Compiler Optimization of Value Communication for Thread-Level Speculation

Antonia Zhai

January 13, 2005

CMU-CS-05-103

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Todd C. Mowry, Chair
Seth C. Goldstein
Peter Lee
Wen-mei W. Hwu, UIUC

This research was sponsored by the National Aeronautics and Space Administration (NASA) under grant nos. NAG2-6054 and NAG2-1230, the National Science Foundation (NSF) under grant no. CCR-0219931, and the Intel Corporation.

Keywords: Thread-Level Speculation, Architecture, Compiler Optimization, Automatic Parallelization, Dataflow Analysis, Dependence Profiling

Abstract

In the context of Thread-Level Speculation (TLS), inter-thread value communication is the key to efficient parallel execution. From the compiler's perspective, TLS supports two forms of inter-thread value communication: speculation and synchronization. Speculation allows for maximum parallel overlap when it succeeds, but becomes costly when it fails. Synchronization, on the other hand, introduces a fixed cost regardless of whether the dependence actually occurs or not. The fixed cost of synchronization is determined by the critical forwarding path, which is the time between when a thread first receives a value from its predecessor to when a new value is generated and forwarded to its successor. In the baseline implementation used in this dissertation, we synchronize all register-resident values and speculate on all memory-resident values. However, this naive approach yields little performance gain due to the excessive cost from inter-thread value communication. The goal of this dissertation is to develop compiler-based techniques to reduce the cost of inter-thread value communication and improve the overall program performance.

This dissertation proposes to use the compiler to orchestrate inter-thread value communication for both memory-resident and register-resident values. To improve the efficiency of inter-thread value communication, the compiler must first decide whether to synchronize or to speculate on a potential data dependence based on how frequently the dependence occurs. If synchronization is necessary, the compiler will then insert the corresponding *signal* and *wait* instructions, creating a point-to-point path to forward the values involved in the dependence. Because synchronization could serialize execution by stalling the consumer thread, we use the compiler to avoid such stalling by applying novel dataflow analyses to schedule instructions to shrink the critical forwarding path.

This dissertation reports the performance impact of several compiler-base value communication optimization techniques on a four-processor single-chip multiprocessor that has been extended to support thread-level speculation. Relative to the performance of the original sequential program executing on a single processor, for the set of loops selected to maximize program performance, parallel execution with the proposed baseline implementation results in 1% performance degradation for integer benchmarks and 21% performance improvement for floating point benchmarks, while with the optimization techniques we developed, parallel execution achieves 22% and 42% performance improvement for integer benchmarks and floating point benchmarks, respectively.

Acknowledgements

After generating several gigabytes of compilation intermediate files and parsing through hundreds of simulation outputs, I am amazed that this document has finally come together. Now, looking back to all my years at CMU, one thing is for sure: I could not have done this alone and I owe my thanks to many people.

First and foremost, I am deeply indebted to my advisor, Todd Mowry, for providing continuous guidance and support for the part ten (!) years (I have been working with Todd since I was a wee undergraduate). Words fail to describe the extend to which he has so patiently shaped my technical thinking, research abilities, presentation skills, among other things. Thank you, Todd!

I would also like to thank the other members of my thesis committee—Seth Goldstein, Peter Lee, and Wen-Mei Hwu—for providing invaluable assistance at all stages of this thesis. Their criticisms and advices are vital in making this thesis more accurate, more complete, and easier to read.

Furthermore, I also owe many thanks to my fellow STAMPeders for providing inspirations, criticisms, etc., during the meetings, and for sitting through so many practice talks! My special thanks go towards Chris Colohan and Greg Steffan, with whom I have worked closely, for building infrastructures and tools that make my research possible.

Last, but not least, I would also like to thank my family and friends for their love and support without which I would not have survived the Ph.D. process.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As technology advances, microprocessors that support multiple threads of execution are becoming increasingly common [18–20, 22, 35, 38, 66]. One way to fully utilize the computational power of such processors to speed up a single application is to create parallel programs by finding independent threads [5, 33, 65]. However, automatic parallelization for many general-purpose applications (e.g., compilers, spreadsheets, games, etc.) is very difficult due to pointer and indirect references, complex data structures and control flow, and input-dependent program behaviors. Thread-Level Speculation (TLS) [1, 14, 24, 29, 31, 32, 36, 40, 43, 51, 53, 54, 63, 67] facilitates the parallelization of such applications by allowing potentially dependent threads to execute in parallel while maintaining the original sequential semantics of the programs. To ensure correct execution under TLS, all true (read-after-write) inter-thread data dependences must be satisfied through some form of *inter-thread value communication*. However, inter-thread value communication, if not properly managed, may become the performance bottleneck by serializing parallel execution. This thesis reports the design, implementation and evaluation of several compiler techniques that mitigate such effects on performance.

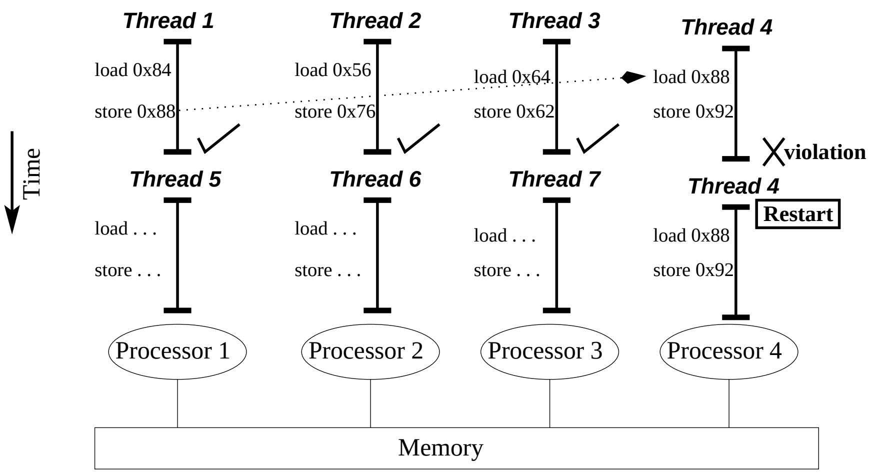
1.1 TLS and Inter-Thread Communication

In TLS, the compiler partitions a program into parallel speculative threads without having to prove that they are independent, while at runtime the underlying hardware checks whether inter-thread data dependences are preserved and re-executes any thread for which they are not. This TLS execution model allows the parallelization of programs that were previously non-parallelizable. The following example demonstrates the basic principles of TLS. With TLS, the loop in Figure 1.1(a) can be parallelized by the compiler without proving that pointer p does not point to the same memory location as pointer q from previous iterations for all executions of the loop. Figure 1.1(b) shows how the program is executed speculatively in parallel on a four-processor, shared-memory multiprocessor that supports TLS, where each thread of execution corresponds to a single iteration of the loop. A thread is allowed to execute until completion if the the load through pointer p in this thread does not load from the addresses stored to by pointer q in the previous threads. However, in the example, the load through pointer p in *thread 4* loads from the address stored to by *thread 1*, creating a read-after-write data dependence and causing the speculation to fail. Consequently, while *threads 1, 2* and *3* have been successfully parallelized, *thread 4* has to be re-executed.

TLS is very efficient when speculation always succeeds since it would allow maximum parallel overlap as shown in Figure 1.2(a), while it becomes very inefficient when speculation fails often since re-execution has to be invoked frequently, as shown in Figure 1.2(b). Thus, for frequently occurring data dependences, we must have some alternative methods to avoid speculation failures. One approach is to synchronize them by inserting a `wait` operation before `load *p` and a `signal` operation after `store *q` to forward the stored value explicitly between the two threads, as shown in Figure 1.2(c). On the other hand, synchronization has its own problem—it stalls the

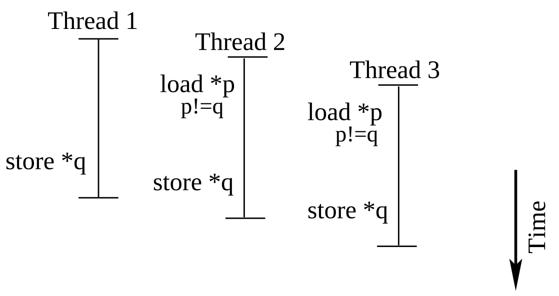
```
do {
  ...
  load *p;
  ...
  store *q;
  ...
} while (condition)
```

(a) Pointer-based code example.

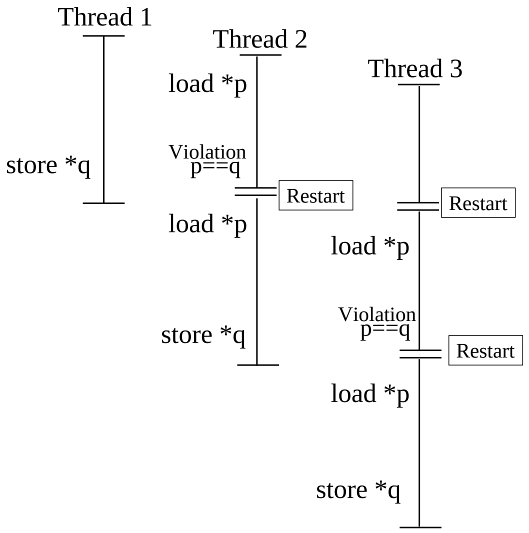


(b) Speculatively parallel threads.

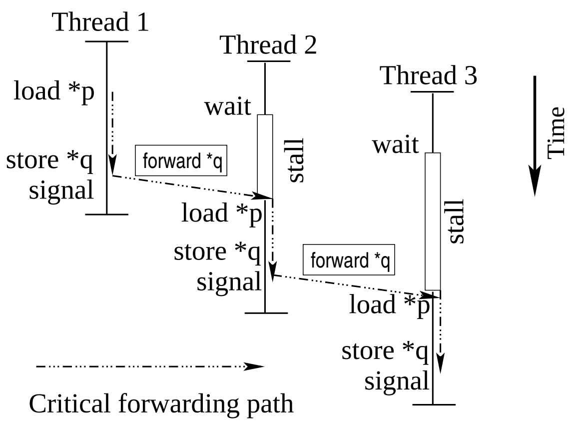
Figure 1.1: Loops with potential inter-thread data dependences can be parallelized under TLS.



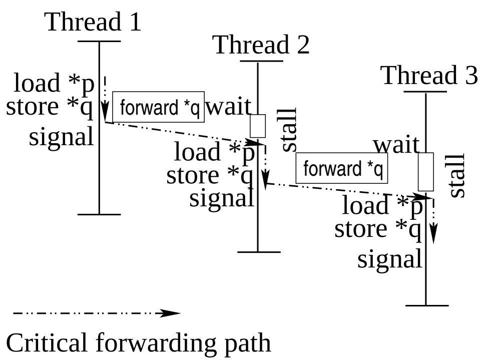
(a) Speculating on a data dependence that always succeeds.



(b) Speculating on a data dependence that always fails.



(c) Synchronizing a data dependence with a long critical forwarding path.



(d) Synchronizing a data dependence with a shortened critical forwarding path.

Figure 1.2: Synchronization vs. speculation under TLS.

consumer threads. The amount of time the consumer threads sit idle during parallel execution is determined by the *critical forwarding path*, which is the time between when a thread first receives a value from its predecessor to when a new value is generated and forwarded to its successor. The shorter the *critical forwarding path*, the more parallel overlap the parallel threads have, as shown in Figure 1.2(c) and 1.2(d).

To summarize, from the perspective of the compiler, there are two ways to communicate values between threads under TLS. The compiler can either (i) pretend that the dependence does not exist, and completely rely on the underlying hardware to detect dependence violations, and invoke recovery operations when necessary; or (ii) schedule an explicit synchronization using the wait/signal instruction pair to forward a value between the threads. We refer to the former as value communication via speculation and the latter as value communication via synchronization.

1.2 Estimating the Performance Impact of Optimizing Inter-Thread Value Communication

To quantify the importance of optimizing inter-thread value communication, we estimate the performance potential of compiler optimization for inter-thread value communication by simulating TLS execution using an optimal inter-thread value communication model. For this experiment, a set of loops is selected to maximize program performance when the cost of inter-thread value communication could be completely eliminated. These loops are parallelized by synchronizing all dependences between register-resident values and speculating on all dependences between memory-resident values. This inter-thread value communication scheme is chosen as the default implementation because the dependences between register-resident values are easy to analyze statically and are often dependent, while memory-resident values are difficult

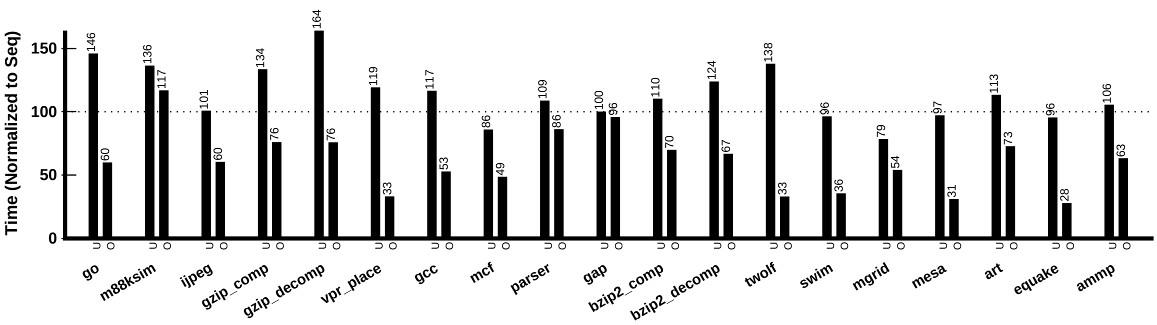


Figure 1.3: Program performance potential of optimizing inter-thread value communication. **U** is unoptimized for inter-thread value communication, all register-resident values are synchronized and all memory-resident values are speculated; **O** assumes a perfect value predictor for all values.

to analyze statically and less likely to be dependent.

This experiment is conducted using a detailed simulator that implements a four-processor single-chip multiprocessor with TLS support. Detailed description of the underlying hardware can be found in Chapter 2. Two sets of simulations are studied, the first simulating the costs of inter-thread value communication, including the costs of both re-executions and synchronization, and the second set estimating the performance upper bound by implementing an oracle that always applies the optimal strategy of inter-thread value communication. The best possible optimization for reducing the cost of value communication is a perfect value predictor that prevents any data dependence speculation from failing and any synchronization from stalling. The results are shown in Figure 1.3, in which each bar represents the program execution time normalized to that of the original sequential program. Bars less than 100 are speedups. The **U** bars, which are the execution time when no value communication optimization is applied, indicate that our baseline spends a significant amount of time on inter-thread value communication, slowing down integer benchmarks by 21.8% while speeding up floating point benchmarks by a mere 1% on average relative to sequential execution. With a perfect value predictor eliminating the cost of inter-thread value communication, integer benchmarks and floating point benchmarks can potentially speed up performance by 33.6% and 52.5% respectively relative to the sequential program execution. The performance gap between the **U** bars and the **O** bars is pursued in this dissertation.

This dissertation proposes to use the compiler to orchestrate inter-thread value communication for both memory-resident and register-resident values. The compiler first decides whether to synchronize or to speculate on a potential data dependence. If synchronization is necessary, the compiler will then insert the corresponding *signal* and *wait* instructions, creating a point-to-point path to forward the values involved in the dependence. The compiler is also used to avoid stalling the consumer threads by scheduling instructions to reduce the critical forwarding path.

1.3 Related Work

Many schemes have been proposed for improving the efficiency of inter-thread value communication, some hardware-based [15, 39, 43, 47, 64] and others software-based [32, 67, 68, 72, 73, 75]. This section explains value communication issues raised from both hardware-based and software-based previous investigations. More detailed descriptions of the related work for each compiler optimization technique can be found in the corresponding chapters.

1.3.1 Hardware-Based Schemes for Improving Inter-Thread Value Communication

For inter-thread value communication, hardware-based approaches usually treat register-resident values and memory-resident values with different strategies. Hardware-based automatic synchronization [15, 47, 64] was proposed specifically for inter-thread memory-resident value communication. Dedicated hardware register-resident value forwarding [39] and reorder buffer prioritization [64] were proposed specifically for improving register-resident values communication. We will also discuss hardware-based value prediction techniques [15, 44, 64] that target both memory-resident and register-resident values.

Hardware Support for Automatic Synchronization for Memory-Resident Values

To detect inter-thread data dependences for memory resident values, Multiscalar [25, 26, 58] proposed using a centralized address resolution buffer (ARB). Based on such a centralized scheme, Moshovos *et al.* [47] used a centralized hardware lookup table to automatically synchronize frequently occurring data dependences by matching dependent load/store pair and demonstrated that the number of speculation failures can be reduced. This approach, however, requires a special functionality that is unique to the Multiscalar proposal—that is, the memory location causing data dependence violation is made available to the consumer immediately after the producer has finished modifying this location. For TLS proposals that are based on a distributed memory coherence protocol, implementing this feature implies performing complex version management. In addition, this scheme could limit performance and is difficult to scale due to the need of centralized structures.

To avoid the centralized structures required in Moshovos' work, Cintra *et al.* [15] and our group [64] both proposed approaches using only decentralized hardware lookup tables for automatically synchronizing memory-resident values. Cintra *et al.* [15] implemented hardware lookup tables to determine whether speculation is likely to fail for a certain cache line, while our proposal [64] used a fully associative buffer to keep track of load instructions rather than memory lines. Although it is relatively easy to identify the cache lines and/or the load instructions that cause speculation to fail, it is more difficult to identify the corresponding store instructions that produce the value. For the producer to identify such store instructions, it must be able to predict (i) the memory locations causing data dependence violations in the consumer thread and (ii) the last stores modifying these memory locations. Thus, both proposals could over-synchronize parallel execution by requiring the consumer thread to stall until the previous threads commit.

The compiler, having a global view of the entire program, could help in this case, and thus this research uses the compiler to identify not only load instructions that cause speculation to fail, but also the corresponding stores. Consequently, load instructions could be issued as soon as their corresponding store instructions are executed. Details are presented in Section 5.8.

Hardware Support for Fast Inter-Thread Register-Resident Value Forwarding

The *critical forwarding path* introduced by synchronizing register-resident values can serialize parallel execution and degrade performance. It includes two components, (i) the time required to compute the forwarded value and (ii) the time required to forward the value between two processors. Our group's previous work [64] proposed to reduce the former by prioritizing instructions required to compute forwarded value in the reorder buffer and computing the forwarded value early. Krishnan *et al.* [40] proposed to reduce the latter using a dedicated distributed register file system that is referred to as the *distributed scoreboard*. Since synchronization for register-resident value is often the performance bottleneck during parallel execution, both approaches have been demonstrated to be effective. However, we believe that the compiler, having the global knowledge of the entire program, can schedule instructions across a larger distance and can thus reduce the *critical forwarding path* more aggressively. In addition, the compiler-based approach eliminates the need for expensive hardware support. Details are presented in Section 5.7.

Value Prediction for Register-Resident and Memory-Resident Values

Hardware can also avoid the costs of inter-thread value communication through value prediction [15, 43, 44, 64]. Previous work [44] has shown that value prediction can be used efficiently to communicate register-resident values, but it is less efficient at communicating memory-resident values. Cintra *et al.* [15] and our group's [64] previous studies have also pointed out that since incorrect value prediction can incur expensive recovery operations, it should be applied only to values that are predictable.

1.3.2 Improving Inter-Thread Value Communication Using Compiler Techniques

In this section, we briefly discuss two previous proposals that rely on the compiler to manage inter-thread values communication.

Multiscalar [25, 26, 58] developed compiler algorithms to insert synchronization and to move `signal`s and their dependent instructions early within a thread (which is also referred to as a *task* in Multiscalar). However, the Multiscalar scheduler was designed specifically for Multiscalar tasks, which usually consist of only a few basic blocks and do not contain procedure calls or loops. In contrast, the work in this dissertation targets threads that are much larger on average and may contain complex control flow. We propose and implement a dataflow-based scheduler that is able to move instructions past inner loops and procedure calls. More implementation details and a contrast to Multiscalar are presented in Section 4.2 and the performance comparison is in Section 5.7.2.

The Superthreaded architecture [67] proposed a different execution model. In this model, each thread first computes the addresses stored to by the current thread on which later threads could be dependent (a.k.a target store addresses), and forwards these addresses to its successors. It then forwards the contents of these addresses as they become available. The consumer thread, on the other hand, will stall before it loads a value from a forwarded target store address. However, computing all the *target store addresses* early requires significant instruction reordering by the compiler, which is not always possible. Thus, it is difficult to obtain the desired parallel overlap.

Concurrently with our work, Zilles and Sohi [75, 76] proposed the master/slave speculative execution paradigm by having a master thread execute a *distilled* version of the program that orchestrates and predicts the values for slave threads. In this scheme, communicated values are pre-computed by the master thread and distributed to the slave threads (as opposed to being updated and forwarded between consecutive speculative threads). Generating the distilled code for the master thread essentially requires us to identify the critical forwarding path for all values that require communication, and the instruction scheduling techniques described in Section 4.3 can be used for this purpose.

1.4 Dissertation Contributions

The primary contributions of this dissertation are the following:

- The proposal of using profiling-based compiler techniques to automatically synchronize frequently occurring memory-resident value dependences to avoid speculation failure. This approach, which aims to improve performance by synchronizing frequently occurring data dependences, trades off the costs of the recoveries from failed speculations with the costs of synchronization stalls. Because the existence of potential aliasing between memory accesses makes accurate static analysis difficult, this research relies on profiling information to identify instructions that need synchronization and on hardware support to ensure correct execution.

- The proposal of using dataflow-based instruction scheduling algorithms to reduce the critical forwarding path. This research identifies the critical forwarding path introduced by register-resident value synchronization as a key performance bottleneck for many applications and presents novel dataflow-based instruction scheduling algorithms to reduce the length of the critical forwarding path. With the proper hardware support, this technique has also been extended to speculatively schedule instructions across control and data dependences within a thread.

- A thorough evaluation of the proposed compiler algorithms on a detailed architectural model with the help of a simulator. The evaluation shows that the proposed compiler-based techniques are effective in reducing the cost of inter-thread value communication.

- A comparison between compiler-based and hardware-based inter-thread value communication techniques. This demonstrates that compiler-based instruction scheduling techniques are more effective in reducing the critical forwarding path than hardware-based techniques, since the compiler, having global knowledge of the entire program, is able to schedule instructions across a larger distance. On the other hand, this study also shows that the compiler and the hardware should work in tandem to avoid speculation failures, since each of the two techniques benefits a different set of benchmarks.

Chapter 2

Hardware and Software Support for TLS

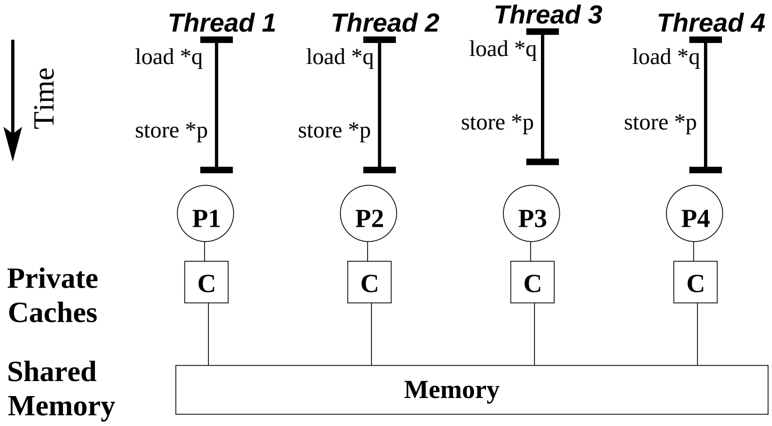
This chapter presents a thorough description of the hardware and software requirements for implementing thread-level speculation. It begins with a high-level introduction of the TLS execution model to establish the background for the rest of this chapter, followed by a section on the necessary hardware support. It then defines the hardware and software interface that allows the compiler to manipulate speculative execution. Finally, it demonstrates how various compiler passes can be combined to generate optimized parallel programs from sequential applications.

2.1 Execution Model

This section describes the TLS execution model targeted by the compiler, with particular focus on inter-thread value communication. This execution model targets a single-chip multiprocessor in which each processor has its own private first-level cache while sharing a second-level cache, as shown in Figure 2.1. A sequential program is

```
while(continue_cond) {
  ...
  load *q;
  ...
  store *p;
  ...
}
```

(a) Example loop with ambiguous inter-thread data dependence.



(b) TLS execution.

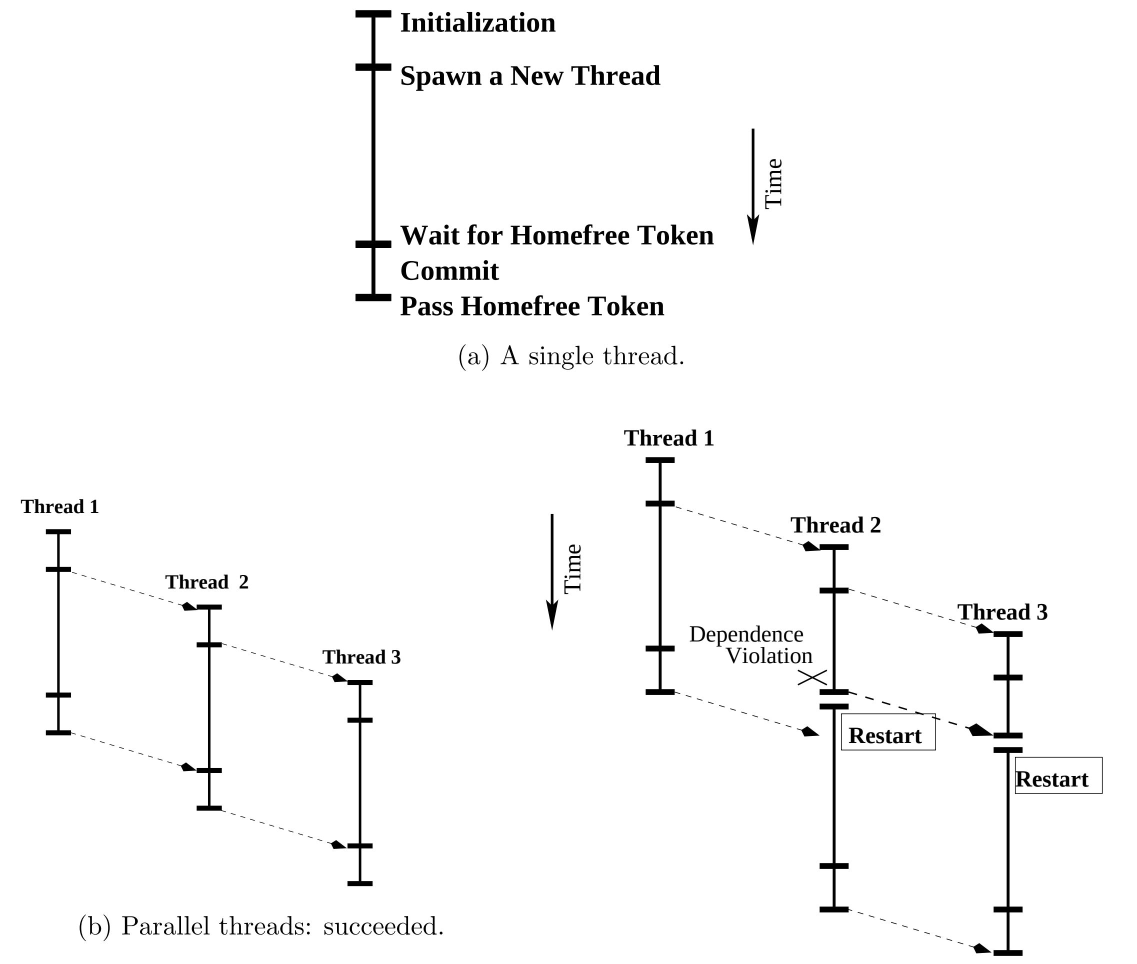
Figure 2.1: Basic architecture and execution model for TLS.

divided into threads, and each thread is assigned to a different processor. These parallel threads can be potentially dependent, but are executed *speculatively* in parallel. We rely on the underlying hardware to detect inter-thread data dependences and recover from incorrect execution. Figure 2.2(a) shows the life cycle of a single thread: each thread is first created by its predecessor through a lightweight fork, called a `spawn`, and executed in parallel with its predecessor. All threads must be committed sequentially to preserve the original execution order. This is achieved with the help of a *homefree token*: by obtaining this token, a thread can ensure that all previous threads have made all of their speculative modifications visible to the memory system and hence it is safe to commit. The thread that holds the token is *homefree*, and speculation succeeds if there is no inter-thread data dependence. This execution model allows us to exploit thread-level parallelism, as shown in Figure 2.2(b). However, when there is an inter-thread data dependence, the hardware will detect the dependence violation, and restart the thread containing the consumer of the dependence. All logically later threads must also be restarted, as shown in Figure 2.2(c).

