with Speculative Multithreading. *IEEE Transactions on Computers, Special Issue on Multithreaded Architecture*, September 1999.

[39] V. Krishnan and J. Torrellas. The Need for Fast Communication in Hardware-Based Speculative Chip Multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT 1999)*, October 1999.

[40] M.S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *ACM SIGPLAN 88 Conference on Programming Language Design and Implementation (PLDI'88)*, June 1988.

[41] J. Lin, T. Chen, W.C. Hsu, P.C. Yew, R. Ju, and T.F. Ngai. A Compiler Framework for Speculative Analysis and Optimizations. In *ACM SIGPLAN 03 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 289–299, June 2003.

[42] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: A TLS Compiler that Exploits Program Structure. In *ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming*, March 2006.

[43] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM SIGPLAN 05 Conference on Programming Language Design and Implementation (PLDI'05)*, June 2005.

[44] P. Marcuello and A. Gonzalez. Clustered Speculative Multithreaded Processors. In *13th Annual ACM International Conference on Supercomputing*,

Rhodes, Greece, June 1999.

[45] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based Transactional Memory. In *12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, February 2006.

[46] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[47] J. Oplinger, D. Heine, and M. Lam. In Search of Speculative Thread-Level Parallelism. In *Proceedings PACT 99*, October 1999.

[48] V. Packirisamy, S. Wang, A. Zhai, W.-C. Hsu, and P.-C. Yew. Supporting Speculative Multithreading on Simultaneous Multithreaded Processors. In *13th IEEE International Conference on High Performance Computing*, December 2006.

[49] I. Park, B. Falsafi, and T. N. Vijaykumar. Implicitly-Multithreaded Processors. In *30th Annual International Symposium on Computer Architecture (ISCA '03)*, June 2003.

[50] Manohar Prabhu and Kunle Olukotun. Using Thread-Level Speculation to Simplify Manual Parallelization. In *ACM SIGPLAN 2003 Symposium on Principles and Practice of Parallel Programming*, 2003.

[51] Manohar Prabhu and Kunle Olukotun. Exposing Speculative Thread Parallelism in SPEC2000. In *ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming*, 2005.

[52] C. G. Quinones, C. Madriles, J. Sanchez, P. Marcuello, A. Gonzales, and D. M. Tullsen. Mitosis Compiler: An Infrastructure for Speculative Threading Based on Pre-Computation Slices. In *ACM SIGPLAN 05 Conference on Programming Language Design and Implementation (PLDI'05)*, June 2005.

[53] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pages 5–17. Oct 2002.

[54] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing Transactional Memory. In *32nd Annual International Symposium on Computer Architecture (ISCA '05)*, June 2005.

[55] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas. Tasking with Out-of-Order Spawn in TLS Chip Multiprocessors: Microarchitecture and Compilation. In *19th Annual ACM International Conference on Supercomputing*, June 2005.

[56] V. Sarkar and J. Hennessy. Partitioning Parallel Programs for Macro-Dataflow. In *ACM Conference on LISP and Functional Programming*, 1986.

[57] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pages 414–425, June 1995.

[58] J. G. Steffan, C. B. Colohan, and T. C. Mowry. Architectural Support for Thread-Level Data Speculation. Technical Report CMU-CS-97-188, School of Computer Science, Carnegie Mellon University, November 1997.

[59] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *27th Annual International Symposium on Computer Architecture (ISCA '00)*, June 2000.

[60] J. G. Steffan and T. C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *4th International Symposium on High-Performance Computer Architecture (HPCA-4)*, February 1998.

[61] S. Tjiang, M. Wolf, M. Lam, K. Pieper, and J. Hennessy. *Languages and Compilers for Parallel Computing*, pages 137–151. Springer-Verlag, Berlin, Germany, 1992.

[62] J.-Y. Tsai, J. Huang, C. Amlo, D.J. Lilja, and P.-C. Yew. The Superthreaded Processor Architecture. *IEEE Transactions on Computers, Special Issue on Multithreaded Architectures*, 48(9), September 1999.

[63] J. Tubella and A. Gonzalez. Control Speculation in Multithreaded Processors through Dynamic Loop Detection. In *4th International Symposium on High-Performance Computer Architecture (HPCA-4)*, February 1998.

[64] T. N. Vijaykumar and Gurindar S. Sohi. Task Selection for a Multiscalar Processor. In *31st Annual IEEE/ACM International Symposium on Microarchitecture (Micro-31)*, November 1998.

[65] T.N. Vijaykumar. Compiling for the Multiscalar Architecture. In *Ph.D. Thesis*, January 1998.

[66] S. Wang, K. S. Yellajyosula, A. Zhai, and P.-C. Yew. Loop Selection for Thread-Level Speculation. In *The 18th International Workshop on Languages and Compilers for Parallel Computing*, Oct 2005.

[67] S. Wang, A. Zhai, and P.-C. Yew. Exploiting Speculative Thread-Level Parallelism in Data Compression Applications. In *The 19th International Workshop on Languages and Compilers for Parallel Computing*, Oct 2006.

[68] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison Wesley, 1996.

[69] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler Optimization of Scalar Value Communication Between Speculative Threads. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, Oct 2002.

[70] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler Optimization of Memory-Resident Value Communication Between Speculative Threads. In *The 2004 International Symposium on Code Generation and Optimization*, Mar 2004.

[71] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transaction on Information Theory*, 23(3):337–343, 1977.