2.1.1 Inter-Thread Value Communication

The goal of this section is to examine the value communication mechanisms available under TLS from the compiler's perspective. TLS supports two forms of communication and the compiler can decide which mechanism is appropriate for a particular data dependence to obtain maximum parallel overlap:

Synchronization explicitly forwards a value between the source and the destination of a data dependence, as shown in Figure 2.3(a). It allows for partial parallel overlap and is thus suitable for frequently occurring data dependences that can be clearly identified. However, if the instructions that compute the communicating value are sparsely located in a thread, explicit synchronization could also

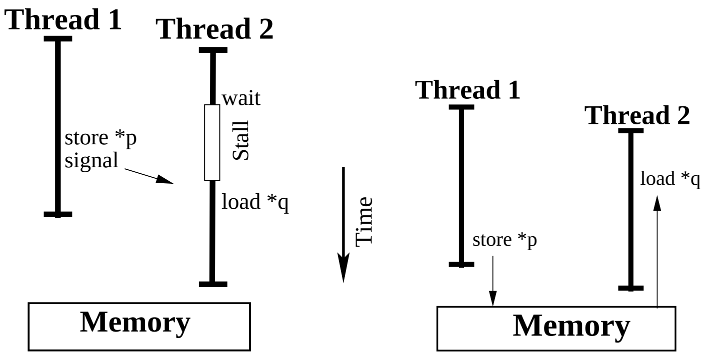


(a) A single thread.

(b) Parallel threads: succeeded.

(c) Parallel threads: violated.

Figure 2.2: TLS execution model.



(a) Value communication via synchronization.

(b) Value Communication via speculation.

Figure 2.3: TLS value communication model.

limit performance by stalling the consumer threads more than necessary.

Speculation relies on the underlying hardware to detect data dependence violations at runtime and trigger re-execution when necessary, as shown in Figure 2.3(b). It allows for maximum parallel overlap when speculation always succeeds, however, if speculation always fails, this mechanism would introduce a significant performance penalty. Thus, this form of value communication is suitable for data dependences that are difficult to analyze and occur rarely.

2.2 Hardware Support

This section describes the hardware required to support speculative threads. There has been many proposals on hardware support for TLS [1, 15, 29, 31, 32, 43, 51, 63, 67], and a thorough treatment is beyond the scope of this dissertation. We will describe only the TLS hardware support used as the framework of this study.

TLS support has two key components: (i) recovering from incorrect execution when speculation fails, and (i) detecting inter-thread data dependences at runtime. Our hardware support uses the private first-level cache to keep track of speculative accesses to the memory system. Each cache-line is extended in the first-level cache with two extra bits: a speculatively modified bit (SM) and a speculatively loaded bit (SL). When a thread is executing speculatively, all speculative modifications are buffered in the first-level cache and these modifications only affect the memory system when the thread becomes non-speculative.

Detecting inter-thread data dependences involves comparing the addresses of load and store instructions belonging to different threads and checking whether they have violated data dependence constraints defined by the sequential execution. This can be done by having the producer thread report to the consumers the locations that it has modified and by having the consumer check whether these locations have been speculatively consumed. We have extended the invalidation-based cache coherence protocol to implement this functionality: whenever a cache line shared by multiple first-level caches is modified, an invalidation message is sent to the cache that has a copy of the line along with an identification number of the thread that performs the modification. The identification number is a logical time-stamp that corresponds to the sequential order of the threads. The consumer thread records the locations it has *speculatively* loaded, and whenever a logically earlier thread modifies the same location (as indicated by an invalidation message from a logically earlier thread), a violation is issued. This process is illustrated in Figure 2.4, with three speculative threads, `thread 1, thread 2 and thread 3`. `Thread 1` and `thread 3` both perform a speculative load from address `0x88`, so the speculatively modified bit is set for the corresponding cache line. When `thread 2` stores to that same cache line, it generates an `invalidation` message containing its thread identification number to the other two processors. Processors that have speculatively loaded from this cache

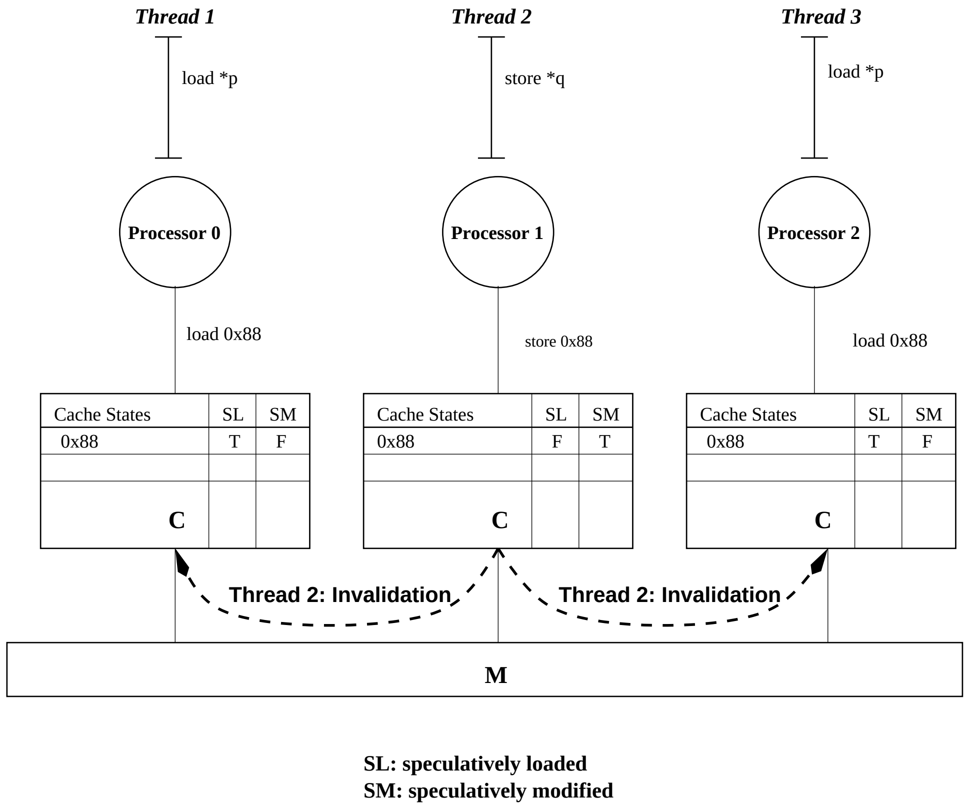


Figure 2.4: Hardware support for TLS.

line, in this case `thread 1` and `thread 3`, are potential candidates for speculation failure. However, data dependence is violated only when the invalidation message comes from a logically earlier thread; for instance, `thread 3` is violated in the figure. With this implementation, TLS can be supported with minimum hardware.

2.3 Hardware/Software Interface

For the compiler to initiate and maintain parallel execution, it is necessary for the underlying system to provide the compiler with the ability to manage parallel threads, manipulate speculative states, and schedule inter-thread value communication. For this purpose, a set of simple primitives [61, 62] are defined. There are a number of

```
counter = 0;
value = 77;
do {
    value = work(value);
    counter++;
} while (loop_test());
```

(a) The original loop.

```
counter = 0;
value = 77;
do {
    value = work(value);
    counter++;
    next_iteration = loop_test();
    if (!next_iteration)
        break;
} while (1);
```

(b) The test for loop termination is folded in the loop body to ease parallel transformation.

Figure 2.5: A simple `while` loop.

issues to consider for such an interface. While some issues are analogous to those for traditional parallel applications, such as creating threads and managing the stacks, others are unique to TLS, such as passing the *homefree* token and recovering from failed speculation.

In this section, we describe the primitives used to perform each task by the compiler. We will illustrate how a simple *while* loop, written in sequential language, as shown in Figure 2.5(a), is transformed into a parallel program. This loop has: (i) undetermined loop bounds, (ii) ambiguous data dependences since the computations residing in work() and loop_test() can be aliased, and (iii) scalar dependences through two variables, `counter` and `value`. The test for loop termination is folded into the loop body to ease parallel transformation, as shown in Figure 2.5(b).

```
do {
  tls_td = spawn(); ............................ ⓐ
  if (tls_td != 0) {
    value = work(value);
    counter++;
    next_iteration = loop_test();
    if (tls_td != SPAWN_FAILED) {
      if (next_iteration)
        end_thread(); .........................  ⓑ
      else
        break;
    }
    else if (!next_iteration)
        break;
  }
} while (1)
```

Figure 2.6: Interface for thread creation and termination.

2.3.1 Thread Creation and Termination

The compiler can initiate a parallel thread using `spawn()`. `Spawn()` returns twice, once to the parent and once to the child. In the parent, the thread descriptor of the child thread is returned, and in the child, 0 is returned if the fork succeeds. Each thread creates its successor at the beginning of execution to maximize parallel overlap, as shown in line ⓐ in Figure 2.6. The thread descriptor is an identifier which can be used for further communication from the parent to the child. When the thread has completed its work, it invokes `end_thread()` to terminate, as shown in line ⓑ. If the fork fails, due to insufficient resources (such as no more processors being available), it returns the special value `SPAWN_FAILED` to the parent. The spawn mechanism forwards initial parameters to the appropriate processor.

```
do {
  tls_td = spawn();
  if (tls_td != 0) {
    value = work(value);
    counter++;
    next_iteration = loop_test();
    wait_for_homefree_token(); .......................... ⓒ
    commit_speculative_writes(); ........................ ⓔ
    if (tls_td != SPAWN_FAILED) {
      if (next_iteration) {
        pass_homefree_token(tls_td); .................... ⓓ
        end_thread();
      }
      else
        break;
    }
    else if (!next_iteration)
      break;
  }
} while (1)
```

Figure 2.7: Interface for managing homefree token.

2.3.2 Homefree Token

To ensure that speculative threads are committed in sequential order as defined by the sequential program, a homefree token is passed between the active threads using the primitives shown in Figure 2.7. The compiler is responsible for (i) waiting for homefree when it has completed execution, as shown in line ⓒ; (ii) committing its speculative states to the memory system once it holds the homefree token, as shown in line ⓔ; (iii) and passing the homefree token to its successor before thread termination, as shown in line ⓓ. The thread that holds the homefree token is the oldest thread and is referred to as the homefree thread.

2.3.3 Speculative State Manipulation

A thread can execute in one of two modes, the speculative mode or the non-speculative mode. When a thread is executing in the speculative mode, all modifications made to the memory system stay in the local cache and do not affect the value in the main memory. When a program is executing in non-speculative mode, all modifications made to the memory system are shared with all the other threads using standard memory consistency mechanisms.

When a thread is created, it executes in the non-speculative mode by default. The thread continues this mode of execution as long as the compiler is able to determine that this thread is independent of all active earlier threads. When the compiler decides that there is not enough static information to ensure independence, it switches to the speculative execution mode by invoking `become_speculative()`, as shown in line f in Figure 2.8. All load operations that are performed in the speculative execution mode of the thread are subject to violation if the address from which a load operation retrieved a value was written to by a store in an earlier thread. When a thread decides to commit all speculative modifications to the memory, it invokes `become_nonspeculative()`, as shown in line g in Figure 2.8. At this point, the thread is ready to commit all speculative loads and stores by waiting for the homefree token.

2.3.4 Stack Management

A key design issue in TLS is the management of references to the stack. A naive implementation would maintain a single stack pointer (shared among all threads) and a stack in memory that is kept consistent by the underlying data dependence tracking hardware. The problem with this approach is that speculation would fail frequently and unnecessarily: for example, whenever multiple threads store values

```
do {
  tls_td = spawn();
  if (tls_td != 0) {
    become_speculative(); .......................... (f)
    value = work(value);
    counter++;
    next_iteration = loop_test();
    wait_for_homefree_token();
    become_nonspeculative(); ....................... (g)
    commit_speculative_writes();
    if (tls_td != SPAWN_FAILED) {
      if (next_iteration) {
        pass_homefree_token(tls_td);
        end_thread();
      }
      else
        break;
    }
    else if (!next_iteration)
      break;
    }
  }
}
```

Figure 2.8: Interface for managing speculative modes.

to the same location on the stack. These dependence violations would effectively serialize execution. In addition, whenever the stack pointer is modified, the new value must be forwarded to all successive threads. An alternative approach is to create a separate *stacklet* [28] for each thread to hold local variables, spilled register values, return addresses, and other procedure linkage information. These stacklets are created at the beginning of the program, assigned to each of the participating processors, and re-used by the dynamic threads. The stacklet approach allows each thread to perform stack operations independently, allowing speculation to proceed unhindered. Figure 2.9 shows how to switch between stacklet and the regular stack: before entering a parallel region, the stack pointer is saved, as shown in line h, and after exiting a parallel region, the regular stack pointer is restored, as shown in line i.

2.3.5 Exception Handling

In this section, we will discuss how three different types of exceptions, *cancel-thread*, *violate-thread*, and *hardware* exception, are handled. The *cancel thread* exception can only be invoked explicitly by the compiler with the `cancel_thread` instruction. This instruction takes one argument, a thread descriptor, and it will send a signal that causes an exception at the targeted thread. The *cancel thread* exception is very useful in the following scenario: for an unbounded loop, a thread can be spawn beyond the last iteration of the loop. When the loop iteration test fails and a thread realizes that it has reached the last iteration of a loop, it has to kill all its successors. This can be done with a `cancel_thread` instruction specified by the compiler, as shown in line j in Figure 2.10. A thread must first register a cancel thread handler, as shown in line l, and asynchronously jump to it when a *cancel thread* exception occurs. The exception handler defines the operation to perform when the exception is invoked. In the case of handling an unbounded loop, it is necessary to cancel all successors and

```
saved_sp = save_sp(); ......................... (h)
do {
  tls_td = spawn();
  if (tls_td != 0) {
    become_speculative();
    value = work(value);
    counter++;
    next_iteration = loop_test();
    wait_for_homefree_token();
    become_nonspeculative();
    commit_speculative_writes();
    if (tls_td != SPAWN_FAILED) {
      if (next_iteration) {
        pass_homefree_token(tls_td);
        end_thread();
      }
      else
        break;
    }
    else if (!next_iteration)
      break;
  }
}
restore_sp(saved_sp); ......................... (i)
```

Figure 2.9: Interface for stack manipulation.

then terminate, as shown in lines ⓚ. When no actions are defined, the default action is taken, which erases all speculative states and restarts the execution from the first speculative instruction.

The *violate-thread* exception can be invoked either by an explicit `violate_thread` instruction or by the hardware when a data dependence violation is detected. *Violate-thread* exceptions and *hardware* exceptions can be handled in a similar manner as the *cancel thread* exceptions. Although the default action for a thread upon receiving a *violate thread* exception is to restart all speculative execution, the `set_violation_handler()` primitive could offer an alternative re-start point. The `set_violation_handler()` and the `set_cancel_handler()` instructions are similar to the semantics of the Unix `set_jmp()` instruction: when called normally from the program, the function returns `-1`; when returned from an exception, it returns the violation distance, which is the difference between the thread identification number of the consumer and that of the producer. Hardware exceptions, such as null-pointer-deferences and divided-by-zero occurring in speculative threads, should not cause the entire program to terminate, since they could be caused by inconsistent speculative states. Hence, if a hardware exception occurs during speculative execution, speculation will fail and the thread is re-executed.

2.3.6 Inter-Thread Value Communication

The underlying TLS hardware is able to check and ensure that *all* memory dependences are preserved at execution time without explicit directions from the compiler. An alternative mechanism for inter-thread value communication, namely explicit synchronization, is also available for communicating values. This mechanism provides the compiler with the ability to perform point-to-point synchronization between the producer and the consumer threads of a data dependence through a *forwarding frame*.

```
saved_sp = save_sp();
do {
  tls_td = spawn();
  if (tls_td != 0) {
    become_speculative();
    if (set_cancel_handler() == -1) { ..................... (l)
      cancel_thread(tls_td); ............................. (k)
      end_thread(); ...................................... (k)
    }
    value = work(value);
    counter++;
    next_iteration = loop_test();
    wait_for_homefree_token();
    become_nonspeculative();
    commit_speculative_writes();
    if (tls_td != SPAWN_FAILED) {
      if (next_iteration) {
        pass_homefree_token(tls_td);
        end_thread();
      }
      else {
        cancel_thread(tls_td); .......................... (j)
        break;
      }
    }
    else if (!next_iteration)
        break;
  }
} while (1)
restore_sp(saved_sp);
```

Figure 2.10: Interface for handling exception.

The forwarding frame is a structure on the stack where all data items are thread-private. In this case, the synchronization mechanism is used to communicate value for the two scalar dependences.

A program with such synchronization inserted is shown in Figure 2.11. Before entering the speculatively parallelized region, the forwarding frame is first declared (lines m and n). The initial values are also loaded onto the forwarding frame (line o). Upon exiting the speculatively parallelized region, values are copied from the forwarding frame back to the registers (line p). The entire forwarding frame is copied from the parent to the child thread when the child is created. During the execution of a thread, explicit communication can be performed on individual data items on the forwarding frame by specifying the offset of the data item on the forwarding frame. In this example, two scalar values, `counter` and `value`, are communicated through synchronization between the speculative threads. `Counter` is an inductive variable, and thus it is computed before the successor is spawned so that its value is available when thread is created, as shown in lines q. However, the value of `value` is not available until much later in the thread. So, the producer thread uses the `send_value(destination_thread, offset, value)` instruction, as shown in line r, to remotely write `value` to the forwarding frame of the destination thread at the location specified by *offset* and set a flag to indicate that the value is available. The consumer of the data dependence, on the other hand, checks if the flag is ready with the `wait_for_value(offset)` instruction, as shown in line s.

2.4 Compiler Support

We rely on the compiler to define where and how to speculate. Our compiler infrastructure is based on the Stanford SUIF 1.3 compiler system [65], which operates on `C`

```
saved_sp = save_sp();
struct forwarding_frame { int counter; int value; } ff; ...... ⓛ
forwarding_frame(&ff); .......................................... ⓜ
forwarding_size(sizeof(forwarding_frame)); .................... ⓝ
ff.counter = 0; .................................................ⓞ
ff.value = 77; .................................................. ⓞ
while(1) {
  tls_td = spawn();
  local_counter = ff.counter; .................................. ⓠ
  ff.counter = local_counter + 1; .............................. ⓠ
  if (tls_td != 0) {
    become_speculative();
    if (set_cancel_handler() == -1) {
      cancel_thread(tls_td);
      end_thread();
    }
    wait_for_value(4); .......................................... ⓢ
    ff.value = work(ff.value);
    send_value(tls_td, 4, ff.value); ..........................   ⓡ
    next_iteration = loop_test();
    wait_for_homefree_token();
    become_nonspeculative();
    commit_speculative_writes();
    if (tls_td != SPAWN_FAILED) {
      if (next_iteration) {
        pass_homefree_token(tls_td);
        end_thread();
      } else {
        cancel_thread(tls_td);
        break;
      }
    }
    else if (!next_iteration)
        break;
  }
}
counter = ff.counter; .......................................  ⓟ
value = ff.value; ............................................. ⓟ
restore_sp(saved_sp);
```

Figure 2.11: Interface for value communication.

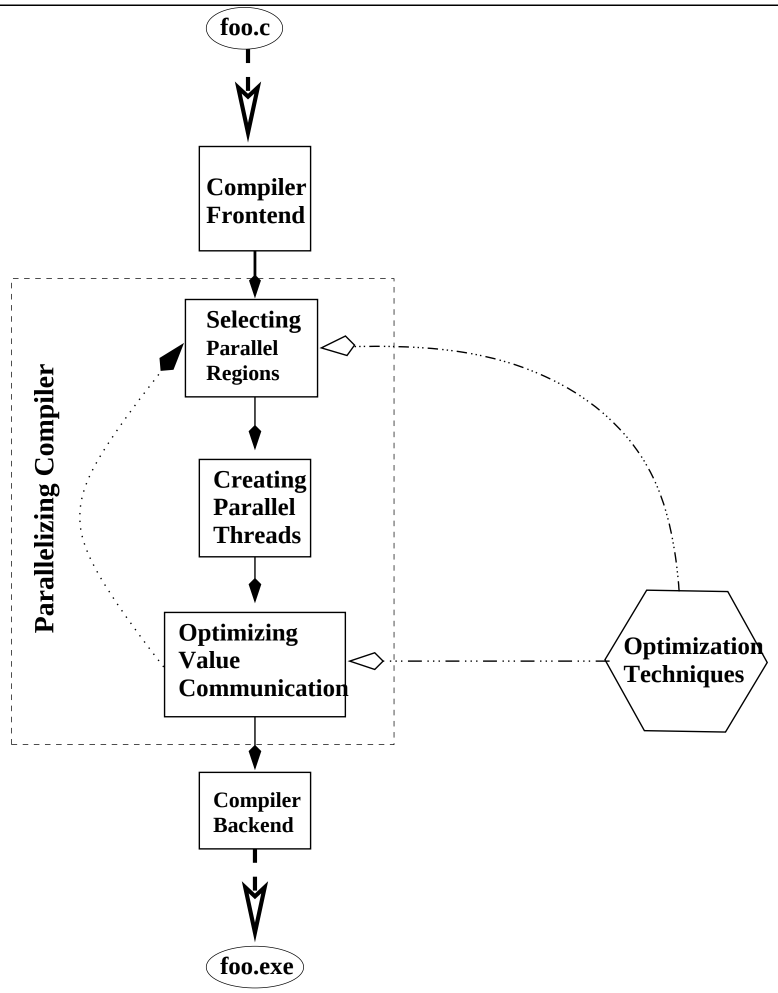


Figure 2.12: Compiler infrastructure.

code. For `Fortran` applications, the source files are first converted to `C` using `sf2c`[1] and then converted to SUIF format. Our infrastructure is illustrated in Figure 2.12. The parallelizing compiler consists of the following passes: (i) selecting a set of loops that is likely to maximize program performs, (ii) inserting parallelization primitives to create parallel programs, and (iii) optimizing inter-thread value communication to create efficient codes.

2.4.1 Parallel Loop Selection

One of the most important tasks in a thread-speculative system is to decide which portions of code to speculatively parallelize [16]. Because a thorough treatment of region selection is beyond the scope of this work, only a brief discussion of the issues involved is provided. This dissertation focuses on loops for two reasons: first, loops take up a significant portion of the execution time (coverage) and hence can have large impact on program performance; and second, loops are fairly regular and predictable, hence straightforward to transform into threads.

Here is a brief description of the loop selection process. First the following filter is applied to remove from consideration loops that have no potential for speedup under TLS:

- coverage (fraction of dynamic execution) is less than 0.1% of execution time;
- there is less than one iteration per invocation (on average);
- the number of instructions per iteration is more than 16000 (on average);
- the total number of instructions per loop invocation is less than 30 (on average);
- loops contain calls to `alloca()`.[2]

[1]We have made minor manual modifications to limit the scope of certain variables.

[2]`alloca` would interfere with stack management.

All loops that have passed this filter are evaluated twice, the first time running sequentially to estimate the performance baseline, the second time running in parallel to estimate the performance of parallel execution. Thus, for each loop there is a gain factor that is the performance improvement obtainable for the entire program if this loop is executed in parallel. The gain factor is used to decide which loops are best to parallelize for maximizing program performance. The decision must take into consideration loop nesting: two loops that can both speed up relative to sequential execution can be nested. This relationship is represented by loop graphs, as shown in Figure 2.14, in which each node is a loop in the original program. A directed edge indicates loop nesting either in the form of directly nested loops or nested through procedure calls. To identify the set of loops that maximize total gain, the following gain-distribution procedure is applied to each node in the graph in reverse depth-first-search order.

1. Calculate the performance improvement if this node is parallelized.
2. Sum the performance gain at all child nodes.
3. Choose the greater of the two values to be the gain at this node.
4. If the value of parallelizing this node is greater, disconnect all out going edges.
5. Remove nodes with zero or negative gain at the graph.
6. For nodes with multiple parents, divide the gain equally between its parents by assuming all path are equally likely (a simplification).

Once this procedure is completed, the leaf nodes of the graph are selected for parallelization.

The last piece of the puzzle for region selection lies in estimating the parallel execution time for each loop. Note that, on one hand, the parallel execution time

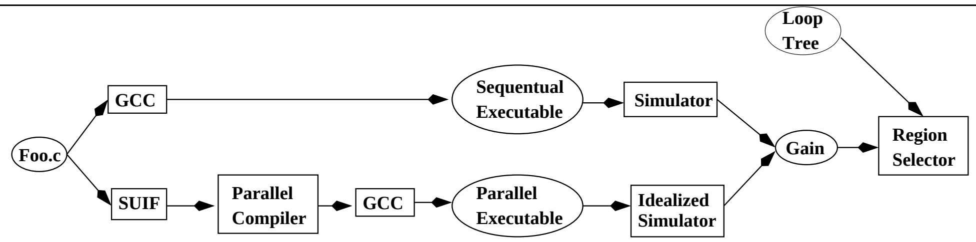


Figure 2.13: Region selection process.

used for region selection must be optimized for value communication; on the other hand, what optimizations are appropriate depends on the set of loops chosen by the region selection process. Thus, the region selection and optimization selection are interdependent. To break the deadlock, we equip the region selection pass with the knowledge of the available optimization techniques, as shown in Figure 2.12.

The actual implementation uses a hypothetical simulation model, as illustrated in Figure 2.13: (i) a sequential and a parallel version (with no optimization for value communication) of a program are created; (ii) the sequential version runs through a simulator to obtain the sequential execution time for each loop; (iii) the parallel version runs through the hypothetical simulation model to obtain the upper-bound of parallel execution time for each loop; (iv) the execution time from the two simulations are compared and a gain factor for each loop is generated. The hypothetical simulation model ignores the cost of communication for a subset of data dependences that could potentially be optimized; These gain factors are fed into the region selector algorithm described earlier.

2.4.2 Parallelization

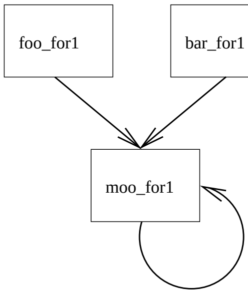
Once speculative regions are chosen, the compiler inserts the TLS instructions (described in Section 2.3) that interact with hardware to create and manage the specu-

```
void foo()  {
   for (i = 0; i < 10; i++)
      printf("%d\n", moo(i));
}

void bar()  {
   for (i = 0; i < 20; i++)
      printf("%d\n", moo(i));
}

int moo(int i)  {
   sum = 1;
   for (j = 0; j < i; j++)
      sum = sum + moo(j);
   return sum;
}
```

(a) Source code.



(b) Loop graph.

Figure 2.14: Converting a program into a loop graph.

lative threads and forward values.

2.4.3 Optimization

Without optimization, execution can be unnecessarily serialized due to either excessive data dependence violations or over-synchronization. To address these issues, two optimization techniques are applied. The first optimization is identifying and synchronizing frequently occurring data-dependences since they correspond to the majority of violations. This optimization is described in detail in chapter 3. The second optimization is scheduling instructions to reduce the critical forwarding path since synchronization could also serialize execution by stalling the consumer thread longer than necessary. This optimization is described in detail in Chapter 4.

2.4.4 Code Generation

Our compiler takes a sequential program and produces `C` source code which includes the TLS instructions as in-line MIPS assembly code using `gcc`'s "`asm`" statements.

This source code is then compiled with `gcc` v2.95.2 using the "`-O3`" flag to produce optimized, fully functional MIPS binaries containing new TLS instructions.

2.5 Chapter Summary

This chapter has provided the details of the hardware and software support for thread-level speculation. It has explained how the cache coherence protocol is extended to detect data dependence violations and buffer speculative execution. It has also described the compiler infrastructure that creates speculative parallel executables from sequential programs. With the help of an example loop, it has demonstrated how the compiler can manipulate speculative execution and schedule inter-thread value communication using simple primitives. The rest of this dissertation is built upon the infrastructure described in this chapter.

Chapter 3

Automatic Synchronization

In TLS, it is desirable to synchronize frequently occurring data dependences and to speculate on infrequently occurring ones for performance purposes. This chapter describes the compiler techniques for automatically inserting synchronization for both register-resident and memory-resident values.

3.1 Synchronization Versus Speculation

In thread-level speculation, speculation failures incur high costs and thus should be invoked only occasionally. Consequently, alternative methods to deal with frequently occurring data dependences between speculative threads must be sought. One way to reduce speculation failures caused by data dependence violations is to synchronize frequently occurring data dependences. Figure 3.1 shows an example loop that the compiler has speculatively parallelized by turning each loop iteration into a thread. In each thread, a value is loaded through the pointer `p` and another value is stored through the pointer `q`. When `p` in a later thread points to the same memory location as `q` in an earlier thread, there is a *read-after-write* dependence. Figure 3.1(b) and 3.1(c) show two methods of communicating a value between these two threads.

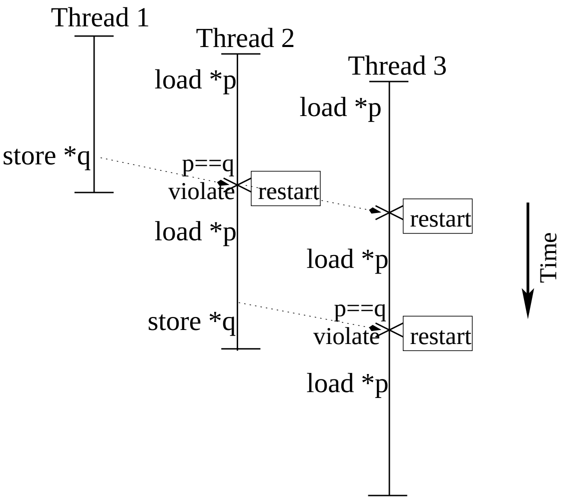
The first method is *speculation*: the consumer thread executes as if there were no data dependence with previous thread and is re-executed if the hardware detects a dependence violation. The second method is *synchronization*: the consumer thread stalls and waits for the producer thread to produce and forward the correct value. Synchronization serializes parallel execution and only allows partial overlap between parallel threads, but is more efficient than speculation when data dependences occur frequently, since restarts are avoided.

Value communication for register-resident values and memory-resident values are handled differently. Register-resident value dependences are likely to occur frequently, and thus it makes sense to always synchronize them. Furthermore, since TLS does not preserve data dependences between register-resident values, compiler scheduled synchronization for register-resident scalars is not only a performance issue, but also a correctness constraint. Fortunately, automatic synchronization of scalar values is relatively straightforward, since scalar values are explicitly named. Even though complex control flow is pervasive in general purpose applications, it is still relatively easy to identify the first use and the last definition of each scalar and to insert synchronization with the help of data-flow analysis.

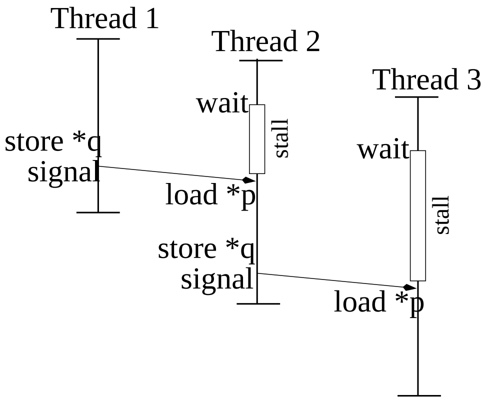
Memory-resident values are more difficult to synchronize due to aliasing. Figure 3.4(a) on page 52 shows three threads running speculatively in parallel. `Load *p` can potentially depend on any of the five stores in the figure, although each access to the memory uses a different pointer. The compiler must prove that `load *p` depends on `store *q` in all possible executions before synchronizing the two instructions and directly forwarding a value between them. However, for general-purpose applications, such proofs are difficult and sometimes impossible to construct. If the compiler decides to synchronize `store *q` and `load *p` without such a proof, we must confirm at runtime that (i) `p` and `q` refer to the same memory location, and that (ii) stores through pointers `y` and `z` do not modify this location.

```
do {
  work1();
  ...= *p;
  work2();
  *q = ...;
  work3();
} while (1);
```

(a) Code with communication through memory-resident values.



(b) Communicating through speculation: if `p` and `q` alias between threads then the later threads are violated and restarted with the correct value.



(c) Communicating through synchronization: the later threads always stall until the value is available.

Figure 3.1: Performance trade-off between speculation and synchronization.

Previously, a number of studies [15, 47, 64] proposed using hardware implementations to dynamically insert synchronization for frequently occurring and unpredictable data dependences in TLS. Moshovos *et al.* [47] demonstrated how to identify frequently occurring data dependences with a centralized structure. However, a centralized structure can limit performance [30] and is difficult to scale. On the other hand, using distributed structures to perform the same task is much more complicated, since it involves replicating and updating the tables that predict/synchronize load-store pairs via broadcast. In a distributed environment, it is relatively easy for the hardware to identify dynamically loads that frequently cause speculation to fail using hardware lookup tables, but it is more involved to identify the corresponding stores. To dynamically identify an inter-thread dependence pair, the hardware has to (i) compare the addresses accessed by loads and stores in different threads, and (ii) dynamically determine whether a store is the last store that modifies an address in a thread. To avoid this complexity, recent proposals for hardware-inserted synchronization [15, 64] choose to delay the load instructions until previous threads have committed. However, such simplification tends to stall the consumer threads more than necessary and reduces time spent on failed speculation while incurs a higher synchronization cost. In contrast, compilers have the advantage of thoroughly analyzing the entire program, and thus can speculate on which stores are more likely to produce the desired value and therefore only stall the consumer until the value is produced (rather than wait for the entire producer thread to complete). Compilers can also schedule instructions to produce the forwarded value early to reduce synchronization time. Furthermore, compiler-inserted synchronization avoids hardware complexity by eliminating lookup tables.

3.2 Related Work on Automatic Synchronization

Previous work on synchronizing loop-carried data dependences for DOACROSS loops [3, 9, 21, 53, 70] only focuses on array-based numeric codes. The techniques described in this chapter apply to arbitrary control flow and memory access patterns in general-purpose programs and is able to (i) forward data for dependences that may or may not occur and (ii) ensure correct execution if subsequent stores invalidate the data that have already been forwarded.

To avoid excessive failed speculations when using TLS, two types of hardware mechanisms have been proposed: *value prediction* [15, 44, 47, 51, 54, 64] and *synchronization* [15, 47, 64]. Value prediction allows the consumer of a potential data dependence to use a predicted value whenever possible to avoid a dependence violation. Automatic synchronization uses the hardware to identify store-load dependences that frequently cause violations and synchronize them dynamically.

Dynamic synchronization of memory accesses can benefit both uniprocessors and multiprocessors. In superscalars, loads are usually issued as early as possible, but no earlier than prior stores that write to the same memory address to avoid memory-order violations. Chrysos and Emer [13] present a design that uses a prediction table for synchronizing dependent store-load pairs in an out-of-order issue uniprocessor. Moshovos *et al.* [47] demonstrate how to implement a hardware-based synchronization mechanism in the context of a Multiscalar processor (a thread-speculative chip-multiprocessor) using centralized lookup tables to match dependent load/store pairs from different processing units.

A major drawback of previous proposals is the need for centralized lookup tables, which can limit performance and are difficult to scale in the context of TLS. Two groups [15, 64] propose alternative implementations to manage synchronization information in a distributed manner. Cintra and Torrellas [15] propose building a distributed hardware lookup table to keep track of frequently occurring violations. Cintra and Torrellas divide data dependences into three categories and handle them accordingly. For violations caused by false dependences, they optimistically allow the consumer to proceed and use the per-word access bits in its cache hierarchy to check for correctness before committing. In the case of a true dependence where the value is predictable, the consumer uses a predicted value and later verifies the value before committing. In the case of a true dependence with an unpredictable value, violations are avoided by stalling the consumer until the producer has committed. Their evaluation shows that these optimizations can substantially improve value communication for floating point benchmarks.

Our previous work [64] use two hardware-based techniques to avoid violation, value prediction for predictable values and dynamically inserted synchronization for unpredictable values. We found that value prediction and dynamic synchronization can incur a significant cost and should only be applied to those dependences that limit performance. Loads that frequently cause violations are delayed until the producer thread commits rather than until the desired value is produced, due to the difficulty in identifying dependent stores. Thus, such dynamically inserted synchronization can serialize parallel execution more than necessary.

3.3 Compiler-Inserted Synchronization

This dissertation proposes to use the compiler to automatically insert synchronization for two purposes: (i) to satisfy inter-thread data dependences for scalar values, and (ii) to reduce speculation failures caused by memory-resident values. Since the TLS mechanism ensures that data dependences between memory-resident values are preserved, the sole purpose of compiler-inserted synchronization for memory-resident values is to improve performance. However, the TLS mechanism does not track dependences for register-resident values, and thus compiler-inserted synchronization for these values must ensure correctness.

Traditional data-flow analyses are employed to decide where to insert synchronization for register-resident values. Traditional data-flow analyses can (i) identify all the uses and definitions of all scalars, (ii) extract inter-thread data dependences and (iii) compute the necessary synchronization. Details of these algorithms can be found in Section 3.4.

Automatic synchronization for memory-resident values aims only to improve the efficiency of inter-thread value communication for frequently occurring data dependences, since the hardware is responsible for ensuring correctness. This approach first identifies frequently occurring data dependences using profiling information, then inserts *signal* and *wait* instruction pairs—the same synchronization primitives used for communicating scalars—to create point-to-point synchronization to forward values between threads. Hardware support is used to verify that the synchronized loads and stores are indeed dependent at runtime and to recover from incorrect synchronization if otherwise. Details of this hardware support can be found in Section 3.5.

Although the research in this dissertation uses an instrumentation-based profiling tool to match up all dependent load/store pairs and to identify frequently occurring data dependences, pointer analysis [46, 69], especially probabilistic, inter-procedural and context-sensitive pointer analysis [4, 11, 42, 50], could potentially help us obtain this information with less detailed profiling information. Detailed description of the compiler passes that insert synchronization can be found in Section 3.5.2, and the details of these profiling tools are described in Appendix A.1.

3.4 Synchronizing Register-Resident Values

Register-resident values have the following characteristics which make them easy to synchronize: (i) there is no aliasing in accessing these values, since all accesses (reads or writes) must explicitly refer to the same variable name; and (ii) static instructions that access these values occur only in the body of the loop being optimized, never in the procedures called from the loop. Therefore, it is straightforward to identify all accesses to these values and to determine their last definitions and the first exposed uses using data-flow analysis. This section targets the set of register-resident values that are defined in the enclosing scope of the parallelized loop and satisfy one of the following criteria: i) any register-resident value in the intersection of the set of values with a downward-exposed definition and an upward-exposed use (i.e., the value is *live* between threads); ii) any register-resident value that is defined in the loop and is *live* when the loop exits. These values are also referred to as *communicating scalars*. In contrast with local scalars, global scalars and values referenced with pointers may be modified by instructions from the outside of the loop body; synchronization for this type of references is addressed in Section 3.5.

The compiler uses two value communication primitives, `wait` and `signal`, to forward values. The `wait` instruction stalls execution until the value is produced by the previous thread, which communicates that value through the `signal` instruction as described in Section 2.3.6. For the first thread of a speculatively parallelized loop, the `wait` instruction does not stall (since there is no producer). This section presents a general algorithm for inserting synchronization to communicate scalar values between threads.

3.4.1 Constraints on Synchronization Insertion

Properly inserted `wait` and `signal` instructions must ensure correct execution by satisfying the following constraints. First, the last write to a scalar in the producer thread must occur before the consumer thread reads that scalar, regardless of the execution path taken by either thread, formally,

1. A `wait` must occur before any use of the scalar on any path;

2. A `signal` must occur after the last definition of the scalar on any path.

In addition, the producer must ensure that a correct value is forwarded to the consumer thread despite the execution path taken, formally,

3. A `signal` must occur for all synchronized scalars on all possible execution paths.

Given these three constraints, a correct program can be created by trivially placing all `wait` instructions at the beginning of each thread and all `signal` instructions at the end of each thread. However, such a transformation would completely serialize execution. To remedy this situation, two additional constraints are used for the sake of improving performance:

4. `Wait` instructions should be placed as late as possible;

5. `Signal` instructions should be placed as early as possible.

3.4.2 Synchronization Insertion Algorithm

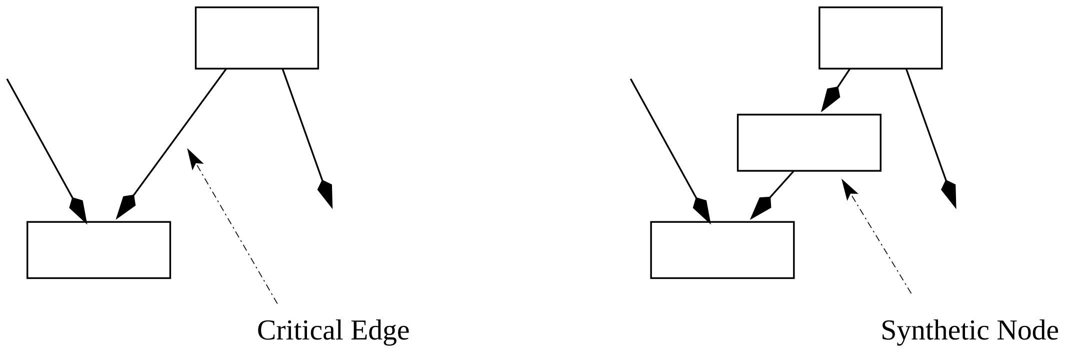
Intuitively, a synchronization insertion algorithm for `wait` instructions would involve placing a `wait` for a scalar at the beginning of the thread, and then pushing the `wait` towards the end of the thread. When a branch is encountered, the `wait` can be replicated on both sides of the branch. The motion stops when a use of the scalar is encountered. For the insertion of `signal` instructions, the converse of this algorithm is used. To decide which basic blocks to insert `wait`s and `signal`s, we have implemented dataflow analyses (described in detail below). Within a basic block, the `wait`s are placed directly before the first use of the scalar, and the `signal` is placed directly after the last definition of the scalar.

The following describes the dataflow algorithm for placing `wait` and `signal` instructions in accordance with the above constraints. While only the algorithm for inserting `signal` instructions is shown, the converse of this algorithm is used to place `wait` instructions.

The dataflow analysis is defined over the set of communicating scalars V on the control flow graph $G = (N, E, s, e)$ of a thread where N is the set of nodes which represent basic blocks, E is the set of edges, and s and e are the unique *start node* and *end node* of G (the start node and end node contain no code). *Critical edges* (i.e., any edge connecting a node with more than one successor to a node with more than one predecessor) would make it difficult to prove the correctness of our algorithm; thus, any such edges are eliminated by inserting *synthetic nodes* [37]. Figure 3.2 illustrates how synthetic nodes are inserted.

At each node $n \in N$, a predicate $\mathit{LocalDef}(n)$ is defined, which is the set of communicating scalars that are defined at n. Since the `signal` instruction that forwards the value of $v \in V$ must occur after the last definition to v on all possible execution paths, *No-More-Definitions* is defined at node n ($\mathit{NMD}(n)$) to be the set of scalars that are not defined on any execution path from n to e. This function can be computed using dataflow analysis as described in the following equation:

$$\mathit{NMD}(n) = \begin{cases} \{V\} & \text{if } n = e \\ \bigcap\limits_{s \in succ(n)} (\mathit{NMD}(s) - \mathit{LocalDef}(s)) & \text{otherwise} \end{cases} \tag{3.1}$$



(a) A control flow graph with a critical edge. (b) Synthetic node inserted to eliminate the critical edge.

Figure 3.2: Inserting synthetic nodes to eliminate critical edges.

For the example shown in Figure 3.3, the shaded boxes in Figure 3.3(c) indicate where $NMD(n)$ is true for the scalar `a`.

While it would be correct to insert `signal` instructions for a scalar at all nodes n for which $NMD(n)$ contains that scalar, this may result in multiple `signal`s being issued on a single execution path. To avoid such redundant `signal`s, we constructed the function $signal(n)$ which determines the final insertion of `signal` instructions:

$$signal(n) = \begin{cases} \{\} & \text{if } n = s \\ NMD(n) - \bigcap_{p \in pred(n)} NMD(p) & \text{otherwise} \end{cases} \tag{3.2}$$

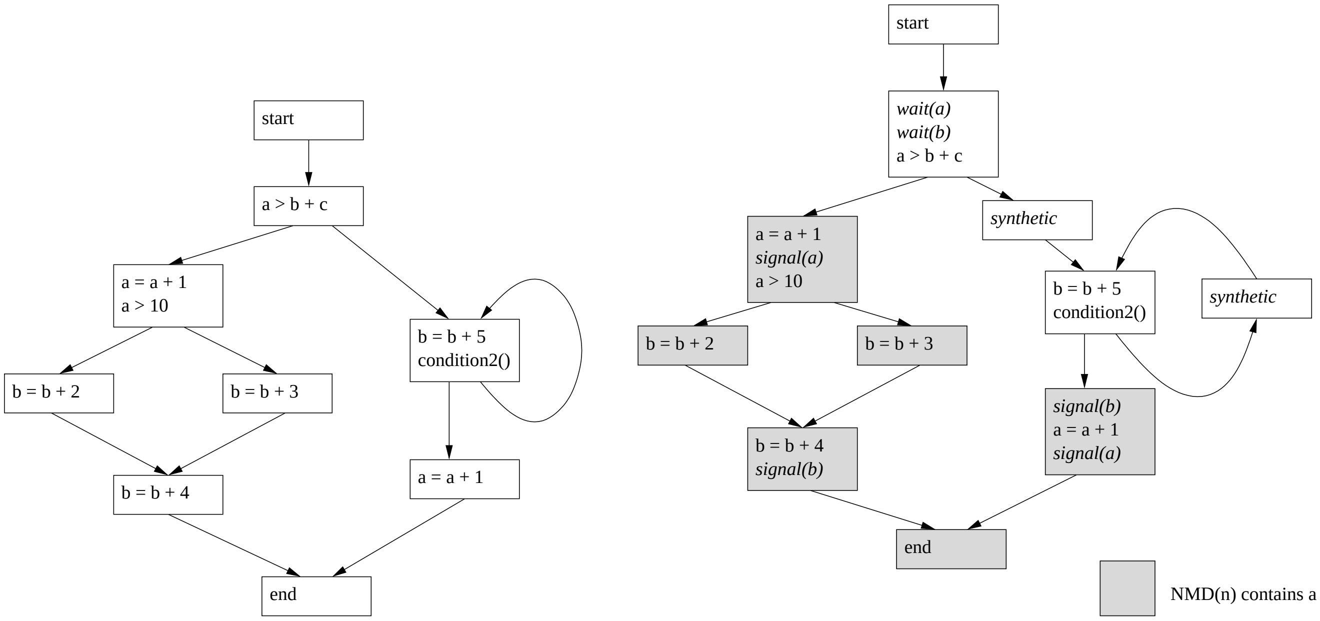
Figure 3.3(c) shows the synchronization points for variables `a` and `b` for the piece of code shown in Figure 3.3(a).

3.4.3 Proof of Correctness

This section provides the proof of correctness for the synchronization insertion algorithm described in Section 3.4.2.

```
while (condition())  {
   if (a > b + c)  {
      if (++a > 10)
         b = b + 2;
      else
         b = b + 3;
      b = b + 4;
   }
   else {
      do {
         b = b + 5;
      }
      while (condition2());
      ++a;
   }
}
```

(a) Original code.



(b) Control flow graph. (c) Inserting waits and signals.

Figure 3.3: Example of how waits and signals are inserted.

Lemma 1 *On every execution path from s to e, no definition for scalar $v \in V$ occurs after the* `signal` *for this scalar.*

Proof: Let $(s, \ldots, n_i, \ldots, n_j, \ldots, e)$ be an execution path, with $v \in \mathit{signal}(n_i)$ and $v \in \mathit{LocalDef}(n_j)$. $v \in \mathit{LocalDef}(n_j)$ implies that $v \notin \mathit{NMD}(n_i)$, while $v \in \mathit{signal}(n_i)$

implies $v \in \mathit{NMD}(n_i)$. This is a contradiction, thus the assumption must be false. Therefore, there does not exist an execution path from s to e where $v \in \mathit{signal}(n_i)$ and $v \in \mathit{LocalDef}(n_j)$ are both satisfied.

Lemma 2 *On every execution path from s to e, there exists one and only one* `signal` *for every $v \in V$.*

Proof: *Existence:* $\mathit{NMD}(e)$ contains v, by the definition of NMD, while $\mathit{NMD}(s)$ does not contain v because there is a definition for v somewhere in the thread and the start node (s) dominates all nodes in the control flow graph. Thus, for every scalar v, on every execution path through the thread, there must exist an edge (n_i, n_j) such that $\mathit{NMD}(n_i)$ does not contain v but $\mathit{NMD}(n_j)$ does. Furthermore, for every such edge, $\mathit{signal}(n_j)$ contains v. Thus, there must exist at least one node for which the *signal* function contains v on every execution path from e to s.

Uniqueness: Let $(s, \ldots, n_i, \ldots, n_j, \ldots, e)$ be an execution path, with $v \in \mathit{signal}(n_i)$ and $v \in \mathit{signal}(n_j)$. $v \in \mathit{signal}(n_i)$ implies $v \in \mathit{NMD}(n_i)$. The definition of $\mathit{NMD}()$ implies that $v \in \mathit{NMD}(m)$ for all m on the execution path (n_i, n_j). $v \in \mathit{signal}(n_j)$ implies that $\exists x$, a predecessor of n_j, such that $v \notin \mathit{NMD}(x)$. However, since x cannot be on the execution path $(n_i, \ldots, n_j)$, n_j has two predecessors. But, critical edges are removed, x can only have one successor, which is n_j. $v \notin \mathit{NMD}(x)$ implies that $v \notin \mathit{NMD}(n_j)$ or $v \in \mathit{LocalDef}(n_j)$. Now we have a contradiction and the assumption must be false. Therefore, there is only one `signal` for v on any execution path from s to e.

3.5 Synchronizing Memory-Resident Values

Synchronizing frequently occurring memory-resident values is much more complicated than synchronizing register-resident scalar values due to the existence of potential aliasing (i.e., a pointer through which the memory location in question is unexpectedly modified). The mechanism for forwarding register-resident scalar values with `signal` and `wait` instructions cannot be directly applied to forwarding memory-resident values for the following reasons. First, it is difficult or even impossible to decide, using traditional data-flow analysis, whether two memory accesses refer to the same data item when the same location can be accessed using different names through pointers [4, 11, 69]. Furthermore, the existence of potential aliasing also makes it hard to determine the last definition and the first exposed use of a data item within a thread. As opposed to synchronizing dependences at definite program points for register-resident values, we synchronize *probable* data dependences that could occur anywhere in the program in the case of memory-resident values.

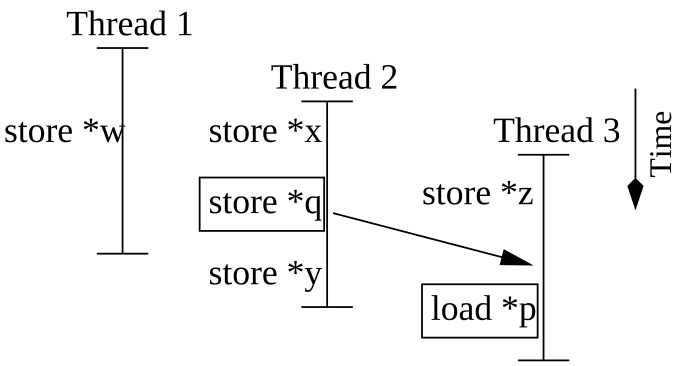
Here is a detailed description on how aliasing in memory accesses makes it more difficult to synchronize memory-resident values. An inter-thread dependence occurs between a store and a load when: (i) the store occurs in a logically earlier thread, (ii) both the store and the load access the same memory location, and (iii) no other store, between the store and load in question, modifies this location. Figure 3.4(a) shows three threads executing speculatively in parallel. Assuming that `load *p` could depend on any of the five store instructions and it depends on `store *q` most frequently, a way to synchronize and forward a value between `store *q` and `load *p` is needed. Traditional pointer analysis [4, 11, 69] can help reduce the set of pointers that `p` aliases to, but cannot provide the set of *frequently* dependent load/store pairs that are needed for synchronization. For instance, *must*-alias pointer analysis would claim that `store *q` and `load *p` are not always dependent, and hence would not synchronize them. On the other hand, *may*-alias pointer analysis would indicate that `load *p` may depend on any of the five store instructions, hence synchronizing all of them. Since neither provides the desired information in this situation, profiling-based tools are to used to identify *likely* dependences [11, 12]. Furthermore, it is also essential to provide mechanisms that allow synchronization for frequently occurring data dependences while ensuring correct execution for whatever dependences that may actually occur at runtime. The rest of this section first describes the hardware support necessary for communicating memory-resident values, then the compiler algorithms for inserting explicitly synchronization.

3.5.1 Hardware Support

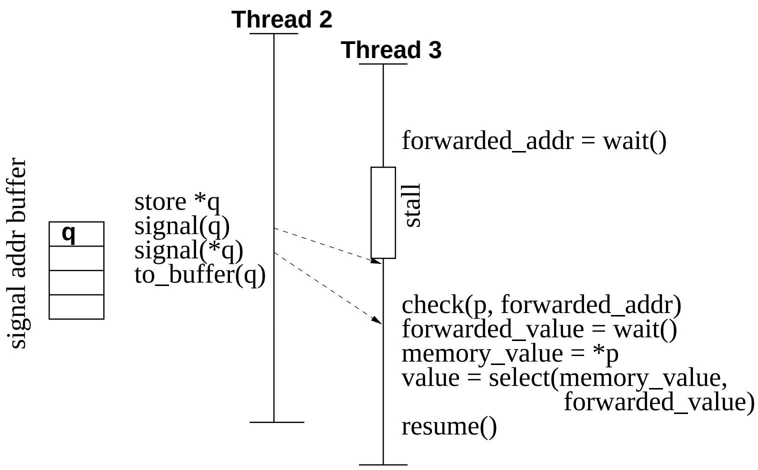
This section describes the hardware mechanism for synchronizing memory-resident values while preserving correct execution, illustration in detail in Figure 3.4.

The producer of the forwarded value must (i) store the value to the memory location, since loads from other parts of the program may read this value; (ii) forward the value being stored to the consumer; (iii) forward the address to which the values is *stored to* to the consumer; and (iv) enter the forwarded address to the `signal_address_buffer`. The `signal_address_buffer`, a small per-cpu buffer, issues a violation to the consumer thread whenever the processor attempts to modify a memory location whose address is already in the buffer.

The consumer of the forwarded value must wait for the value and its address to arrive through the `wait` instructions. Intuitively, the consumer thread then performs a list of checks to decide whether to use the forwarded value or to use the value loaded from the memory system—the forwarded value is used only when the forwarded address matches the load address and there is no prior modification to this location. The consumer first checks to see if the forwarded address matches the



(a) Original program: `load *p` often depends on `store *q`.



(b) Transformation: synchronizing `load *p` and `store *q`.

Synchronization operation	**Description**
store *q;	The original store operation.
signal(q);	Forward the address **q** to the next thread.
signal(*q);	Forward the value stored as well.
to_buffer(q);	Save the address **q** in the signal address buffer.
forwarded_addr = wait();	Wait for the address to arrive from the previous thread.
check(p, forwarded_addr);	If `p` equals `forwarded_addr` set the `use_forwarded_value` flag. Loads issued while this flag is set will not cause violations.
forwarded_value = wait();	Wait for the value to arrive from the previous thread.
memory_value = *p;	Load a value from the memory system using the load operation. If the address `p` has been previously modified by the current thread, this instruction clears the `use_forwarded_value` flag. If the `use_forwarded_value` flag is set when this load is issued, this instruction only accesses the speculative cache and will *not* cause a violation.
value = select (memory_value, forwarded_value);	If `use_forwarded_value` flag is set, select `forwarded_value`, otherwise, select `memory_value`. The selected value is placed in `value`.
resume();	Reset the `use_forwarded_value` flag.

(c) Description of operations inserted for synchronization.

Figure 3.4: Program transformation to synchronize frequently occurring memory-resident dependences between threads.

load address (to make sure a useful value is received), and if so, sets a cpu-local flag called `use_forwarded_value`. The consumer then issues the actual load. If the `use_forwarded_value` flag is set, the load is issued as a non-speculative load that only accesses the first-level cache, which maintains the speculative states. Since this instruction only accesses the first-level cache, it will *not* cause a dependence violation. This load also checks to see if the value has been overwritten locally and clears the `use_forwarded_value` flag if it has. The value of the `use_forwarded_value` flag determines whether the forwarded value or the value loaded from memory is used in subsequent computations. Once the decision is made, the `use_forwarded_value` flag is reset.

Figure 3.4 shows how correctness is ensured by enumerating all possible data dependences and describing how each case is handled. When a true data dependence occurs between `store *q` and `load *p`, the forwarding mechanism forwards the correct address and value, and the forwarded value is used by `load *p`. If `load *p` depends on `store *w` or `store *x`, the forwarded address `q` cannot point to the same location as `p`. Thus, the `use_forwarded_value` flag is not set by the `check` instruction, and the `select` instruction chooses the `memory_value` for subsequent computations. In this case, the forwarded value is not used, and the underlying hardware support for TLS ensures correct execution. If `p`, `q` and `y` all point to the same memory location, the forwarding instruction will forward the correct address, but an incorrect value. The producer thread will notice that it stores to an address that is already in the `signal_address_buffer` and send a signal to restart the consumer thread. If `load *p` depends on `store *z`, the `use_forwarded_value` flag is reset by the load instruction and the value loaded from the memory is used. This will not cause dependence violation, since this memory access is not exposed and the local cache holds the correct value.

It is possible that on some paths through a thread, the forwarded value is never produced. In this case, the producer thread should still signal the consumer thread by sending a `NULL` value in the address field so that the consumer does not wait indefinitely. If `p` points to a valid address, it will not match this `NULL` address, and the load in the consumer thread will be read from memory. If `p` happens to be a `NULL` pointer, the program will de-reference this `NULL` pointer as it is intended in the original program and cause an exception.

The size of the `signal_address_buffer` is equal to the number of forwarded values. In practice, this number is quite small and a buffer with 10-entries suffices in all the experiments.

3.5.2 Compiler Support

With the hardware support described in Section 3.5, the compiler is able to synchronize probable dependences without concerns about correctness. Using the example in Figure 3.5, this section describes how the compiler inserts synchronization. In the example, a loop, which calls the procedures *free_element()* and *use_element()* to add and remove members of a linked list called *free_list*, is to be parallelized. In every iteration of the loop, the global variable *free_list* is read and modified, potentially causing frequent data dependences and speculation failure unless prevented by proper synchronization. Note that this example is complicated by the fact that the variable *free_list* can be accessed using other names (i.e., *aliases*). The compiler performs the following steps to synchronize the accesses to this variable.

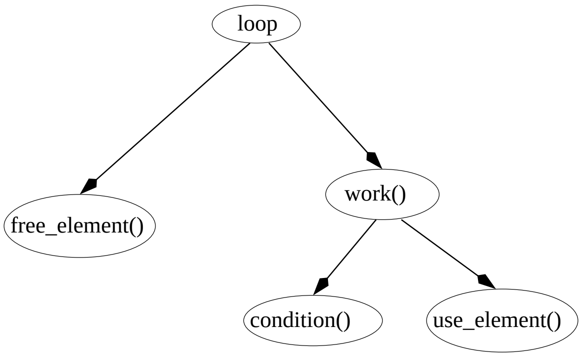
Profiling Dependences: The compiler identifies frequently occurring, memory-resident data dependences by profiling all inter-thread data dependences for every parallelized loop. (This profile information is context-sensitive but flow-insensitive.) To acquire the profile information, we first associate a unique identifier with each static load and store instruction and each procedure call

```
void  free_element(element) {
   element->next = free_list;            st_1, ld_1
   free_list = element;                  st_2, ld_2
}

int  use_element() {
   element = free_list;                  st_3, ld_3
   free_list = element->next;            st_4, ld_4
   return element;
}

void  work() {
   if(condition())                       call_1
      use_element(some_element);         call_2
}

main()  {
   do {
      free_element(some_element);        call_3
      work();                            call_4
   } while (test);
}
```

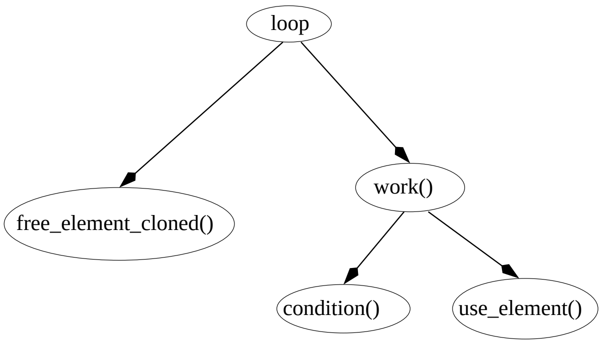


(a) The original program and the corresponding call tree. Function calls, loads and stores are instrumented with labels to identify them.

```
void free_element_cloned(element) {
   f_addr = wait();
   check(f_addr, &free_list);
   f_value = wait();
   m_value = free_list;
   actual_value = select(f_value, m_value);
   resume();
   element->next = actual_value;
   free_list = element;
   signal(&free_list);
   signal(free_list);
}

...free_element(), use_element() and work()
functions omitted for brevity...

main()  {
   do_parallel {
      free_element_clone(some_element);
      work();
   } while (test);
}
```



(b) The cloned call tree and the program with synchronization inserted.

Figure 3.5: Compiler-directed procedural cloning and synchronization insertion.

point. During execution, each load and store instruction can be named by a combination of the instruction identifier and the current *call stack*. (The call stack for an instruction, rooted at the parallelized loop, is the list of procedure calls invoked when that instruction is executed.) During profiling, each load is matched with any store on which it depends, and the frequency of each dependence is recorded. In Figure 3.5(a), *ld_1*, *ld_3*, *st_2* and *st_4* all access the same memory location denoted by *free_list*, and their dependence relation is illustrated in Figure 3.6. Note that two memory references with the same identification number but different call stacks are treated separately (i.e., represented by two different vertices in the graph). Detailed description of this profiling tool can be found in Appendix A.2.

Identifying Frequently Occurring Dependences: Unlike scalar values, the same memory-resident value can be accessed with multiple names through pointer aliasing. To overcome this difficulty, we collect loads and stores that frequently access the same memory location into *groups*. It is important to understand that a *group* is different from an alias set. An alias set of pointers is defined conservatively to be a set of pointers that *may* point to the same memory locations. In contrast, (i) pointers in a group will *definitely* access the same memory locations frequently, and (ii) pointers that access the same location might not be in the same group if the corresponding data dependences are infrequent.

The compiler chooses groups of pointers using the dependence profiling information described above to construct a dependence graph, where each load or store instruction with a different call stack is represented by a vertex, and each frequently occurring dependence is represented by an edge. In the resulting graph, each connected component represents a *group*, and all loads and stores belonging to the same group are then synchronized by the compiler as a single entity. Infrequently occurring dependences are ignored for performance reasons: synchronizing infrequently occurring data dependences in the graph could create large groups (as shown in Figure 3.6) and lead to over-synchronization and poor performance.

Cloning: For best performance, two instructions are only synchronized if they are frequently dependent. For example, when a load with a particular call stack is chosen for synchronization, the corresponding synchronization code would ideally be executed only when the load is reached on a path matching that call stack, i.e., the synchronization code should not be executed when the load is reached through a different call path.

The compiler uses the following steps to implement this code specialization, which basically clones the appropriate procedures on the call stack of a synchronized memory reference. First, a call tree with the parallelized loop as the root and each call instruction as a decedent of this loop is created, as shown in Figure 3.5(a). Second, the locations of all frequently occurring data dependences are identified on the call tree: for any node containing frequently occurring dependences, that node and its parents are cloned, and the original call instructions are modified to reflect this. In this example, the synchronized load and store occurs on the call stack *call_3*; hence the procedure *free_element* is cloned as shown in Figure 3.5(b). There is an upper bound on the total number of procedures that can be cloned for a given parallel loops; optimization is abandoned if this limit is exceeded.

Inserting Synchronization: *Wait* instructions are inserted before the synchronized load instructions, as shown in Figure 3.4(b). However, *signal* instructions cannot be inserted after every store instruction, since multiple store instructions belonging to the same *group* could occur on a single execution path. A signal

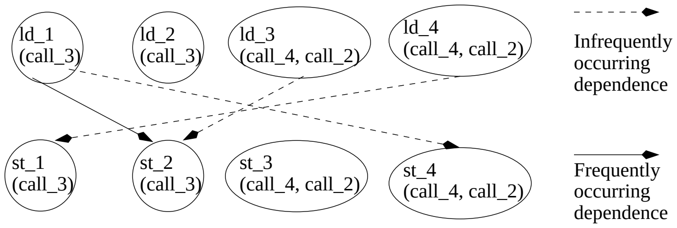


Figure 3.6: An example dependence graph. Each vertex represents a load or store, identified by the combination of a unique number and call stack. Each edge shows a true data dependence between memory references. Ignoring infrequent data dependences, a group is formed with two vertices: *ld_1* and *st_2* (both having call stack *(call_3)*). Accounting for infrequent data dependences would result in an overly large group.

instruction must be issued once *after* the last store instruction from that group has been issued for every group on every execution path through the thread. Thus, the same data-flow analyses for synchronizing scalar values can be applied by treating loads and stores belonging to the same group as accesses to a single scalar and treating each synchronized load/store instruction as a use/definition of that scalar, respectively. To deal with the fact that synchronization may be scheduled in callee procedures, results of these data-flow analyses are propagated through the call tree. $LocalDef$ information is collected by traversing the call tree in post-order: $LocalDef$s are first calculated at the callee procedures, and the union of $LocalDef$s in a callee procedure represents the $LocalDef$ of the calling instruction in the parent's control-flow graph. The NMD and the *signal* problem are solved through a pre-order traversal of the call tree. We first solve the data-flow analyses at the root node and then for each callee procedure using the subset of groups that need synchronization on its calling instruction in the parent.

3.6 Chapter Summary

This chapter describes the compiler and hardware support required for synchronizing register-resident and memory-resident values. In the case of register-resident values, the compiler is responsible for ensuring that all data dependences are preserved through explicit synchronization, since the underlying hardware does not track data dependences between register-resident values. Dataflow analyses are used to find appropriate program points to insert synchronization. In the case of memory-resident values, the compiler inserts synchronization to reduce speculation failures. The compiler uses profiling information to decide which load/store pairs to synchronize, and the hardware checks if the synchronized dependences really do occur at runtime and ensures recovery from incorrect synchronization.

Chapter 4

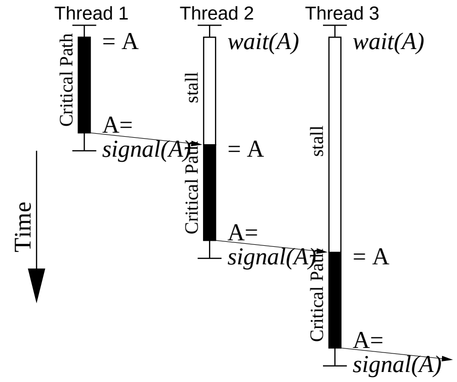
Instruction Scheduling

In Chapter 3, we described compiler-based techniques for inserting synchronization in speculatively parallelized programs. Such synchronization creates a point-to-point communication to forward values between speculative threads and reduces speculation failure. However, it may also serialize execution by stalling the consumer thread more than necessary. This chapter introduces instruction scheduling algorithms to address this problem.

4.1 Critical Forwarding Path

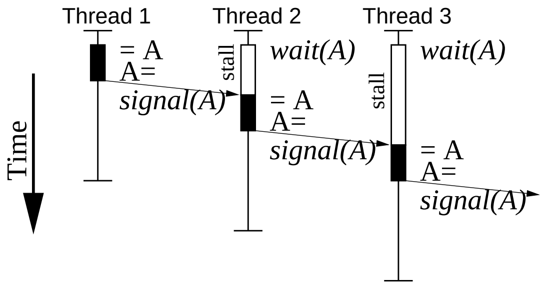
We will begin with an example. Figure 4.1(a) shows a loop the compiler has speculatively parallelized by partitioning the loop into speculative threads. The scalar `A` is not only register-resident, but also read and written in every iteration; thus, it has to be synchronized by the compiler for both correctness and performance. Synchronization is established by inserting a `wait` operation before the first *use* of `A`, and a `signal` operation after the last *definition* of `A`; the algorithm for inserting such synchronization is described in Section 3.4. The synchronization results in the partially parallel execution shown in Figure 4.1(a), where each thread stalls until the value of

```
do {
   wait(A);
   if (condition(A)) {
      A = A + *p;
   }
   else{
      B = 2;
      work2();
   }
   A = A + 1;
   signal(A);
} while (1);
```



(a) Before instruction scheduling.

```
do {
   wait(A);
   if (condition(A))  {
      A = A + *p + 1;
      signal(A);
   }
   else {
      A = A + 1;
      signal(A);
      B = 2;
      work2();
   }
} while (1);
```



(b) After instruction scheduling.

Figure 4.1: Impact of scheduling on the critical forwarding path.

`A` is produced by the previous thread. The flow of the value of `A` between threads serializes the parallel execution, and is referred to as the *critical forwarding path*. Although synchronization is better than speculation for a data dependence that occurs frequently, the resulting serialization can still limit performance. We have developed instruction scheduling techniques to address this issue.

4.1.1 Instruction Scheduling

The key to avoiding over-synchronization and improve performance is to reduce the critical forwarding path. What can the compiler do? The idea is to reduce the number of instructions between each `wait`/`signal` pair. However, this becomes more difficult in the presence of conditional control flows. Figure 4.1(b) shows the example loop after the compiler has scheduled the code to reduce the critical forwarding path. The scheduling algorithm has duplicated the computation of `A=A+1` as well as the `signal` and moved them into the `if-then-else` structure. If the condition on `A` is rarely true, then less work will be performed before reaching each `signal` (by deferring the computation of `B=2` and `work2()`). As shown in the figure, this reduces the stall time for each thread, thereby improving overall execution time. We have devised an algorithm for reducing the critical path, which is elaborated in Section 4.3.1.

4.1.2 Aggressive Instruction Scheduling

All of the transformations that have been described so far, including all synchronization insertion and instruction scheduling algorithms, preserve the control and data dependences within each thread: the transformed codes perform the same operations as the original ones, but possibly reordered within each control structure and between ambiguous data dependences. It may potentially be beneficial to move code across control and data dependences [6, 23, 27, 49] to further reduce the critical forwarding path. For example, if a certain path is executed more frequently than the other paths, then it is advantageous to speculatively schedule the critical forwarding path to exploit this fact. To illustrate, if the *else* clause is more frequently executed than the *then* clause in Figure 4.1(b), "`A=A+1;signal(A);`" could be moved from the *else* clause to before the *if* structure to further shrink the critical forwarding path in the common case. This new scheme requires the ability to recover whenever the speculation is incorrect. Similarly, codes from the critical forwarding path can be scheduled across ambiguous data dependences, given additional hardware support to detect when such speculation fails. This algorithm and the hardware support are both described in Section 4.3.2.

The rest of this chapter focuses on reducing the critical forwarding path created by synchronizing register-resident scalar values, although the techniques can also be applied to memory-resident values.

4.2 Related Work

Parallelizing loops with loop-carried data dependences is known as *DOACROSS* parallelization [21, 52], which has been exploited in previous works [8, 45, 74]. Almost all schemes for TLS support include some form of DOACROSS synchronization [15, 32, 39, 43, 47, 64, 67, 68, 72, 73, 75], although fewer use the compiler to optimize such synchronization [68, 72, 73, 75].

The most relevant work is the Wisconsin Multiscalar [25, 58, 68] compiler, which synchronizes register-resident values and schedules instructions for performance [68]. The Multiscalar scheduler was designed with Multiscalar tasks in mind, and these tasks usually consist of a few basic blocks that do not contain procedure calls or loops. In contrast, the speculative threads in this research are much larger on average than Multiscalar tasks and often contain more complex control flows. This inspires the dataflow-based scheduler presented in this chapter, which is able to move instructions across inner loops and procedure calls. The Multiscalar compiler does not schedule codes beyond the point within a task where it is no longer critical, which is determined by a simplified machine model. In contrast, we schedule producer instructions as early as possible, because accurately determining such points at compile-time is difficult.

Our work also evaluates the benefits of speculatively scheduling code past control and data dependences while Multiscalar does not (as discussed later in Section 4.3.2).

Other schemes for TLS hardware support provide the means to synchronize and forward values between speculative threads but do not use the compiler to optimize loop-induction variables or synchronize frequent dependences [1, 31, 32, 43], while others provide such support but do not schedule instructions to reduce the critical forwarding path [15, 67]. Research on TLS hardware support has shown the importance of the critical forwarding path and how the prediction of forwarded values may be used to increase parallelism [44, 64]; it also shows that hardware is ineffective at improving performance by scheduling the critical forwarding path [64]. Other hardware techniques for improving the efficiency of speculation include prediction of loads to memory, dynamic synchronization, and squashing of *silent stores* (which overwrite memory with the same value that is already there) [1, 15, 43, 64].

Concurrent with our work, Zilles and Sohi [75] recently proposed decomposing a program into speculative threads by having a master thread execute a *distilled* version of the program that orchestrates and predicts values for slave threads. In this scheme, values are pre-computed by the master thread and distributed to the slave threads (as opposed to being updated and forwarded between consecutive speculative threads). A potential advantage of this master/slave approach is that it effectively removes interprocessor communication from the critical forwarding path. The scheduling techniques presented later in this chapter could potentially be applied to the distilled code in the master thread.

Given two parallel threads, the producer and the consumer, the set of instructions that need to be scheduled backward in the producer is essentially a *slice* of the producer with respect to the forwarded value and the instruction that produces it [48, 56]. However, our algorithm must go one step further and decide where these instructions should be scheduled to so that the critical forwarding path is reduced.

Our algorithm for reducing the critical forwarding path builds upon previous dataflow approaches to code motion, namely *partial redundancy elimination* [37], path-sensitive dataflow analysis [34], and *hot paths* [2]. Relevant previous work on speculative code motion also includes trace scheduling [23] and superblock scheduling [6].

There has also been work on aggressive load/store reordering where the runtime check and recovery are performed entirely in software [49] or through hardware and software hybrid [27]. These transformations can be classified into three main types: control flow speculation, data dependence speculation, and value speculation. Control flow based optimizations typically involve hoisting operation across basic-blocks, introducing them onto paths on which they were not originally presented [7, 17, 23, 57]. Data dependences based optimizations are employed to hoist load operations above store operations that could potentially alias [27, 55]. Value-speculation [41, 44] relies on the fact that loaded values are frequently predictable, and it supplies the load instructions with values from a value predictor.

4.3 Instruction Scheduling Algorithms

Compilers can improve the performance of speculatively parallelized codes by using scheduling techniques to move the `signal` operations (and the codes that these operations depend upon) upward through the control flow graph to reduce the length of the critical forwarding path and to expose more parallelism. For example, closer examination of Figure 3.3 reveals that the forwarded value for variable `a` depends only on the result of a single addition. While the forwarding path between the `wait` and the `signal` shown in Figure 3.3 contains many instructions, only the following three instructions are really necessary to wait for, compute, and forward the new value of `a`:

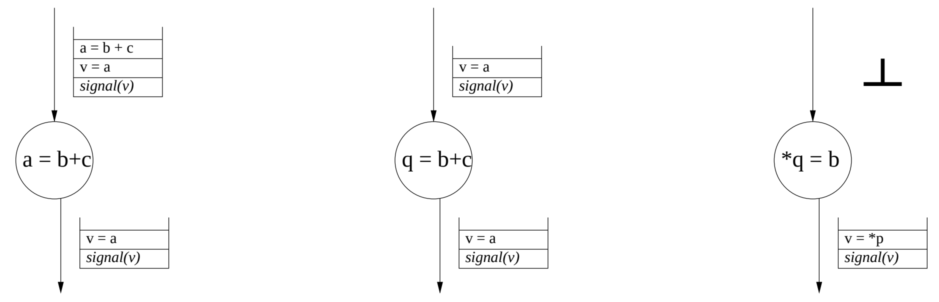
```
wait(a);
a = a + 1;
signal(a);
```

In the rest of this section, we describe a scheduling algorithm that achieves this minimum critical forwarding path for this example.

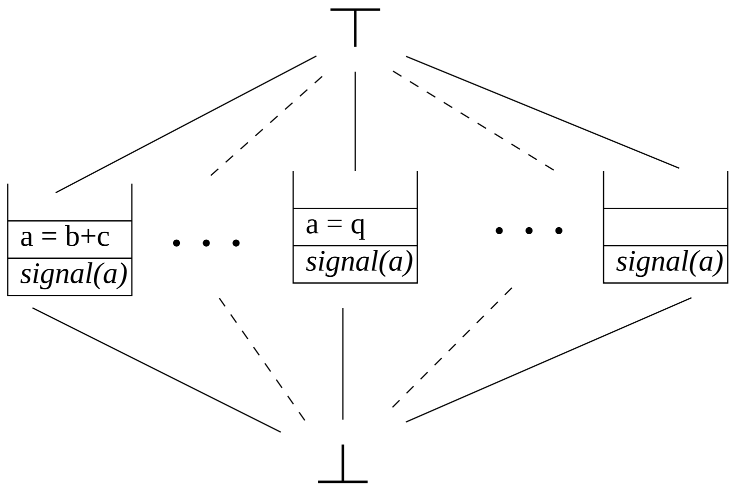
4.3.1 Conservative Scheduling

This section describes two dataflow analysis algorithms to insert synchronization for each forwarded value: (i) the *stack* analysis to determine how to compute the forwarded value, and (ii) the *earliest* analysis to determine where to insert the synchronization. Similar to the synchronization algorithms described in Section 3.4, the conservative scheduling algorithm is defined over the set of communicating scalars V on the control flow graph $G = (N, E, s, e)$, with critical edges broken by synthetic nodes as described in the Section 3.4.2. We initialize the algorithm by placing all `signal`s at the exit node e. (It is equivalent to start all `signal`s in the position indicated by the placement algorithm, but placing them at the end node simplifies the proof of correctness.) Although we only describe the algorithms for moving `signal` instructions (and the instructions they depend upon) earlier in the control flow graph, the converse of this algorithm can be applied to moving `wait` instructions (and the instructions that depend upon them) later in the control flow graph.

When scheduling the instructions, the computation that the eventual `signal` depends upon must be identified. Since it is difficult to represent a list of computations as binary values, bit-vector analysis is inadequate. Hence at each node n, a stack of computations—denoted as $stack(n, v)$—must be maintained for each communicat-



(a) Dependence exists. (b) No dependence. (c) Unresolved dependence.



(d) Lattice.

Figure 4.2: Illustration of the *transfer* function (parts (a), (b), and (c)) used for computing the value of the *stack* in equation (4.1) and the lattice (part (d)) over which the *stack* dataflow analysis is defined.

ing scalar. This stack records the computations necessary to produce the value of a communicating scalar v if it is to be sent from the corresponding node.

The domain of the *stack* dataflow problem is the set of all possible configurations of the computation stack. This domain, along with the meet operator (described later), defines a semi-lattice, shown in Figure 4.2(d). Before data-flow analysis begins, all nodes are initialized to $\top$. If a given node is found to be a safe place for the `signal` instruction, then *stack* returns a non-empty stack of computations; otherwise *stack* returns $\perp$. The following dataflow equation computes $stack(n, v)$ at the exit of each node:

$$stack(n, v) = \begin{cases} \boxed{\texttt{signal } v} & \text{if } n = e \\ \prod_{m \in succ(n)} transfer(m, v, stack(m, v)) & \text{otherwise} \end{cases} \tag{4.1}$$

where the $transfer$ function is defined as follows:

- If $stack(m, v) = \top$, then $transfer = \top$.

- If the computation chain for v in the stack $stack(m, v)$ depends on a value w produced by node m, then the computation that produces w is added to the computation stack, as illustrated in Figure 4.2(a).

- If the computation chain in the stack $stack(m, v)$ does not depend on a value produced by the computation at node m, then $transfer = stack(m, v)$, as illustrated in Figure 4.2(b).

- If the dependence between the computation chain for v and the computations in node m cannot be resolved statically by the compiler (e.g., due to ambiguous pointer references), the code motion should be stopped; hence $transfer = \perp$, as illustrated in Figure 4.2(c).

- If a `wait` is issued for any exposed scalar in the computation chain, the code motion should stop; hence $transfer = \perp$.

The meet operator $\prod$ for the *stack* problem is defined over the set of all possible configurations of the computation stack. The lattice shown in Figure 4.2(d) defines the following operations for meet: (i) the meet operator is communicative and associative; (ii) the meet operator returns $\perp$ if one of its operand is $\perp$; (iii) the meet operator returns $\perp$ if the two operands differ and neither is $\top$; (iv) the meet operator returns X if both operands equal to X; (v) the meet operator returns X if one operand is X and the other operand is $\top$. The meet operator combined with the domain of the *stack* function defines a semi-lattice of height three, thus this dataflow problem is well-defined.

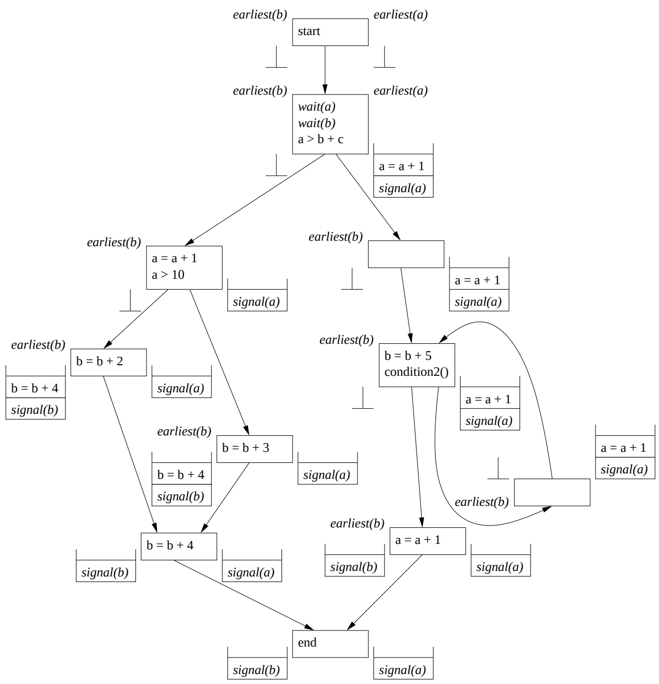
The *stack* analysis tells us only what computations are needed to compute forwarded values, we must now determine where to insert these computations. The dataflow problem, *earliest*, is formulated to find the synchronization point to compute the forwarded value as early as possible. *Earliest* is a bit-vector problem defined over the set of communicating scalars V on the control flow graph G. The $\mathit{earliest}(n, v)$ function is true at node n for v if no node prior to n is a safe place to schedule the `signal` on some execution path starting at s:

$$\mathit{earliest}(n,v) = \begin{cases} \mathit{true} & \text{if } n = s \\ \bigvee_{m \in \mathit{pred}(n)} (\neg \mathit{safe}(m,v) \wedge \mathit{earliest}(m,v)) & \text{otherwise} \end{cases} \tag{4.2}$$

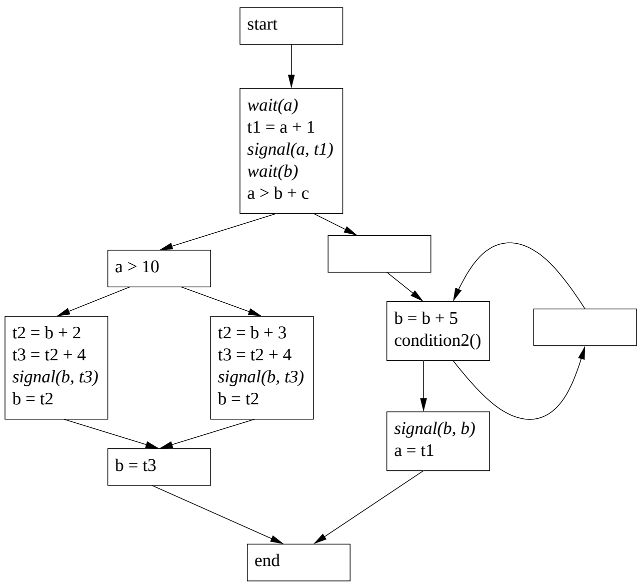
where $\mathit{safe}(m, v) = (\mathit{stack}(m, v) \neq \perp)$, and all nodes are initialized to false.

Code Transformation: For each node that is both *safe* and *earliest* for a variable v, the contents of v's stack are inserted either at the beginning of the node or immediately after the computation that stops code motion (a `wait` instruction or ambiguous pointer reference) if it exists. All references to v are replaced with the temporary variables, and all unscheduled computations are updated to use these temporaries.

Figure 4.3(a) illustrates solutions for *stack* and *earliest* for the example shown ear-



(a) Solutions to the *stack* and *earliest* problems.



(b) After code transformation.

Figure 4.3: Applying conservative scheduling algorithm to the codes in Figure 3.3.

lier in Figure 3.3. *Earliest* is true for variable a only at the entry node. The stack for the variable a at the entry node contains only the two instructions required to compute a—this matches the optimal result we derived manually at the beginning of this section. Figure 4.3(b) is the transformed program. Note that this transformation can either expand code size (by duplicating computations at join points) or reduce code size (by performing a form of common subexpression elimination at branch points). In all experiments, the code size is increased by less than 1.3% for all benchmarks.

Lemma 3 *If node n is safe, then the computations in the computation stack deliver the correct forwarding value.*

Proof: By induction on the length of the execution path from (e, n).

Lemma 4 *On every execution path from the start node s to the end node e, there is one and only one node that is both safe and earliest for v, the communicating scalar.*

Proof: *Existence:* Node e is *safe* for v by the definition of *safe*, and node s is *earliest* for v by the definition of *earliest*. Assume that no node is both *safe* and *earliest* for v on the path $[s, e]$, s is not *safe*, since s is *earliest* for v. Thus, its successor must be *earliest*. However, by the assumption, the successor must not be *safe* for v. By induction on the length of execution path, we can show that all nodes on the path to e, including e, are not *safe* for v. Hence, we have established a contradiction. Therefore, there is at least one node that is both *safe* and *earliest* on the path $[s, x]$.

Uniqueness: Consider a path $[s, e]$. We want to show that only one node on this path is both *safe* and *earliest*. Assume that there are two distinct nodes, n_i and n_j, that are both *safe* and *earliest* for v. Since n_i is *safe* for v, we know that all nodes m on any path between n_i and e are *safe* for v and that $\mathit{transfer}(m, v, \mathit{stack}(m, v))$ is also safe for v. Since n_j is *earliest*, the definition of *earliest* ensures that it has a predecessor q that is both $\neg$*safe* and *earliest*. And q is not on the path between n_i and n_j since n_i is safe for v. By the definition of the meet function, if q is $\neg$*safe*, it must have a successor r that is $\neg$*safe* for v or $\mathit{transfer}(r, v, \mathit{stack}(r, v))$ is not safe for v. However, since all critical edges are eliminated from the graph, q can only have one successor, n_j. But n_j is safe and that $\mathit{transfer}(n_j, v, \mathit{stack}(n_j, v))$ is also safe. A contradiction is established. Therefore, there is no node n_j, which is different from n_i, that is both *safe* and *earliest* if there already exists n_i that is both *safe* and *earliest*.

4.3.2 Aggressive Instruction Scheduling

In the scheduling algorithms described so far, the backward motion of `signal` operations (and the instructions on which they depend) can be obstructed for the following two reasons:

Control Dependences: If incompatible computation stacks from multiple execution paths meet at a single node during the backward dataflow analysis, then code motion stops. This implies that the conservative scheduling algorithm cannot move instructions out of the `then` or `else` parts of an `if-then-else` statement unless those same instructions are executed along both conditional paths.

Data Dependences: A computation stack cannot be moved across a store instruction whose target address may alias locations referenced in the computation stack. This scenario often arises when a load instruction is placed on the stack. Similarly, the code motion also stops at any call instruction in the absence of information on the side-effect of that call).

Consequently, instructions can only be issued at after all intra-thread control and data dependences are resolved and it is desirable to schedule more aggressively. In

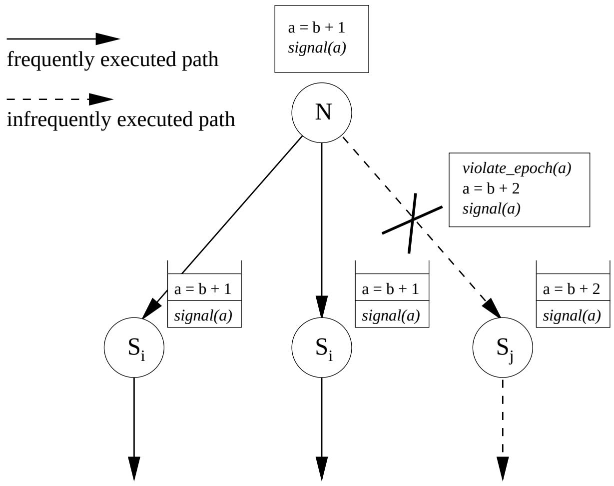


Figure 4.4: Modified meet operator for speculatively scheduling instructions across control dependence.

this section we will discuss both the compiler techniques and the hardware support necessary to allow for instruction scheduling beyond intra-thread control and data dependences.

Scheduling Across Control Dependences

Dataflow analyses conservatively assume that all execution paths are possible and find the minimal solution that satisfies all possible execution paths. In practice, however, only a small number of execution paths are frequently executed at runtime. By taking this into account, instructions can be scheduled more aggressively for the common cases at the cost of possibly incurring an expensive recovery operation on the less-frequently executed paths.

When the less-frequently executed paths are taken at runtime, the `signal` instructions could have forwarded incorrect values to the next thread; thus, a recovery mechanism is needed to ensure correct execution when this happens. To recover, a correct value is first forwarded to the consumer thread, then the consumer is notified of the invalid value. At this point, the consumer can either choose to restart the speculative execution immediately or check to see if the incorrect value has already been consumed and restart if so. Instructions speculatively scheduled across control dependences can cause exceptions that will otherwise not have occurred (e.g., NULL pointer checks). Thus, if exceptions occur in a speculative thread with instructions speculatively scheduled, speculation should fail and the thread should restart execution with a non-speculatively scheduled version of the code to ensure that the exception is real. Hence, some code duplication is necessary.

The scheduling algorithm from Section 4.3.1 can be modified to speculate on control dependences. The algorithm is made more aggressive by modifying the meet operator $\prod$ used in the *stack* dataflow analysis in equation (4.1), and add new nodes containing recovery codes on the infrequent edges. Frequently occurring execution paths are identified with the help of profiling information. (Detailed descriptions of the profiling tools can be found in Appendix A.1.) At each merge point, a decision on whether to speculatively schedule instructions across this branch must be made.

The meet operator $\prod$ for the *stack* dataflow analysis is modified as shown in Figure 4.4. At every node n, the meet operator first operates on successor s_i where (n, s_i) is on some frequently executed paths. Then for each node s_j, where (n, s_j) is not on any frequently executed path, the meet operator verifies whether $\mathit{transfer}(s_j, v, \mathit{stack}(s_j, v))$ is compatible with the partially evaluated $\mathit{stack}(n, v)$. If this verification fails, a new node is inserted on the edge (n, s_j), which contains a single `violate_thread` instruction. A minor change is also made to the definition of *earliest*: $\mathit{earliest}(s_j)$ is set to true for all newly inserted `violate_thread` nodes. By doing so, the instruction scheduling algorithm can automatically regenerate `signal` instructions on all execution paths starting from (s_j). Figure 4.4 illustrates how the two compatible computations on the frequently executed nodes are scheduled above node `N`, while a `violate_thread` mode is inserted on the infrequently executed path on the right.

It makes sense to speculate on a branch if it is biased, although not all biased branches should be speculated on. For instance, in the example shown in Figure 4.5, assuming the outer loop is the speculatively parallelized loop, it is not desirable to speculatively schedule instructions across `branch #2` even though it is an extremely biased branch. Since speculatively scheduling instructions across this branch will insert a *violate_thread* instruction on the exiting edge of the inner loop, this edge is taken on every execution path before the thread is terminated. Thus, more information is needed to decide whether to speculate across a certain branch. Our algorithm only speculates on a branch if the less-frequently taken path is infrequent relative to the total number of threads. The largest benefit of speculating across control dependences comes from speculating across branches typified by `branch #1` in Figure 4.5. Without speculation instruction scheduling, the signal for `modified` can only occur at *node 6*, after the inner loop has been completed. With speculative instruction scheduling, the signal for `modified` can be scheduled at *node 1*, before entering the inner loop. As a result, parallel overlap is increased significantly.

Scheduling Across Data Dependences

This section considers how the conservative scheduling algorithm can be extended to allow code motion beyond potential data dependences. Using the output from an automatic data dependence profiling tool (Detailed descriptions of the profiling tools can be found in Appendix A.2.), the compiler can reason about the likelihood of a data dependence. (The compiler can also reason about the likelihood of a data dependence using other information, such as, the results of pointer alias analysis.) If a data dependence is involved in generating a particular `signal` operation but this dependence is unlikely to occur at runtime, we can choose not to add this dependence onto the computation stack. At runtime, we must check if this data dependence is violated and perform recovery operation if it does. Two new instructions, ``‘‘mark_load’’`` and

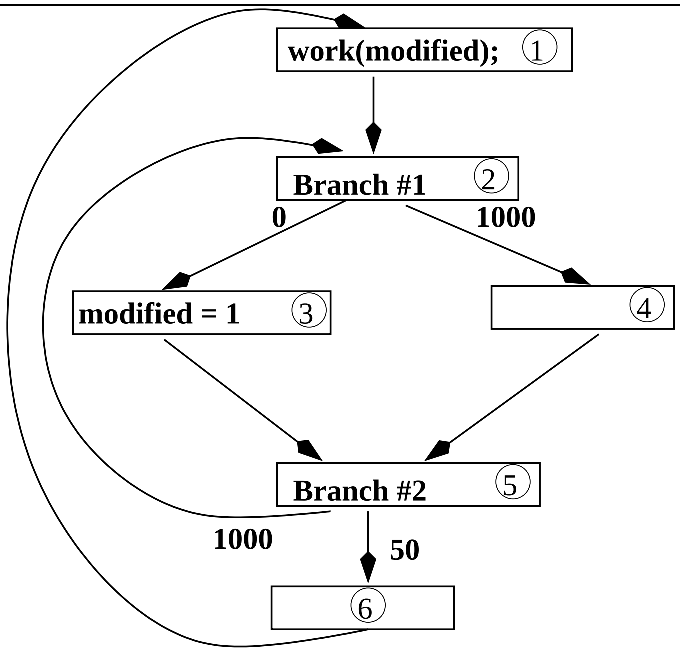
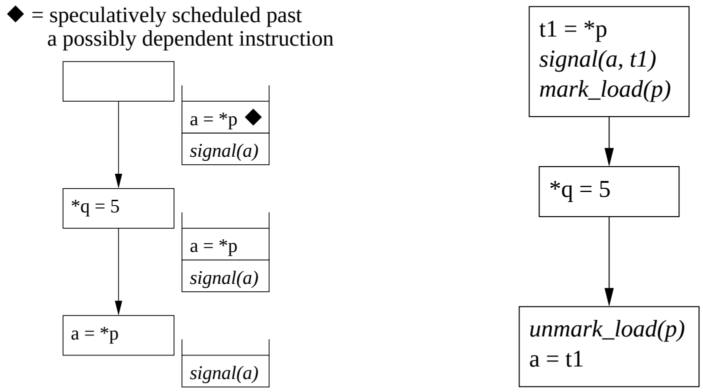


Figure 4.5: Control dependence speculation for regions with an inner loop, and each node is labelled with a unique identification. Edges are labeled with number of times that edge is followed during execution. Both `branch #1` and `branch #2` are biased branches.



(a) Modified transfer function for data speculation.

(b) Code generation for data speculation.

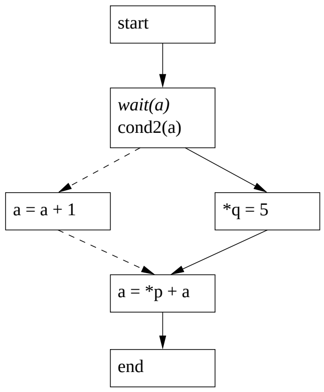
Figure 4.6: Modified dataflow analysis for speculatively scheduling instructions across data dependence.

``unmark_load'', are used for detecting data dependence violations: the `mark_load` instruction instructs the hardware to remember the speculatively loaded memory location. If any subsequent store modifies a marked location, the speculation fails. Once the potential data dependence is resolved, the `unmark_load` clears the mark at that memory location. If a speculation fails or when an exception occurs, the *recovery* action is invoked—i.e., the current thread will be violated—so that when the thread restarts, it runs a different version of the thread where no instruction is speculatively scheduled. It is worth noting that this architectural support for speculative loads is quite similar to the `LD.A` and `CHK.A` instructions [27] implemented in the Intel IA-64 architecture. One important difference, however, is that when the speculative code motion fails in this case, the underlying TLS recovery mechanism rewinds execution to the start of the thread; in contrast, under IA-64 the results of an `LD.A` instruction must be explicitly validated by a `CHK.A` instruction.

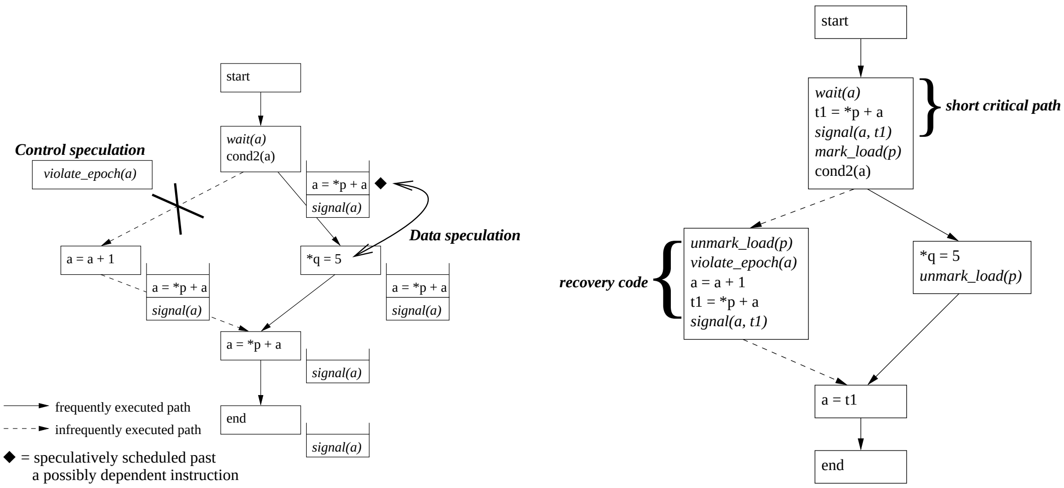
To implement scheduling across potential data dependences, the *transfer* function

```
while (cond1()) {
  if (cond2(a))
    a = a + 1;
  else
    *q = 5;
  a = *p + a;
}
```

(a) Original code.



(b) Control-flow graph for the original code.



(c) Control flow graph.

(d) Transformed code.

Figure 4.7: Illustration of how speculation on control and data dependences can be complementary.

described earlier in Section 4.3.1 (and used in equation (4.1)) is modified as shown in Figure 4.6(a). When scheduling a stack of instructions across a potentially dependent store, all potentially conflicting loads are marked in the stack as being *possibly conflicting*. When two stacks are merged at node n through the meet operator $\prod$, any *possibly conflicting* marks are merged using logical *OR*. At the time of code generation, a `mark_load` instruction is added after each *possibly conflicting* load. For all load instructions that are marked as *possibly conflicting*, an `unmark_load` is inserted at the original location of the load instruction.

Complementary Effects

Control dependence speculation and data dependence speculation can be complementary. Figure 4.7 shows an example where the combination of a control dependence and a data dependence prevents the code from being scheduled early, and where speculation on either type of dependence alone will not yield any benefit. By speculating on both control and data dependences in tandem, the computation of variable `a` can be moved backwards next to the `wait` operation, thereby resulting in a much shorter critical forwarding path.

4.4 Chapter Summary

When the compiler inserts wait/signal instructions to synchronize speculative threads, it can potentially create critical forwarding paths that stall the consumer thread and serialize execution. This chapter describes a dataflow analysis algorithm that moves the *signal* operation and the instructions that are dependent on it backward in the control flow to reduce the critical forwarding path. Such backward code motion is sometimes stopped by intra-thread control and data dependences, and therefore we have developed aggressive instruction scheduling algorithms to move instructions across them to reduce the critical forwarding paths even further.

Chapter 5

Performance Evaluation

This chapter presents the results of a detailed simulation designed to evaluate the effectiveness of the value communication optimization algorithms described in the previous chapters. It begins with a description of the simulation framework and the characteristics of all targeted benchmarks, then thoroughly examines the performance impact of various compiler optimizations on all loops in all benchmarks, from SPEC95 integer, SPEC2000 integer and SPEC2000 floating point benchmark suits, to establish a set of region selection criteria and design parameters. Once the region selection criteria and design parameters are established, several sets of regions are selected for parallelization. The performance impact of all compiler optimizations on the selected regions and the entire programs is carefully evaluated and discussed. Finally, the major findings of these experiments are summarized at the end of this chapter.

5.1 Simulation Framework

The evaluations presented in this chapter are performed using a detailed machine model that simulates 4-way issue, out-of-order, superscalar processors similar to the MIPS R10000 [71], but modernized to have a 128-entry reorder buffer. Each proces-

Table 5.1: Simulation parameters.

Pipeline Parameters	
Issue Width	4
Functional Units	2 Int, 2 FP, 1 Mem, 1 Branch
Reorder Buffer Size	128
Integer Multiply	12 cycles
Integer Divide	76 cycles
All Other Integer	1 cycle
FP Divide	15 cycles
FP Square Root	20 cycles
All Other FP	2 cycles
Branch Prediction	GShare (16KB, 8 history bits)
Memory Parameters	
Cache Line Size	32B
Instruction Cache	32KB, 4-way set-assoc
Data Cache	32KB, 2-way set-assoc, 2 banks
Unified Secondary Cache	2MB, 4-way set-assoc, 4 banks
Miss Handlers	16 for data, 2 for insts
Crossbar Interconnect	8B per cycle per bank
Minimum Miss Latency to Secondary Cache	10 cycles
Minimum Miss Latency to Local Memory	75 cycles
Main Memory Bandwidth	1 access per 20 cycles

sor has its own physically private data and instruction caches connected to a unified second-level cache by a crossbar switch. Register renaming, the reorder buffer, branch prediction, instruction fetching, branching penalties, and the memory hierarchy (including bandwidth and contention) are all modeled and parameterized, as shown in Table 5.1.

5.2 Benchmark Characteristics

This section describes and analyzes the benchmark applications that are used to generate the results of this research. All benchmarks in the SPECint95, SPECint2000, and SPECfp2000 benchmark suites [59] are evaluated, with the following exceptions: 252.EON, which is written in C++ and not supported by SUIF; 178.GALGEL, 187.FACEREC, 189.LUCAS, and 191.FMA3D, which are written in Fortran90 and not supported by SUIF; 168.WUPWISE, 173.APPLU, 200.SIXTRACK, and 301.APSI, for which SUIF compilation fails; 126.GCC, which is similar to 176.GCC; 147.VORTEX, which is identical to 255.VORTEX; and 129.COMPRESS, 130.LI, 134.PERL, 186.CRAFTY, and 255.VORTEX which lack the loops that both comprise an interesting portion of execution and is able to speed up with an oracle that perfectly predicts all values that cause inter-thread data dependences. Two compression/decompression benchmarks from SPECInt2000, 164.GZIP and 254.BZIP2, are divided into two phases, compression and decompression, and are evaluated separately since the two phases behave very differently. For 175.VPR, only the placing phase of the benchmark is evaluated due to compilation errors in evaluating the routing phase. A brief description of each remaining benchmark is given in Table 5.2.

5.2.1 Benchmark Input

For each benchmark, the *ref* input set is used to measure the performance while both the *train* and the *ref* input sets are used to collect profiling information. In the case of multiple input files, only the first input set is chosen. To ensure a reasonable speed for profiling and simulation, the initialization portion of execution for all appropriate benchmarks has been skipped and simulation begins with a "warmed-up" memory system loaded from a pre-saved snapshot. Only a maximum of roughly one billion instructions are simulated for each benchmark. Since the sequential and TLS versions

Table 5.2: Benchmark descriptions.

	Benchmark	Description
SPECint95	099.GO	game playing
	124.M88KSIM	microprocessor simulator
	132.IJPEG	image processing
SPECInt2000	164.GZIP	compression/decompression
	175.VPR	FPGA placing and routing
	176.GCC	compiler
	181.MCF	combinatorial optimization
	197.PARSER	natural language parsing
	253.PERLBMK	perl interpreter
	254.GAP	group theory interpreter
	256.BZIP2	compression/decompression
	300.TWOLF	placing and routing
SPECfp2000	171.SWIM	weather simulation
	172.MGRID	computational fluid dynamics multigrid solver
	177.MESA	an OpenGL 3-D graphics library
	179.ART	thermal image recognition with a neural network
	183.EQUAKE	earthquake simulation with an unstructured mesh
	188.AMMP	models molecular dynamics

Table 5.3: Truncation of benchmark execution.

		Simulation Input	Profiling Input	Cycles Simulated	Instruction Simulated
SPECint95	099.GO	ref	train	709M	1003M
	124.M88KSIM	ref	train	617M	1000M
	132.IJPEG	subset of ref	train	350M	649M
SPECInt2000	164.GZIP_COMP	subset of ref	train	549M	1000M
	164.GZIP_DECOMP	subset of ref	train	616M	991M
	175.VPR_PLACE	ref, place phase	train, place phase	656M	949M
	176.GCC	subset of ref	train	746M	1010M
	181.MCF	ref	train	3311M	1065M
	197.PARSER	ref	train	874M	980M
	253.PERLBMK	subset of ref	subset of train	764M	1058M
	254.GAP	ref	train	535M	1010M
	256.BZIP2_COMP	ref	train	606M	962M
	256.BZIP2_DECOMP	ref	train	674M	987M
	300.TWOLF	ref	train	1035M	987M
SPECfp2000	171.SWIM	ref	train	744M	815M
	172.MGRID	ref	train	602M	872M
	177.MESA	ref	train	553M	992M
	179.ART	ref	train	974M	1016M
	183.EQUAKE	ref	train	527M	1019M
	188.AMMP	ref	train	1655M	1004M

of each benchmark are compiled differently, the compiler instruments them to ensure that they terminate at the same point in their executions relative to the source codes so that the executions are comparable. Table 5.3 shows the inputs used both for profiling and for simulation, as well as the number of cycles and instructions simulated for each benchmark.

5.3 Estimating the Performance Upper Bounds for Value Communication Optimizations on All Loops

This section evaluates the importance of value communication optimizations by examining the performance upper bounds of all loops in all benchmarks when different value communication oracles are applied. We first outline the details of our evaluation methodology, then present the potential performance benefit for all loops in all benchmarks when the communication of register-resident and memory-resident values has been optimized.

5.3.1 Evaluation Methodology

For all evaluations in this section, value communication optimizations are not actually implemented in the compiler, but rather estimated by implementing various value prediction oracles in the simulator. This allows us to establish the performance upper bound for each optimization since the best possible way for reducing the cost of communicating a value is to use a perfect value predictor that always provides the desired value on time. Each value communication optimization is implemented with a different predictor in an oracle simulator to emulate the effects of that optimization. The speedups achieved by these oracle simulators are compared.

This study is conducted by following the process shown in Figure 5.1. Each loop is compiled twice to create a sequential executable and a parallel executable. In the parallel executable, all register-resident values are communicated by synchronizing them at their first use and last definition while all memory-resident values are communicated by speculation. Each loop is executed multiple times—first sequentially, then in parallel with various value communication oracles—to obtain a speedup for every loop and every value communication optimization. To take loop nesting into consideration, all the loops in the same benchmark are partitioned into *loop sets*, where loops belonging to the same set are not nested during executions, and hence can be parallelized together without interference. To compare two optimizations, **A** and **B**, three distinct execution times are extracted for every loop: (i) sequential execution time, t_{seq}; (ii) parallel execution time when optimization **A** is applied, t_A; (iii) parallel execution time when optimization **B** is applied, t_B. For each loop, the parallel speedups achieved, with the two optimizations individually applied, are calculated relative to sequential execution: $Speedup_A = t_{seq}/t_A$ and $Speedup_B = t_{seq}/t_B$.

To visually compare the effects of the two optimizations, the results are plotted on a two-dimensional graph, as shown in Figure 5.2. Every parallelized loop is represented as one data point in this graph, where its x-coordinate is the speedup for optimization **A** and its y-coordinate is the speedup for optimization **B**. A 45-degree reference line is also plotted on the graph. Points above the 45 degree reference line correspond to loops that perform better when optimization **B** is applied; points below the reference line correspond to loops that perform better when optimization **A** is applied; and points on the reference line correspond to loops that perform equally well under the two optimizations. Points with x- or y-coordinates greater than one represent loops that speed up relative to the sequential execution. Loops with different coverage[1], are differentiated on the graph using different symbols—loops with less

[1]Coverage is the fraction of execution time spent executing instructions from the parallelized loops in the original sequential execution.

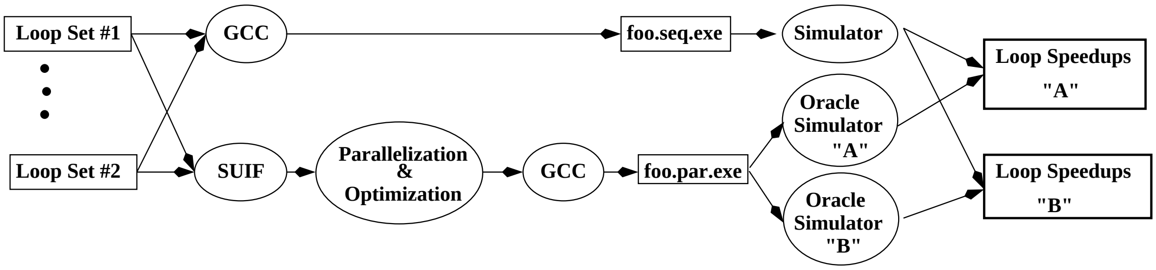


Figure 5.1: Compilation and simulation framework for studying performance potential of different optimizations.

than 5% coverage are represented by dots, and loops with greater than 5% coverage are represented by circles. In the rest of this evaluation, a separate graph is generated for each benchmark.

5.3.2 Reducing Critical Forwarding Path for Register-Resident Values

The impact of reducing the critical forwarding path for register-resident values is shown by the results in Figure 5.3. The best possible inter-thread value communication for register-resident values is a perfect value predictor that always provides the consumer with the correct values and never results in synchronization stalls. This is the oracle implemented in this experiment. In this figure, the x-axis is the speedup due to TLS for each loop when no optimization is applied, and the y-axis is the speedup when register-resident values are perfectly predicted and thus cause no stall. Across all benchmarks, a large number of loops are located above the 45-degree reference line, indicating that optimizing register-resident value communication has a significant impact on them. It also shows that without reducing the critical forwarding path, few loops are able to speedup relative to sequential execution.

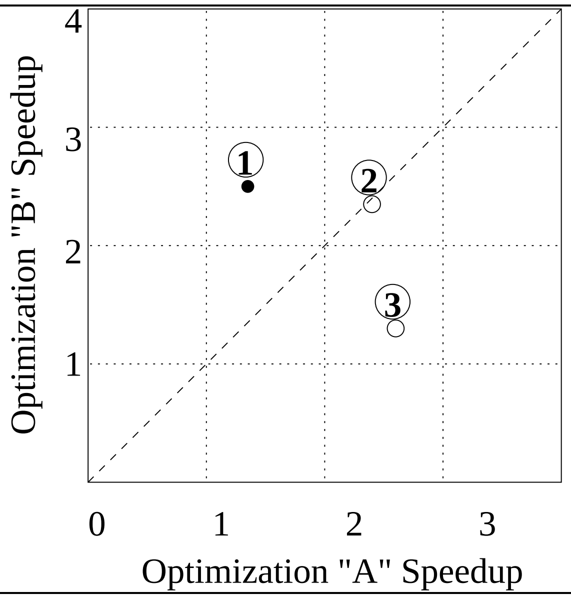


Figure 5.2: Comparing the impact of optimization **A** vs. optimization **B**. Each data point in the graph represents a loop. Its x-coordinate is the loop's speedup when optimization **A** is applied, and its y-coordinate is the loop's speedup when optimization **B** is applied. ① represents a loop with less than 5% coverage that performs better with optimization **B**; ② represents a loop with more than 5% coverage that performs equally well with both optimizations; and ③ represents a loop with more than 5% coverage that performs better with optimization **A**.

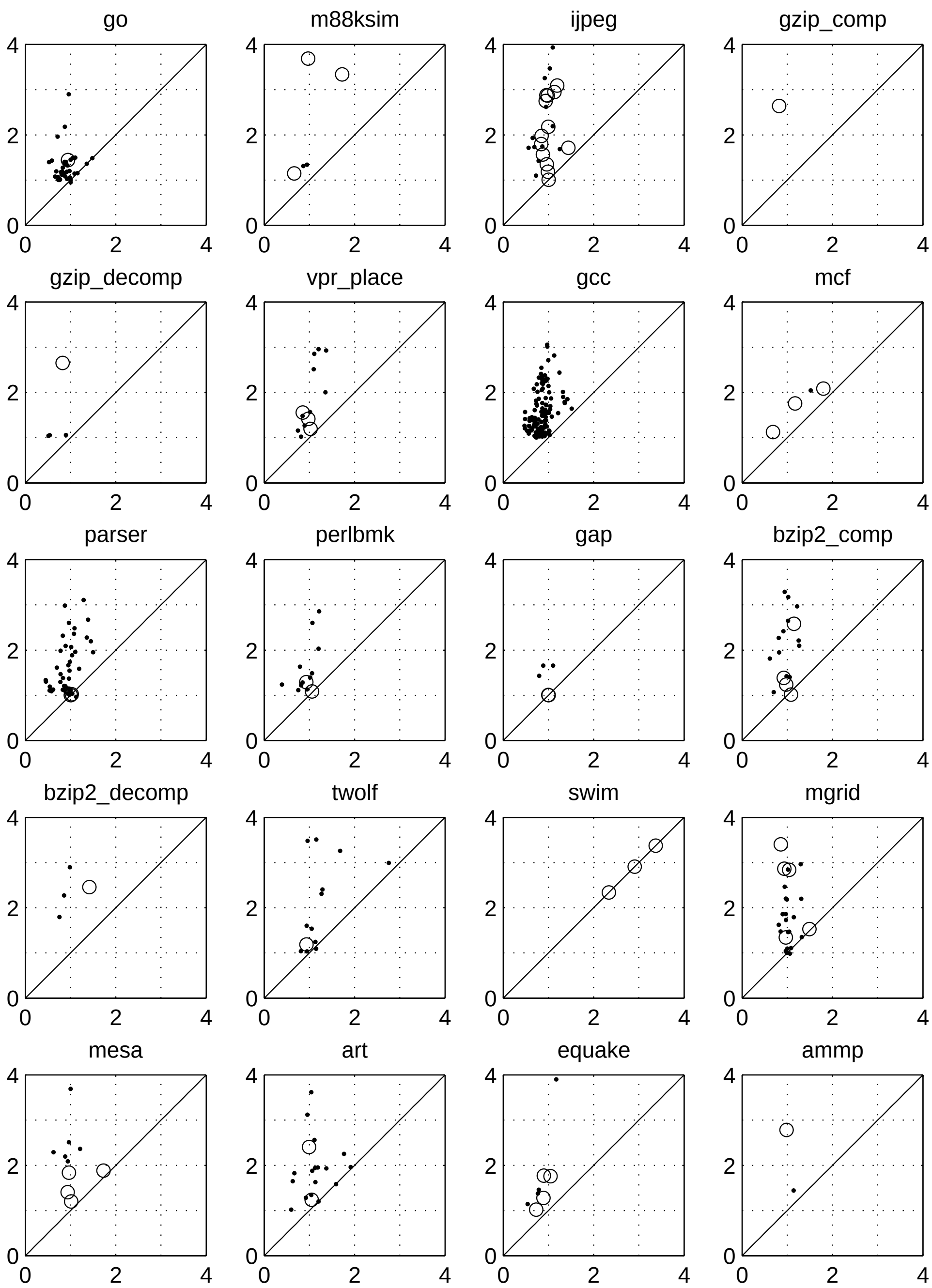


Figure 5.3: **Impact of reducing the critical forwarding path for register-resident values.** In each graph, the x-axis is the speedup with no value predictor, and the y-axis is the speedup with a prefect value predictor for all register-resident values. Loops that do not speed up in both cases are omitted from the graph for clarity.

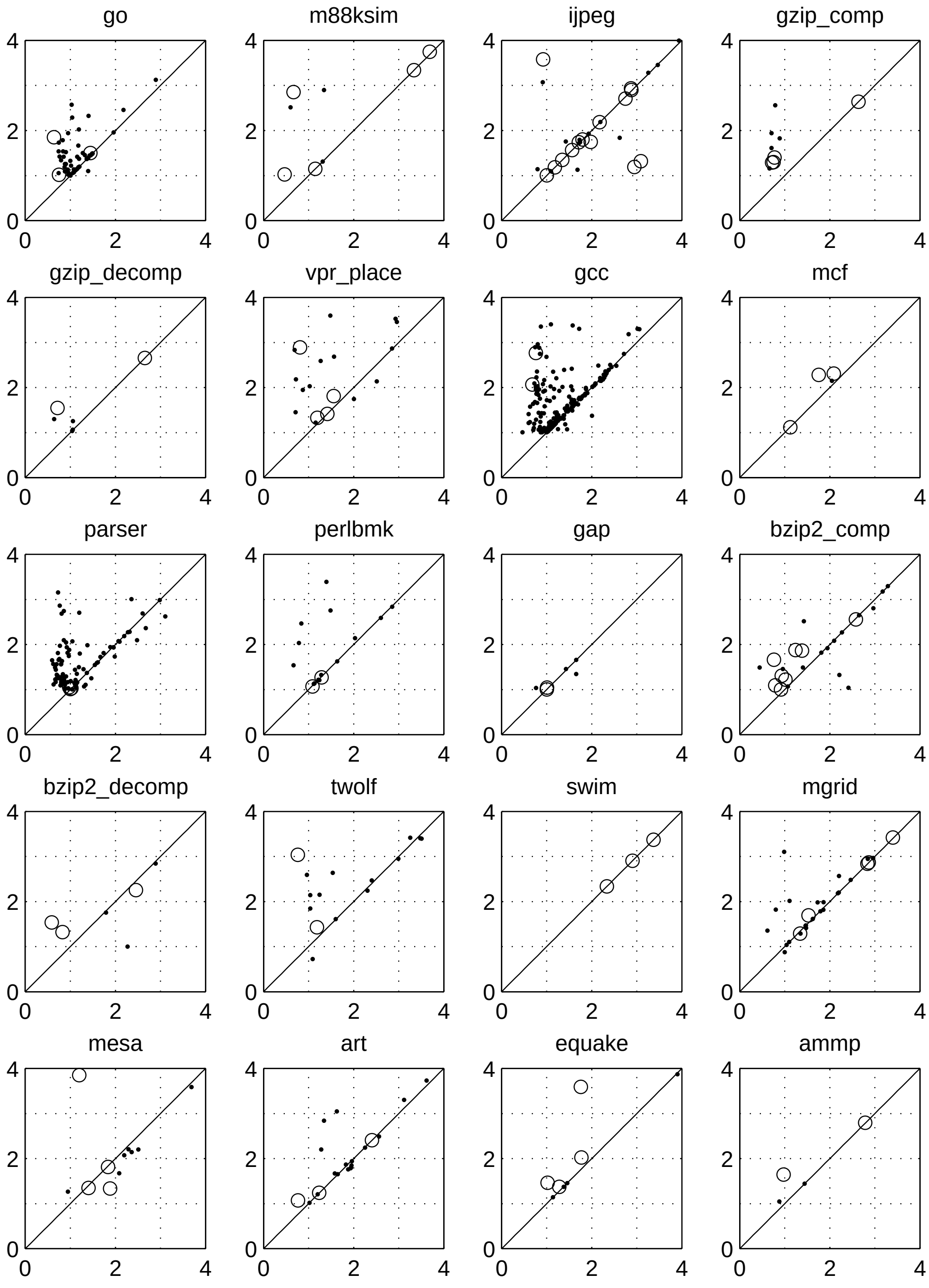


Figure 5.4: **Impact of avoiding speculation failures for memory-resident values.** In each graph, the x-axis is the speedup with a prefect value predictor for all register-resident values, and the y-axis is the speedup with a perfect value predictor for both register-resident and memory-resident values. Loops that do not speed up in both cases are omitted from the graph for clarity.

5.3.3 Avoiding Speculation Failures for Memory-Resident Values

The impact of avoiding speculation failures for memory-resident values is illustrated by the results shown in Figure 5.4. The best possible inter-thread value communication for memory-resident values is a perfect value predictor that always provides the consumer with the correct value without any speculation failure, and this is the oracle implemented in this experiment. In this figure, the x-axis is the speedup when only register-resident values are perfectly predicted, and the y-axis is the speedup when both register-resident values and memory-resident values are predicted, thus causing neither stalls nor speculation failures. Across all benchmarks, a large number of loops are located above the 45-degree reference line, indicating that optimizing memory-resident value communication has a significant impact for a large number of loops.

5.4 Loop Selection

Deciding on which loops to parallelize and which value communication optimizations to apply are two inter-dependent problems. The loops selected for parallelization affect the effectiveness of the value communication optimization passes, while the value communication optimization passes enable more loops to speed up under TLS. Therefore, the loop selection criteria should be adjusted according to the availability of the value communication optimization passes. This circular dependence is broken by providing the loop selection algorithm with an estimation of the performance impact of the optimization passes, as shown in Figure 2.12. Such an estimation can be obtained by executing an unoptimized parallel executable on an oracle simulator that emulates the effects of the compiler optimization passes. For instance, if the compiler optimization pass is able to optimize inter-thread value communication for all register-resident values, this effect can be emulated with an oracle simulator that perfectly predicts all register-resident values. The performance achieved by this oracle simulator is compared with the sequential execution time to obtain a speedup that reflects the performance upper bound achievable by this optimization. This region speedup is used in the loop selection algorithm described in Section 2.4.1. This entire process is illustrated in Figure 5.5.

Three sets of loops, requiring increasingly more aggressive inter-thread value communication optimizations, are selected for performance evaluation,

Register set is the set of loops that maximize program performance when the costs of communicating *all* register-resident values are completely eliminated (a.k.a. the register set);

Realistic set is the set of loops that maximize program performance when the costs of communicating *all* register-resident values and *frequently dependent* memory-resident values are completely eliminated (a.k.a. the realistic set). A data dependence is considered frequently occurring if it occurs in more than 4% of all threads, Appendix C.5 describes how this threshold is established.

Idealistic set is the set of loops that maximize program performance when the costs of communicating *all* values are completely eliminated (a.k.a. the idealistic set);

As more aggressive value communication optimizations are implemented, the coverage for each region set also increases, as shown in Table 5.4.[2] For most benchmarks, the coverage of the realistic set is larger than or equal to that of the register set, and the coverage of the idealistic set is larger than or equal to that of the realistic set.

[2]With hardware support for speculative execution, we are able to parallelize a large set of loops that are previouly not parallelizable. Detailed discussion can be found in Appendix D.

Table 5.4: Fraction of execution being parallelized.

Benchmark	**Register Set**	**Idealistic Set**	**Realistic Set**
099.go	17.9%	93.0%	24.5%
124.m88ksim	6.1%	97.9%	13.5%
132.ijpeg	84.5%	84.5%	97.6%
164.gzip_comp	10.3%	99.9%	47.3%
164.gzip_decomp	32.0%	99.6%	99.6%
175.vpr_place	72.6%	99.7%	73.1%
176.gcc	26.6%	88.4%	34.9%
181.mcf	96.8%	96.8%	96.8%
197.parser	56.7%	89.8%	88.5%
253.perlbmk	20.6%	18.9%	18.9%
254.gap	51.9%	94.0%	53.1%
256.bzip2_comp	68.2%	70.4%	70.5%
256.bzip2_decomp	13.5%	99.7%	100.0%
300.twolf	7.3%	100.0%	100.0%
171.swim	99.8%	99.9%	99.9%
172.mgrid	99.1%	88.8%	99.4%
177.mesa	87.3%	99.0%	99.0%
179.art	46.3%	100.0%	100.0%
183.equake	100.0%	100.0%	100.0%
188.ammp	6.8%	94.8%	94.8%

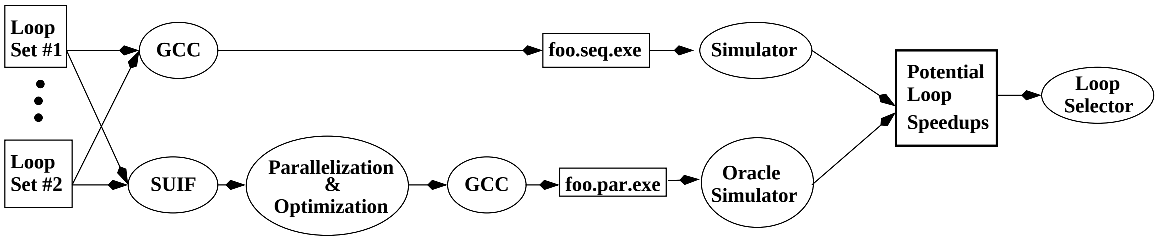


Figure 5.5: Region Selection Process

5.5 Evaluating Value Communication Optimizations

All performance evaluations are conducted on the three sets of loops that are selected to maximize program performance under different optimizations, as described in Section 5.4. A four-processor chip multiprocessor that supports TLS (detailed in Chapter 2) is implemented in the simulator. All figure shows the speedups or the execution time of the benchmarks normalized to the execution time of the original sequential executable (i.e. without any TLS instructions or overheads) running on a single processor. Hence our speedups are *absolute speedups* and not self-relative speedups. Every bar in Figures 5.6 and Figure 5.8-5.17 is broken down into four segments explaining what happens during all potential *graduation slots*. The number of graduation slots is the product of (i) the issue width (4 in this case), (ii) the number of cycles, and (iii) the number of processors (4 in this case). The *fail* segment represents all slots wasted on failed thread-level speculation, and the remaining three segments represent slots spent on successful speculation. The *busy* segment is the number of slots where instructions graduate; the *sync* segment represents slots spent waiting for synchronization for scalar values (scalar values are communicated using explicit synchronization); and the *other* segment is all other slots where instructions cannot graduate. Both the *sync* segment and the *fail* segment represent the amount

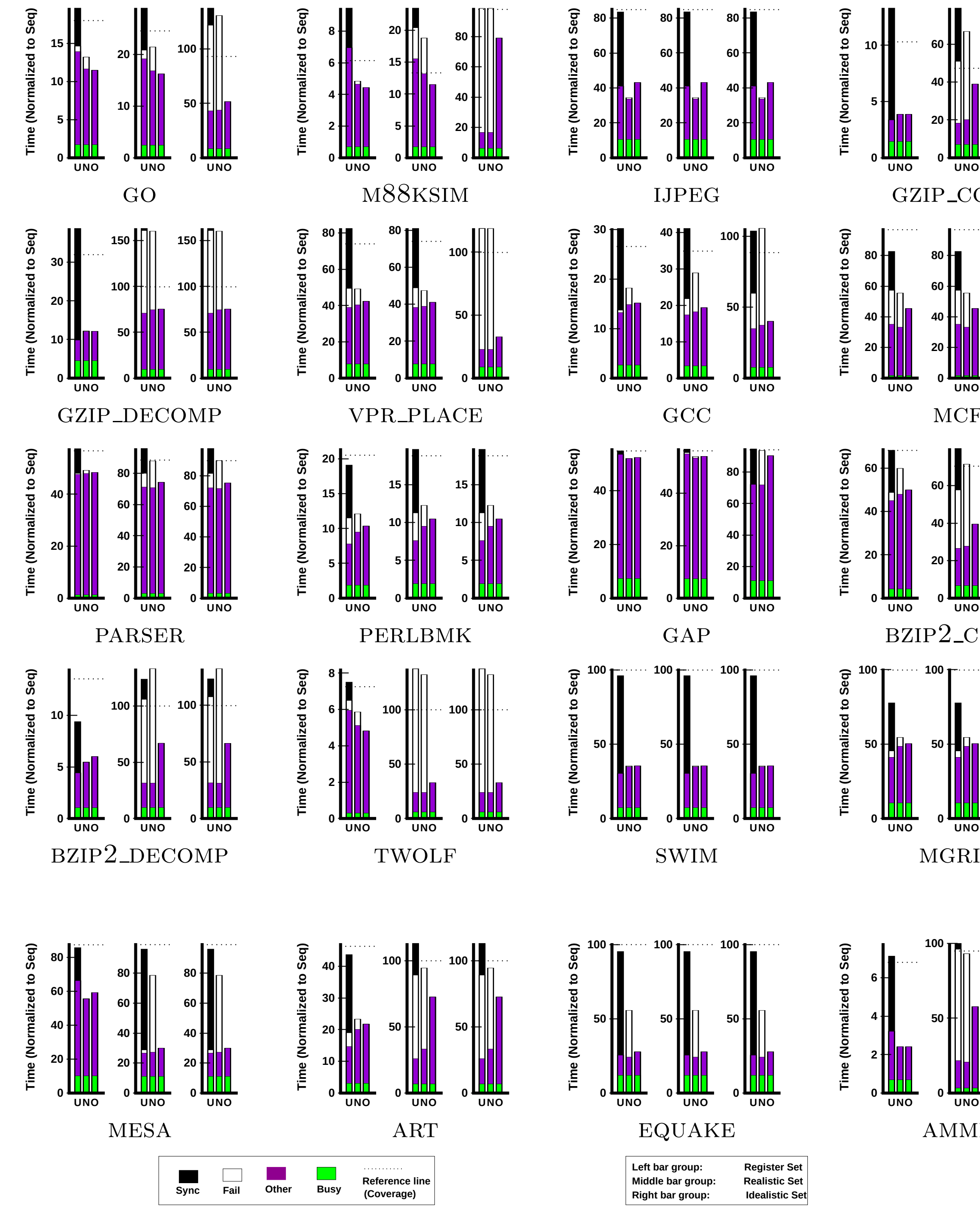


Figure 5.6: **Potential impact of optimizing inter-thread value communication.** For each benchmark, three sets of results are presented, corresponding to the register, the realistic, and the idealistic loop sets, respectively. Bars represent execution time of the parallel loops on a four-processor CMP normalized to the sequential program execution time. **U** is unoptimized, all register-resident values are synchronized and all memory-resident values are speculated; **N** assumes a perfect value predictor for all register-resident values; and **O** assumes a perfect value predictor for all values.

of time the application spends communicating values. For each figure, a reference line (the dotted horizontal line) is drawn corresponding to the sequential execution time of the parallelized loops, i.e., the coverage of the parallelized loops. Hence, the bars below the reference line represent benchmarks that speed up when parallelized. In Figures 5.6-5.12, we show three sets of performance corresponding to the register set, the realistic set, and the idealistic set, respectively. (The selection criteria of these loop sets were described earlier in Section 5.4.)

Figure 5.6 shows the speedup potential for inter-thread value communication for each set of loops. We begin with an un-optimized executable, in which all register-resident values are communicated by inserting synchronization at their first uses and last definitions, while all memory-resident values are communicated through speculation. The bars labeled with **U** show that all benchmarks without optimization spend a significant amount of time on inter-thread value communication as shown by the first two segments (e.g., *Sync* and *Fail*) in each bar. The best possible optimization for reducing the cost of value communication is to prevent any data dependence speculation from failing and any synchronization from stalling. This ideal behavior is measured by simulating the same executables with a hypothetical model that perfectly predicts the values needed by all consumer instructions of inter-thread data dependences. The results are shown as **O** bars in the figure.[3] The **N** bars represent the execution time of the benchmarks when only register-resident values are perfectly predicted to prevent synchronization stalls. Figure 5.6 shows significant performance potential for optimizing inter-thread value communication for all three sets of loops. By completely eliminating synchronization stalls for register-resident values with perfect value predictors, the parallelized portion of the three sets of loops can potentially speed up by 38%, 5% and 10%, respectively. On top of that, by completely elimi-

[3]For some of the benchmarks, the *other* segment for the **O** bars has increased relative to the **U** bars. Such increases are due to second order effects such as increasing in data cache miss rate.

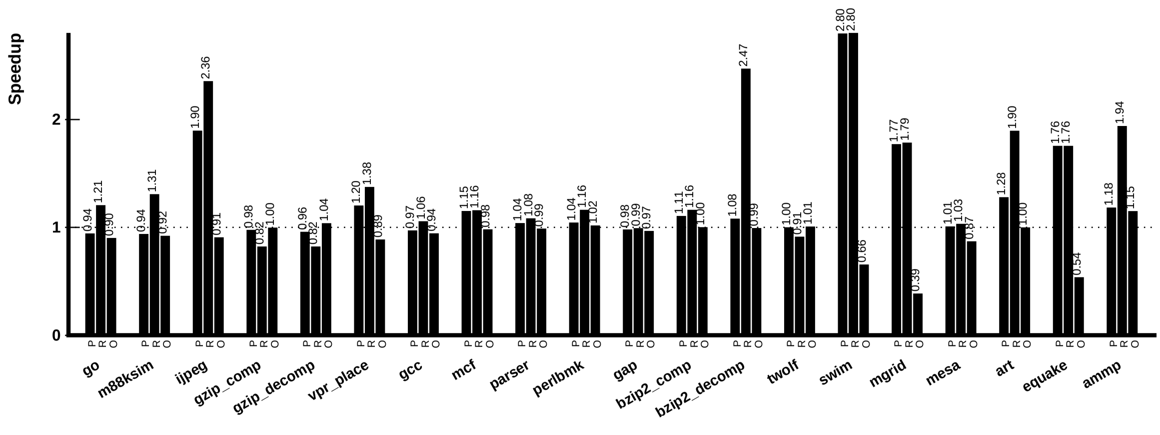
nating speculation failures with perfect value prediction for memory-resident values, the three sets of loops can potentially speed up by an additional 8%, 34%, and 27%, respectively.

The rest of this chapter first summarizes the speedup achieved by the entire program and the parallelized regions (Section 5.6), then presents the results of the compiler optimizations to reduce the critical forwarding paths due to register-resident values (Section 5.7) and the results for the compiler optimizations to automatically synchronize frequently occurring memory-resident values (Section 5.8).

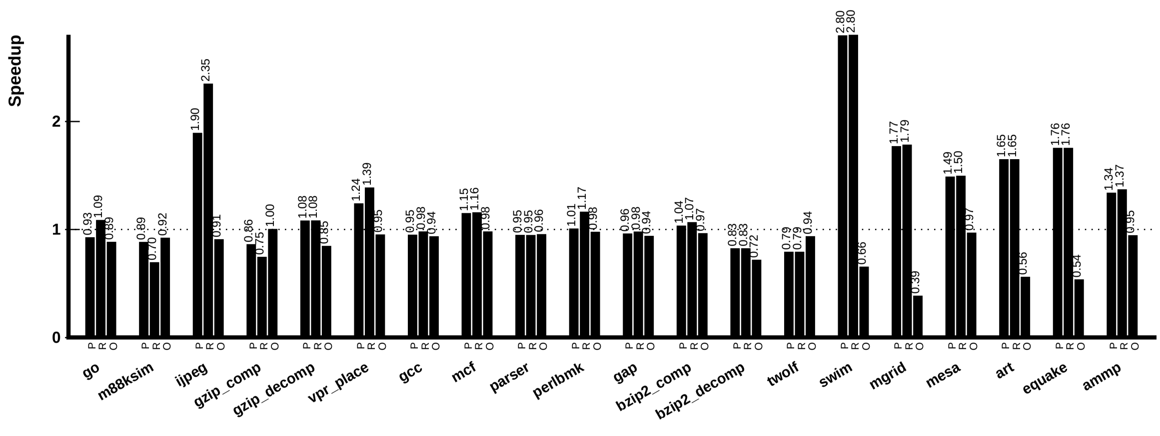
5.6 Program Performance

Figure 5.7 summarizes the speedup achieved by each benchmark with respect to the original sequential executable on a four-processor chip-multiprocessor with conservative instruction scheduling for register-resident values and with a hybrid of hardware-based and compiler-based automatic synchronization for memory-resident values. The *P* bars show the speedups achieved by the entire program, the *R* bars show the speedups achieved by the parallelized loops, and the *O* bars show the speedups for the portion of the program not parallelized. For some benchmarks, such as GO, the parallelized loops speed up relative to the sequential execution, while the program performance is pulled down by the slowdowns in the non-parallelized portion of the program. This behavior is caused by: (i) the hampered compiler optimization (due to instructions inserted to transform the sequential programs into parallel programs), and (ii) the decreased data-cache locality (due to the spreading of data items across multiple first-level caches during parallel execution). A more detailed discussion of this behavior can be found in Steffan's thesis [60].

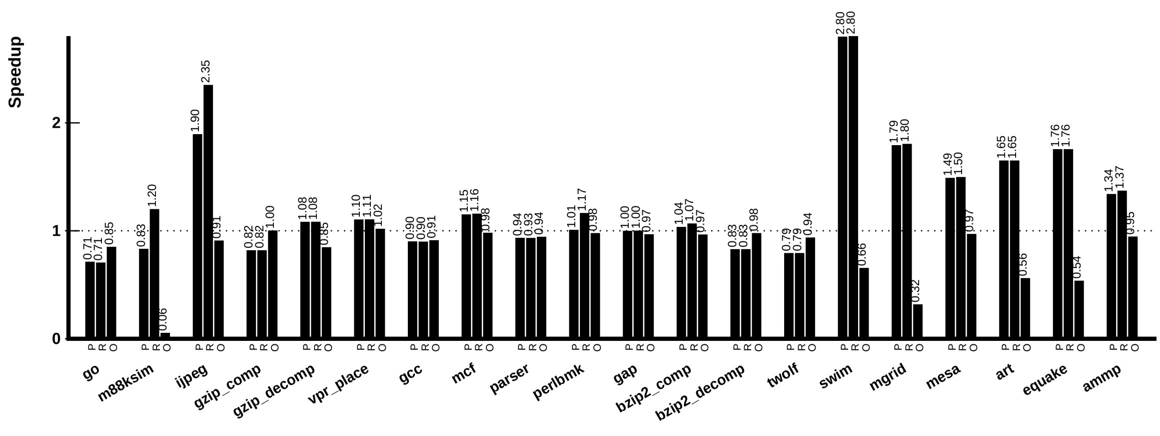
Speedups achieved by different benchmarks vary greatly: some benchmarks, such



(a) Register set



(b) Realistic set



(c) Idealistic set

Figure 5.7: Speedup achieved with TLS on a four processors CMP with previously described optimizations. P is program speedup, R is region speedup, and O is outside-region speedup.

as GZIP_COMP, GAP and TWOLF, are unable to speed up at all relative to sequential execution; while some other benchmarks, such as IJPEG, SWIM, MGRID, MESA, ART and EQUAKE show more than 40% program speedup. One interesting question is which loop set achieves the best program performance when parallelized with inter-thread value communication optimizations. Intuitively, the *realistic set* and the *idealistic set*, having a much higher coverage than the *register set*, would be expected to achieve better program performance. However, nine out of fourteen integer-benchmarks achieve the best performance with the *register set* and two benchmarks achieve the same performance with all loop sets. Only three benchmarks, GZIP_DECOMP, VPR_PLACE, and GAP, achieve the best performance with the *idealistic set* or the *realistic set*. The floating point benchmarks, on the other hand, almost always achieve best performance with the *idealistic set* or the *realistic set*. Thus, selecting the right set of loops to parallelize is very important for overall program performance.

Since the goal of this dissertation is to improve the performance of parallelized regions by improving inter-thread value communication, we focus on the performance improvement achieved by the parallelized loops in the rest of this dissertation.

5.7 Reducing the Critical Forwarding Path

This section presents the experimental results that quantify the performance impact of the scheduling algorithms described in Chapter 4 using the three sets of loops selected in Section 5.4. We also compare our approach against previously proposed techniques that intend to reduce the critical forwarding paths: (i) hardware-based instruction scheduling techniques [64]; and (ii) the Multiscalar instructions scheduling techniques [68].

5.7.1 Impact of Conservative Scheduling

Figure 5.8 shows the performance impact of the conservative scheduling algorithm described in Section 4.3 on parallelized loops. Note that in most cases, the unscheduled version (*U*) slows down relative to the original sequential version (i.e., the *U* bars are almost always above the reference line).

When the scheduling algorithm is applied to loop induction variables alone (*I*), the time spent on synchronization stalls decreases significantly, which results in significant parallel loop speedup: 24% for the *register set*, 19% for the *realistic set*, and 13% for the *idealistic set*. The *register set* responds most positively to this optimization, since its execution is previously dominated by synchronization stalls. By scheduling instructions to reduce the critical forwarding path for all forwarded values, as described in Section 4.3, seven benchmarks from the *register* set enjoy an additional 7% speedup on parallel loops on average, as shown by the *S* bars. The same trends are observed in the other sets of loops, although not as pronounced because their execution is dominated by other performance bottlenecks, such as speculation failures. GZIP_COMP and GZIP_DECOMP of the *register set* are the only two benchmarks that spend significant amount of time on synchronization but are unable to speed up with this optimization. This is because the complex control flow structures that are inherent in GZIP prevent instructions from being scheduled further back.

It is worth noting that the reduction in synchronization stall time does not always translate directly into improved performance. For example, in the *register set*, synchronization time is greatly reduced for BZIP2_COMP with instruction scheduling for all forwarded values rather than just for induction variables. However, the performance of the resulting program does not improve proportionally. The reason behind this behavior is that other inter-thread data dependences are exposed as the critical forwarding path shrinks, causing speculation to fail more often.

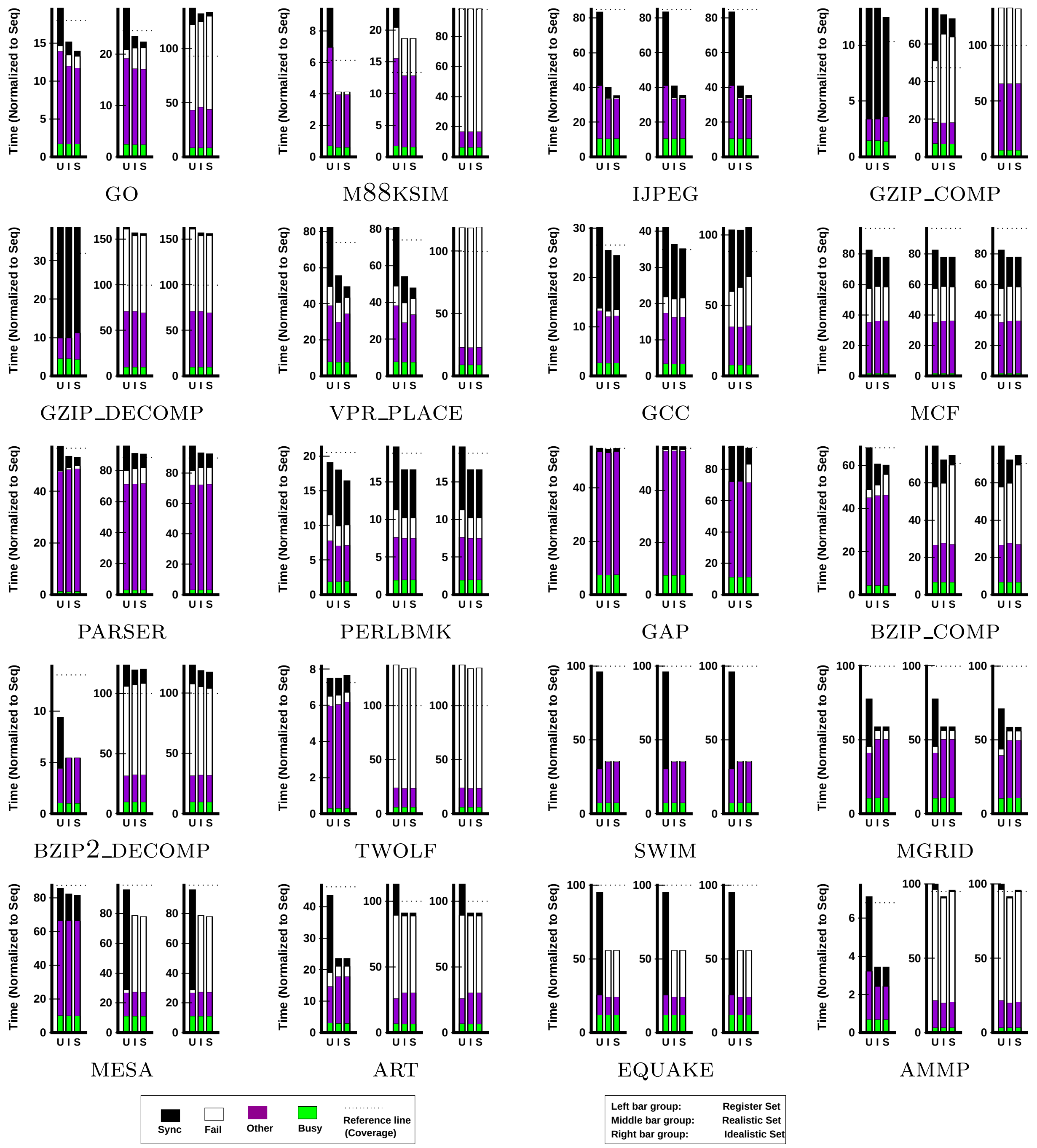


Figure 5.8: **Impact of instruction scheduling on reducing critical forwarding path for register-resident values.** For each benchmark, three sets of results are presented, corresponding to the performance of the register, the realistic, and the idealistic loop sets, respectively. Bars represent execution time of the parallel loops on a four-processor CMP normalized to the sequential program execution time and the reference line represents the coverage of each loop set. **U** is unoptimized, in which all register-resident values are synchronized at first use and last definition; **I** corresponds to only optimizing critical forwarding path introduced by induction variables; **S** corresponds to reducing critical forwarding paths for all register-resident values.

5.7.2 Comparing Conservative Scheduling with the Multiscalar Algorithm

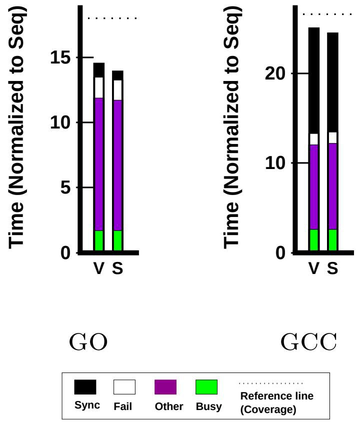
Since the Multiscalar scheduler [68] is essentially a dataflow algorithm that only traverses the control flow graph once, its operation can be estimated by constraining our conservative scheduling algorithm as follows: by modifying the meet operator such that it returns $\perp$ whenever $\top$ meets with any value that is not $\top$—this way, the modified dataflow analysis converges during the first iteration.

Figure 5.9(b) shows a simplified version of a loop in GCC (at line `reorg.c`:2680) that highlights the advantage of the more general dataflow approach of our conservative scheduling algorithm over the Multiscalar algorithm [68]. While the original version of this loop has multiple variables that are forwarded, this experiment focuses on the variable `insn`. The Multiscalar scheduler cannot move the assigning and forwarding of `insn` above the inner loop in the `case` statement, while our approach iterates to a dataflow solution where it can.

Figure 5.9(b) compares the performance of our conservative scheduling technique with that of the Multiscalar algorithm for two benchmarks from the *register set* where there is a significant difference in performance (i.e. GCC and GO). Compared with the Multiscalar algorithm, for the parallelized portion of the programs, our conservative scheduling approach used here reduces synchronization time by 6% for GCC and by 35% for GO, which in turn reduces the respective region execution times by 4% and 2% relative to the Multiscalar approach. This result is not surprising since the Multiscalar algorithm was designed for smaller, simpler regions.

```
for (insn = target; insn; insn = next) {
   rtx this_jump_insn = insn;
   next = NEXT_INSN(insn);
   switch (GET_CODE(insn)){
      case ...:  ...
      case INSN:
         if (GET_CODE(PATTERN(insn)) == USE){
            ...; continue;
         } else if (GET_CODE(PATTERN(insn)) == CLOBBER){
            continue;
         } else if (GET_CODE(PATTERN(insn)) == SEQUENCE){
            for (i=0; i<XVECLEN(PATTERN(insn),0); i++){
               ...
            }
         }
   }
}
```

(a) An example.



(b) Execution time of GCC and GO normalized to sequential program execution for the *register set*. *V* approximates the Multiscalar scheduling technique, and *S* uses the conservative scheduling algorithm described in Section 4.3.

Figure 5.9: Comparison with the Multiscalar scheduling algorithm.

5.7.3 Impact of Aggressive Scheduling

Figure 5.10 shows the impact of aggressive instruction scheduling. The first bar (*S*) for each benchmark shows the performance of the conservative scheduling (as seen earlier in Figure 5.8). The *sync* portion of these bars shows the potential gain from better scheduling. The following discussion focuses on the *register set*, since it has the most performance potential. Comparing with conservative instruction scheduling, PERLBMK achieves 9% parallel loop speedup when speculating on control dependences (*"C"* bars), and GCC and TWOLF achieve 8% and 22% parallel loop speedup when speculating on data dependences (*"D"* bars). However, BZIP_DECOMP slows down when instructions are speculatively scheduled across data dependences. This behavior suggests that aggressively instruction scheduling should be applied only when synchronization stall is the main performance bottleneck. We should also point out that control dependence speculation and data dependence speculation are complementary, and by combining the two techniques, we are always able to get the benefits of both optimizations (*"A"* bars).

5.7.4 Comparison with Hardware-Based Optimizations

This section discusses the effectiveness of the compiler versus the hardware at optimizing the critical forwarding paths and attempts to answer two questions: (i) without compiler-based instruction scheduling, whether hardware is effective in reducing the critical forwarding path, and (ii) with compiler-based instruction scheduling algorithm, whether the hardware can offer additional performance improvement. The hardware can reduce the critical forwarding path by predicting the forwarded values and by giving high priorities to instructions that produce forwarded values in the reorder buffer. Detailed descriptions of the hardware techniques can be found in our previous work [64].

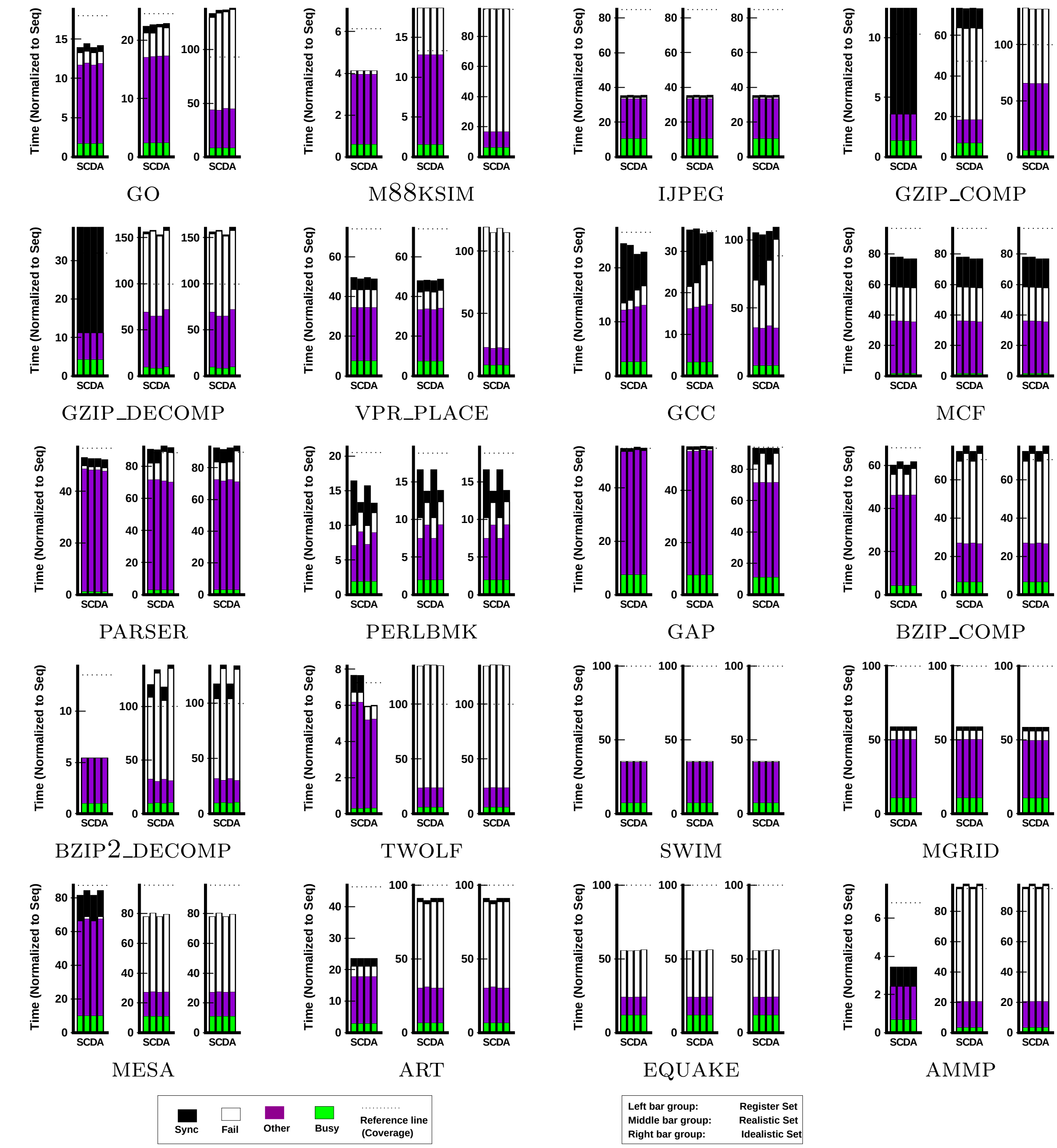


Figure 5.10: **Impact of speculative instruction scheduling on reducing critical forwarding path for register-resident values.** For each benchmark, three sets of results are presented, corresponding to the performance of the register, the realistic, and the idealistic loop sets, respectively. Bars represent execution time of the parallel loops on a four-processor CMP normalized to the sequential program execution time and the reference line represents the coverage of each loop set. **S** schedules instructions using conservative instruction scheduling algorithm; **C** schedules instructions across control dependences; **D** schedules instructions across data dependences; **A** schedules instructions across both control and data dependences.

The first two bars for each benchmark in Figure 5.11 answer the first question: they show the performance of the unoptimized executables running speculatively in parallel without (U bars) and with hardware-based optimization (H bars), respectively. For all benchmarks in all loop sets, the H bars show no significant performance improvement over the U bars. The next two bars evaluate whether the compiler and the hardware are complementary by showing an optimized executable running speculatively in parallel without (S bars) and with hardware-based optimization (G bars), respectively. Once again, the G bars show no significant performance improvement over the B bars. Thus, the hardware-based technique is not effective in reducing the critical forwarding path.

5.8 Automatically Synchronizing Memory Accesses

After eliminating time spent in synchronizing register-resident values, we now study the other performance bottleneck: time wasted on failed speculation. This section presents the experimental results that quantify the performance impact of the compiler-based techniques for automatically inserting synchronization to avoid speculation failures, as described in Section 3.5.2. The effectiveness of these techniques is also compared with related hardware-based approaches [15, 64].

Figure 5.12 shows the performance impact of compiler-based automatic synchronization for frequently occurring memory-resident data dependences. It shows the time spent on parallelized regions normalized to the execution time of the original sequential programs. The S bars, the same as in Figure 5.8, are the baselines in this experiment, where no synchronization is inserted for memory-resident values, but while register-resident scalars are synchronized. The M bars show the performance of these parallelized regions when frequently dependent memory accesses are synchronized. In this experiment, data dependences are considered frequent-occurring if they

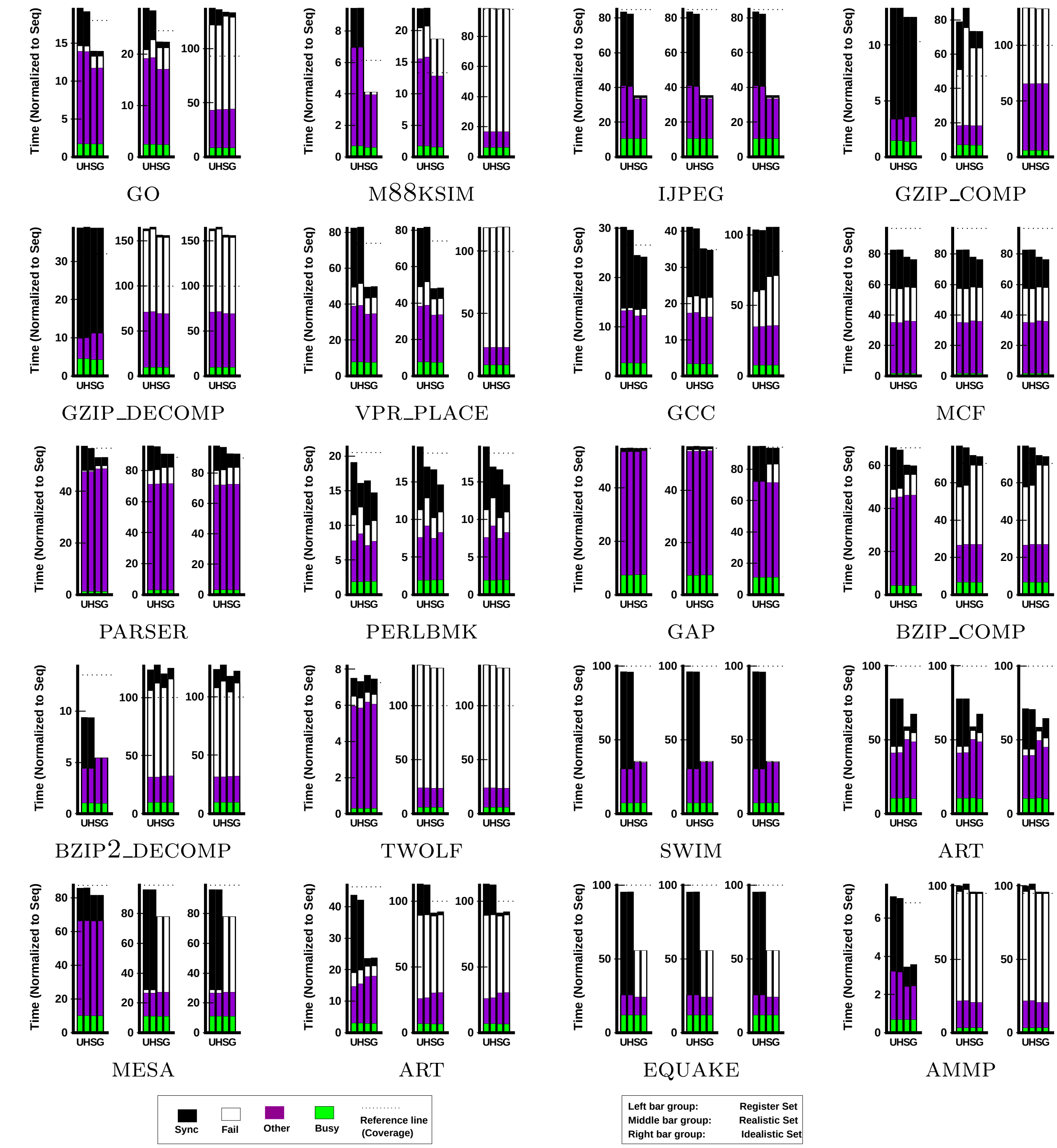


Figure 5.11: **Impact of hardware optimization vs. compiler optimization for reducing critical forwarding path.** For each benchmark, three sets of results are presented, corresponding to the performance of the register, the realistic, and the idealistic loop sets, respectively. Bars represent execution time of the parallel loops on a four-processor CMP normalized to the sequential program execution time and the reference line represents the coverage of each loop set. **U** is unoptimized, in which all register-resident values are synchronized at first use and last definition; **H** uses hardware optimization but not compiler optimization; **S** uses compiler optimization but not hardware optimization; **G** uses both the hardware and the compiler.

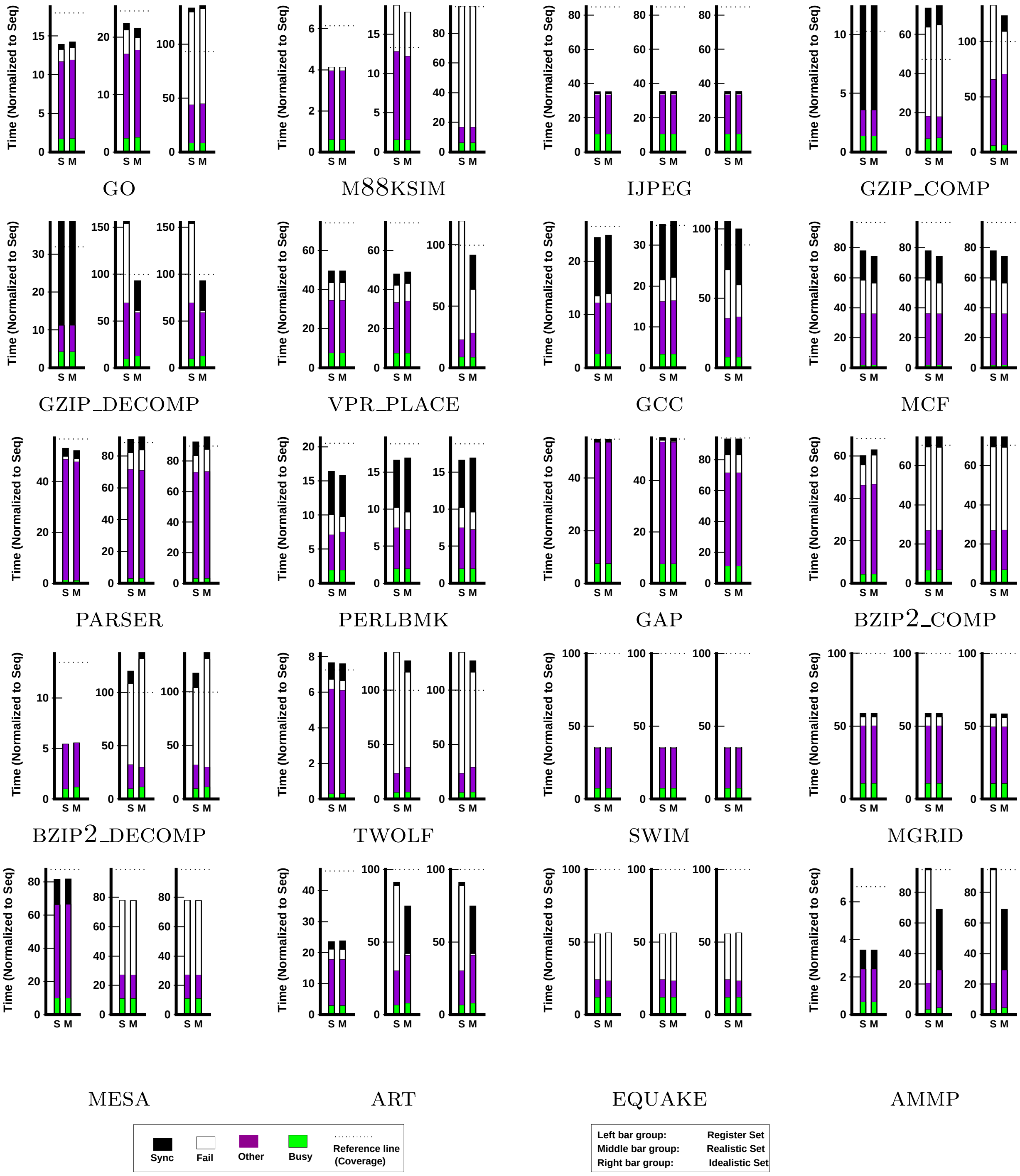


Figure 5.12: **Impact of compiler-inserted synchronization on reducing speculation failures.** For each benchmark, three sets of results are presented, corresponding to the performance of the register, the realistic, and the idealistic loop sets, respectively. Bars represent execution time of the parallel loops on a four-processor CMP normalized to the sequential program execution time and the reference line represents the coverage of each loop set. **S** has no synchronization for memory-resident values; **M** has compiler-inserted synchronization for memory-resident values, as described in Section 3.5.2.

occur in more than 4% of all threads, and we synchronize all load/store instructions that can be reached from the parallelized loops within five function calls. Appendix C provides detailed information about how these thresholds are selected.

Different loop sets respond differently. When comparing the S bars with the M bars, we observe that, in the *register set*, although four benchmarks, MCF, PARSER, PERLBMK and TWOLF, are able to speed up but none by more than 5%. This is because the loops included in this loop set do not suffer from excessive speculation failures to begin with. Thus, we focus on the *realistic set* and the *idealistic set* for the rest of this discussion. For the *realistic set*, seven benchmarks, GO, M88KSIM, GZIP_DECOMP, MCF, TWOLF, ART and AMMP, are able to achieve parallel loop speedup. Among them, M88KSIM speeds up by 5%, GZIP_DECOMP by 41%, TWOLF by 6%, ART by 16%, and AMMP by 28%. Unfortunately, BZIP_DECOMP slows down by as much as 15% due to improper synchronization. Section 5.9 will discuss how improving profiling accuracy can help us avoid such performance degradation. The *idealistic set* also responds well to the optimization—nine benchmarks, GZIP_COMP, GZIP_DECOMP, VPR, GCC, MCF, PERLBMK, TWOLF, ART and AMMP, are able to speed up and the the exceptional performers from the *realistic set* are among them.

5.8.1 Comparing Compiler-Based and Hardware-Based Automatic Synchronization

Previous research [15, 64] proposed two *hardware* techniques to reduce the cost of failed speculation due to memory-resident values: *prediction* and *synchronization*. Neither of the proposed techniques requires centralized structures to match dependence pairs; however, they differ in complexity, from a 2KB violation prediction table [15] to two 32-entry tables that track loads that are exposed and loads that have caused speculation to fail [64]. In the approach described in [64], the hardware identifies

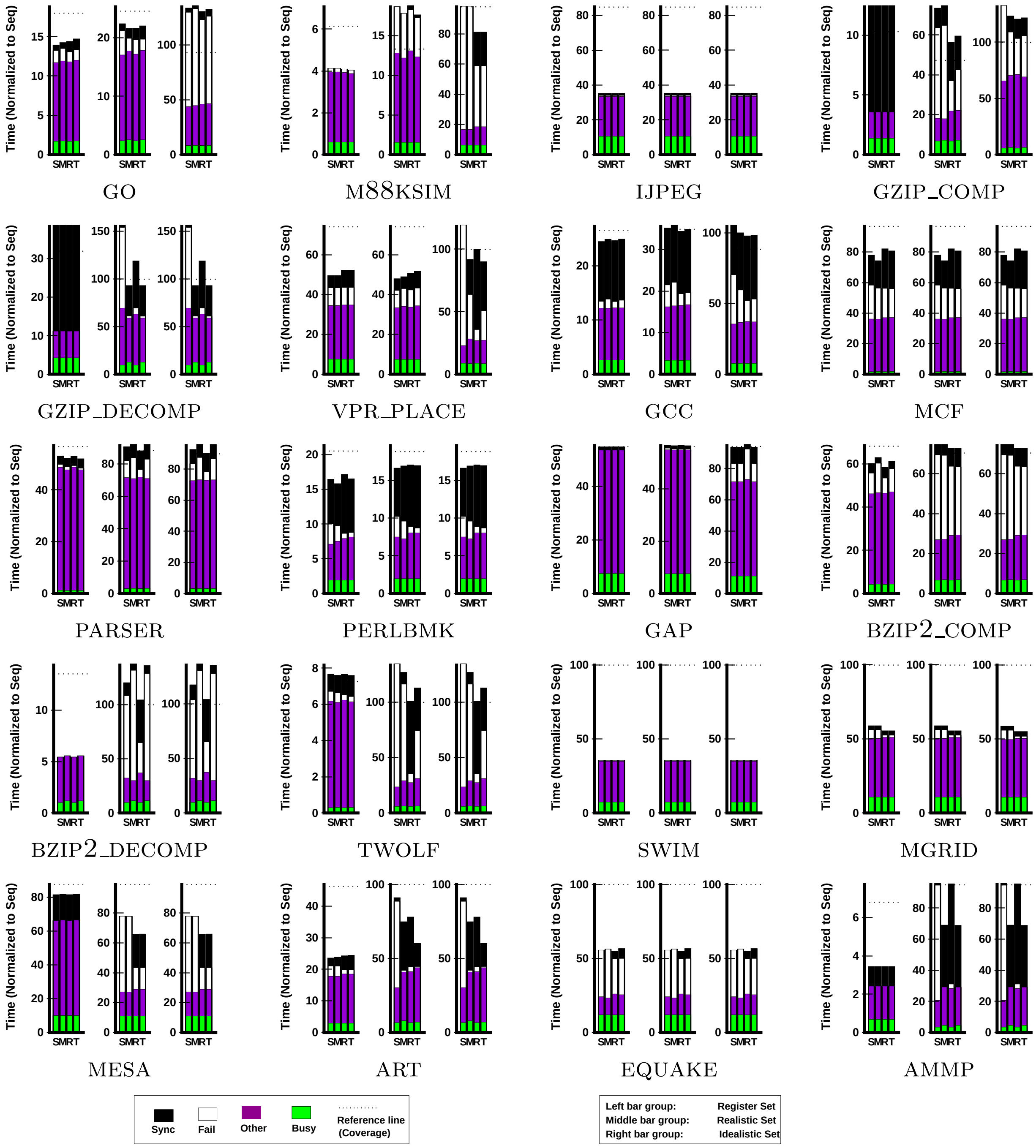


Figure 5.13: **Impact of compiler-inserted vs hardware-inserted synchronization.** For each benchmark, three sets of results are presented, corresponding to the performance of the register, the realistic, and the idealistic loop sets, respectively. Bars represent execution time of the parallel loops on a four-processor CMP normalized to the sequential program execution time and the reference line represents the coverage of each loop set. **S** has no synchronization for memory-resident values; **M** has compiler-inserted synchronization for memory-resident values, as described in Section 3.5.2; **R** has hardware-inserted synchronization for memory-resident values, as described in [64]; and **T** has both compiler and hardware-inserted synchronization.

loads that frequently cause violations with a *load address table* and stalls these loads until the previous threads complete. To avoid over-synchronization of infrequently dependent loads, the *load tables* are periodically reset. This is the hardware technique that we compare against.

In Figure 5.13, the *M* bars show the execution time breakdown for compiler-inserted synchronization, while the *R* bars show the execution time breakdown for hardware-inserted synchronization. A comparison between compiler-inserted and hardware-inserted synchronization techniques reveals that *each of the techniques wins in some cases, but neither technique is consistently better.* For the *realistic set*, eight benchmarks (GZIP_COMP, GCC, PARSER, BZIP2_COMP, BZIP2_DECOMP, TWOLF, MGRID, and MESA) perform better using hardware-inserted synchronization by 14% on average for the parallel loops; six benchmarks (M88KSIM, GZIP_DECOMP, VPR_PLACE, MCF, ART, and AMMP) perform better with compiler-inserted synchronization by 12% on average for the parallel loops. Similarly, in the *idealistic set*, ten benchmarks perform better using hardware-inserted synchronization, and six benchmarks perform better using compiler-inserted synchronization. We will now present some insights for why benchmarks perform differently.

First, for some benchmarks, (e.g., M88KSIM), violations are not caused by true data dependences, but rather by *false sharing.* The compiler attempts to synchronize only true dependences, while the hardware is able to synchronize both true data dependences and false sharing. Although the compiler could also track data dependences at a word granularity, other techniques (such as memory layout optimizations or loop unrolling) are better suited to address false sharing in the compiler.

Second, for some benchmarks, such as GZIP_DECOMP, the compiler and the hardware both insert synchronization; however, the compiler is able to speculatively forward the desired value much earlier than the hardware can. This reduces over-synchronization, resulting in better performance.

The hardware-based and the compiler-based techniques each benefits a different set of benchmarks and for almost all benchmarks at least one optimization is able to improve region performance over the unoptimized case. This suggests that a hybrid of the two techniques may be able to achieve a better overall performance. To evaluate such a compiler-hardware hybrid, both hardware-inserted synchronization and the compiler-inserted synchronization are enabled. The results are shown in Figure 5.13 as bar *T*. With the hybrid, twelve benchmarks are able to speed up relative to the unoptimized cases in the *realistic set*. The hybrid approach is able to captures the performance of the better of the two techniques: M88KSIM benefits from hardware-inserted synchronization and avoids the cost of false sharing, while GZIP_DECOMP benefits from having values be forwarded early by compiler-inserted synchronization. Therefore, it is possible for us to implement a hybrid that can improve the performance of a larger set of programs by taking advantage of both compiler and hardware inserted synchronization.

5.8.2 Impact of Instruction Scheduling on Memory-Resident Values

Reduction in the time spent on failed speculation does not always translate directly into improved performance; sometimes, it is traded with the time spent on synchronization. Five benchmarks (GZIP_DECOMP, VPR, BZIP2_DECOMP, ART, and AMMP) from the *realistic set* and the *idealistic* sets demonstrate significant increases in the time spent on synchronization after synchronizing data dependences for memory-resident values (*"M"* bars). This section describes an attempt to reduce the cost of synchronizing memory-resident values using the same instruction scheduling techniques that have been developed to reduce the cost of synchronizing register-resident values.

Figure 5.14 shows the results of scheduling *signal* instructions and their dependent instructions as early as possible within a thread for those benchmarks suffering from synchronization stalls due to memory-resident value synchronization. As shown by the *"W"* bars, for three benchmarks, VPR_PLACE, ART and AMMP, instruction scheduling is able to significantly reduce time spent on synchronization stalls and improve performance by 5.5%, 7.7% and 17.4%, respectively.

Forwarding memory-resident values before the corresponding stores are issued introduces extra complexity compared with forwarding register-resident values. From the compiler's perspective, addresses and data values must both be forwarded upon synchronization, and thus the dataflow analysis algorithm must propagate both address computations and value computations through the control flow graph. At run-time, we use a `send_addr_buffer` to record the addresses that have been forwarded, and the next thread is restarted when a forwarded address is modified. If the *signal* instructions always occur after their corresponding stores, the forwarded addresses can be entered into the `send_addr_buffer` when signals are issued. However, if the *signal* instructions occur before their corresponding store instructions, the forwarded addresses should not be entered into the `send_addr_buffer` until the stores are issued. Hence, extra `to_buffer` instructions must be inserted after the stores.

5.9 Sensitivity to the Accuracy of Profiling Information

All the experiments in previous sections have assumed *realistic* profiling information: i.e., profiling information is collected with the `train` input set and performance evaluation is done using the `ref` input set. Can we further improve program performance

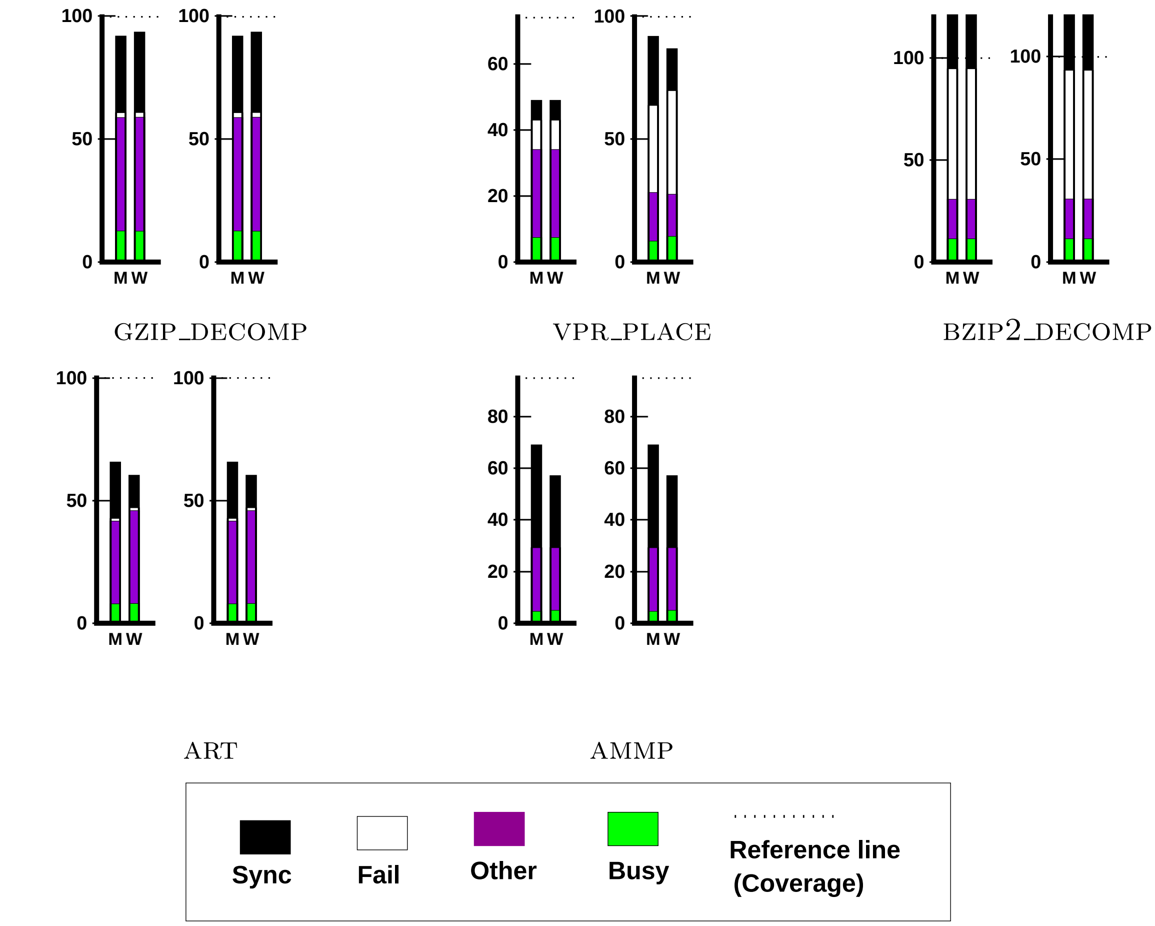


Figure 5.14: **Impact of instruction scheduling for memory-resident value synchronization.** Results shown are for the *realistic* and *idealistic* sets. **M** has compiler-inserted synchronization for memory-resident values; **W** has compiler-inserted synchronization for memory-resident values with *signal* instructions scheduled using the algorithms described in Section 4.3.

with more accurate profiling information? This section demonstrates the sensitivity of our optimization techniques to the accuracy of profiling information with a new set of experiments that perform optimizations using accurate profiling information collected with the `ref` input set. The results of these experiments are compared with the results obtained when realistic profiling information is used. The rest of this section discusses the sensitivity of two optimizations, speculative instruction scheduling for register-resident values and automatic synchronization for memory-resident values.

The impact of profiling accuracy on speculative instruction scheduling is studied by focusing on the *register* set, which is the set of loops that have shown significant performance improvement with this technique. Speculative instruction scheduling is relatively insensitive to the accuracy of data dependence profiling information. Only three out of twenty benchmarks schedule instructions differently with different data dependence profiling information and the results for these three benchmarks are shown in Figure 5.15. Their performance is not affected by improving profiling accuracy. Speculative instruction scheduling is also insensitive to the accuracy of control dependence profiling. Although nine benchmarks schedule instructions differently with different profiling information, Figure 5.16 shows that none of these nine benchmarks is able to obtain a significant performance gain with the more accurate profiling information. Thus, with the help of more accurate profiling information, we can potentially improve inter-thread value communication ever further for memory-resident values (detailed discussion in Section 6.1.

To study the impact of profiling accuracy on automatic synchronization for memory-resident values, we focus on the *realistic set*, which is the set of loops that demonstrate significant performance improvement under this technique. Twelve benchmarks schedule synchronization differently with different profiling information and five benchmarks obtain better performance with more accurate profiling information. The performance comparison for these twelve benchmarks is shown in Figure 5.17.

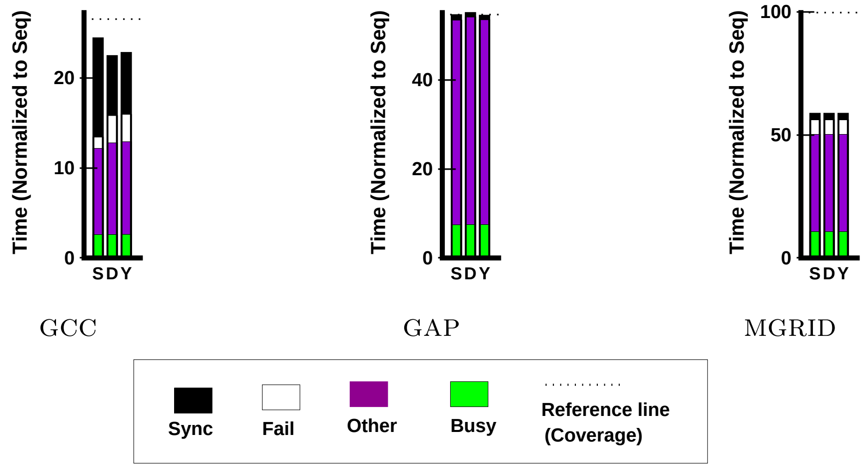


Figure 5.15: **Impact of profiling accuracy on speculatively scheduling instructions across data dependences.** Results are shown for the *register set only*. Bars represent execution time of the parallel loops on a four-processor CMP normalized to the sequential program execution time and the reference line represents the coverage of each loop set. **S** schedules instructions using conservative instruction scheduling algorithm; **D** schedules instructions across data dependences, profiled with the `ref` input set; **Y** schedules instructions across data dependences, profiled with the `train` input set.

There are two reasons for how less accurate profiling information degrades performance: (i) the forwarded addresses are frequently modified after the values have been forwarded, such is the case for GZIP_COMP, BZIP_COMP and BZIP_DECOMP; (ii) the compiler synchronized data dependences that do not occur very frequently, such is the case for PERLBMK and ART. Thus, by improving profiling accuracy, we can potentially improve the efficiency of inter-thread value communication even further. (Details discussed in section 6.1.)

5.10 Chapter Summary

This chapter presents the results of detailed simulations designed to evaluate the effectiveness of the value communication optimization algorithms described in previous chapters. The chapter begins with a description of the simulation framework and

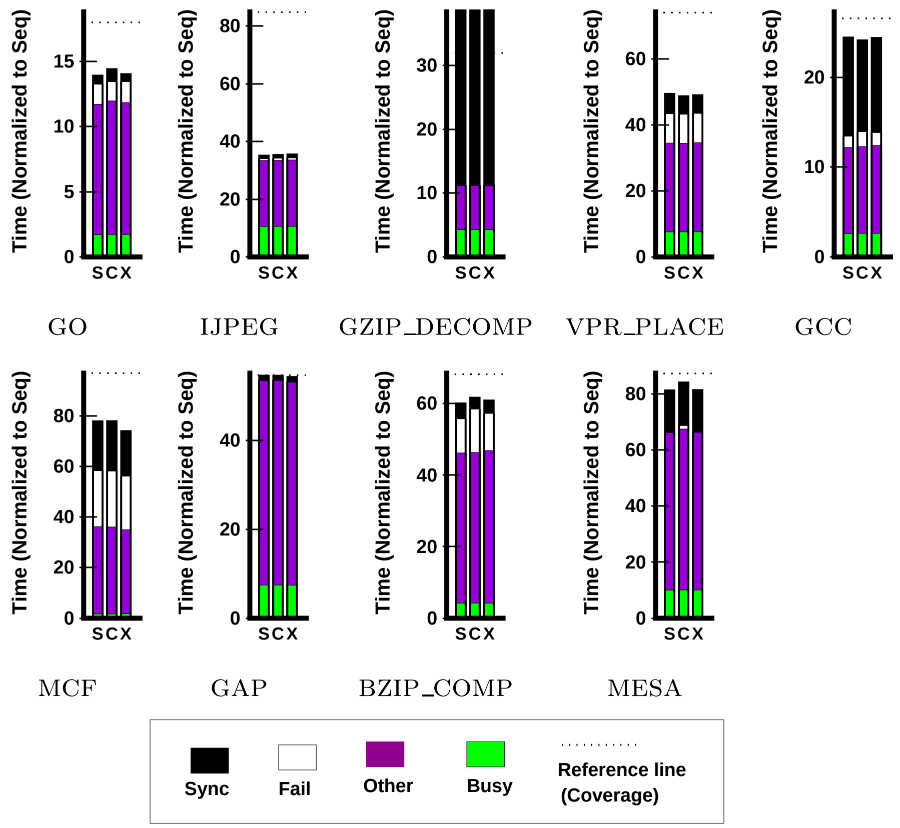


Figure 5.16: **Impact of profiling accuracy on speculatively scheduling instructions across control dependences.** Results are shown for the *register set only*. Bars represent execution time of the parallel loops on a four-processor CMP normalized to the sequential program execution time and the reference line represents the coverage of each loop set. **S** schedules instructions using conservative instruction scheduling algorithm; **C** schedules instructions across control dependences, profiled with the `ref` input set; **X** schedules instructions across control dependences, profiled with the `train` input set.

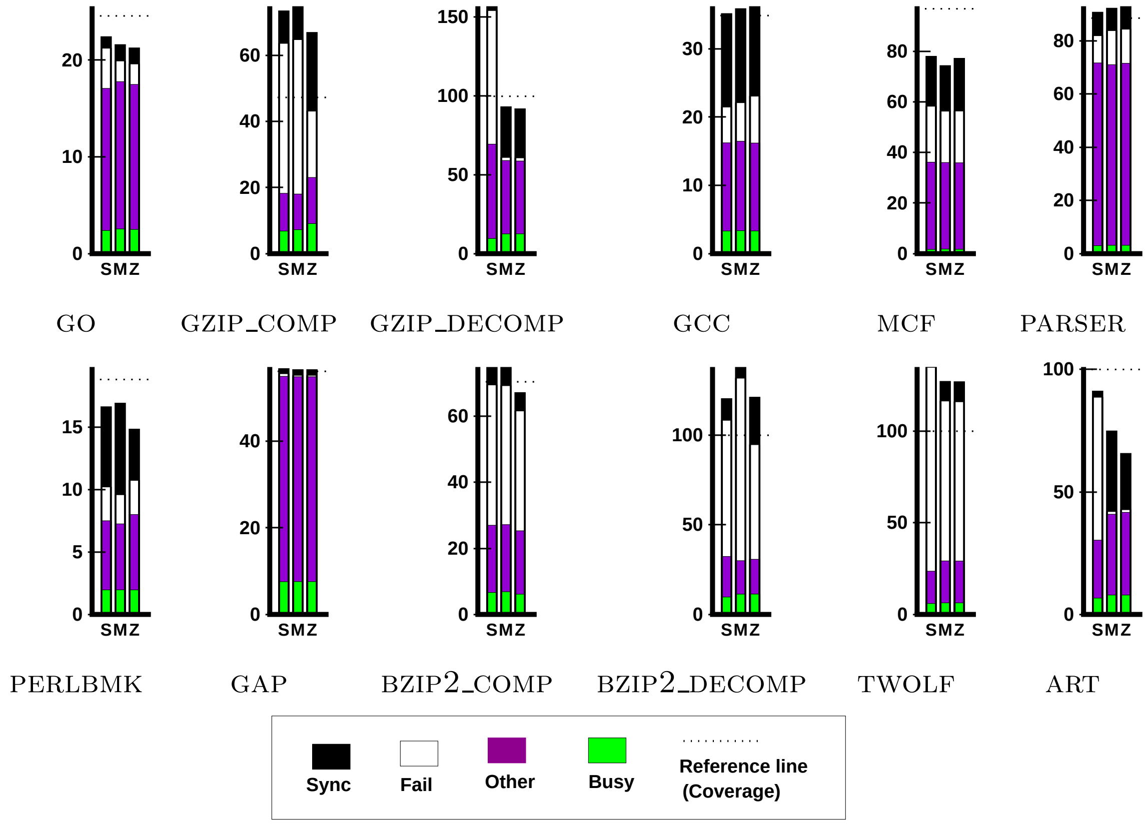


Figure 5.17: **Impact of profiling accuracy on compiler-inserted synchronization.** Results are shown for the *realistic set* only. **S** has no synchronization for memory-resident values; **M** has compiler-inserted synchronization for memory-resident values profiled with the `ref` input set; **Z** has compiler-inserted synchronization for memory-resident values profiled with the `train` input set.

characteristics of all targeted benchmarks.

The performance impact of instruction scheduling techniques for reducing the critical forwarding paths introduced by register resident values is evaluated, finding the following:

- The proposed instruction scheduling technique is effective in reducing critical forwarding path and improve parallel loop performance for the *register set*, the *realistic set* and the *idealistic set* by 25%, 19% and 13%, respectively.
- The proposed algorithm's ability to handle complex control flows performs well in the presence of inner loops.
- The proposed compiler-based techniques are capable of examining the entire program and scheduling instructions across a very long distance, and thus they eliminate the need for hardware-based techniques targeting the same problem.
- Speculative instruction scheduling algorithms that move instructions across intra-thread control and data dependences are useful, especially when the critical forwarding path is still significant after the conservative instruction scheduling has been applied. With speculative instruction scheduling, we are able to obtain: 9% performance improvement for PERLBMK, 8% for GCC and 22% for TWOLF, for the *register set*.
- Overall, speculative instruction scheduling is insensitive the accuracy of profiling information: only GZIP_DECOMP slows down significantly with less accurate profiling information.

The performance impact of automatically synchronizing memory-resident values has also been evaluated, producing the following findings,

- The proposed automatic synchronization algorithms are effective in reducing speculation failures.

- The proposed compiler-based techniques and hardware-based techniques each benefit a different set of benchmarks. However, when they work in tandem, more benchmarks are able to speed up;

- Instruction scheduling techniques that reduce the critical forwarding path also work on memory-resident values. Three benchmarks, VPR_PLACE, ART and AMMP, speed up with instruction scheduling by 5.5%, 7.7% and 17.4%, respectively.

- For most benchmark, profiling-based automatic synchronization is insensitive to profiling accuracy. Although the compiler synchronizes different instructions when inaccurate profiling information is used, the performance is unaffected for most benchmarks. However, for a few benchmark, GZIP_COMP, PERLBMK, BZIP_COMP, BZIP_DECOMP and ART, more accurate profiling information can lead to small performance improvement.

The speedup obtained by different set of loops vary greatly. For the integer benchmarks, the *register set* achieves the best performance: 28% average parallel loop speedup and 9% average program speedup. For the floating point benchmarks, the *realistic set* and the *idealistic set* both achieved the best performance: 81% average parallel loop speedup and 80% average program speedup (all floating point benchmarks have close to 100% coverage).

Chapter 6

Conclusions

Under the context of TLS, efficient inter-thread value communication is the key to achieving good performance. The main contribution of this dissertation is the design, implementation, and evaluation of several compiler-based techniques that improve the efficiency of inter-thread value communication.

Value communication for register-resident values and memory-resident values are handled separately, since register-resident values are usually predictable and easy to analyze statically, while memory-resident values are the opposite. As a baseline, we rely on the compiler which inserts explicit synchronization to communicate register-resident values, and on the hardware, which detects inter-thread data dependences to ensure correct execution for memory-resident values. The key results of this dissertation are the following:

1. Using compiler-inserted synchronization to communicate *memory-resident* values that would otherwise cause frequent dependence violations does improve TLS performance in many cases: a subset of the benchmarks enjoy significant region speedups, while others are unaffected.

2. By comparing our compiler-based approach with our hardware-based approach [64]

for synchronizing memory-resident values, we observe that both approaches are useful, and that neither one consistently outperforms the other. By combining the two approaches, we obtain a hybrid that tracks the better performer.

3. Scheduling instructions, with our conservative instruction scheduling algorithm, reduces the critical forwarding path for all synchronized register-resident scalars, and improves performance significantly.

4. The instruction scheduling techniques are also effective in reducing the critical forwarding path created by synchronizing frequently occurring data dependences between memory-resident values.

5. Comparing our compiler-based approaches and our hardware-based approaches demonstrates that the compiler can be effective in mitigating the performance penalty from the critical forwarding path without requiring additional hardware support beyond what is normally needed for TLS.

6. Speculatively scheduling instructions across control and data dependences yields additional performance improvement and the their performance effects are complementary. We believe if hardware resources are to be devoted to reduce the critical forwarding path, they are best spent on implementing the instructions necessary to support speculative instruction scheduling.

6.1 Future Work

The goal of our research is to improve the efficiency of inter-thread value communication under the context of TLS. In this section, we briefly describe how our work can be further extended.

In this dissertation, the compiler is presented with two mechanisms to communicate values between speculation threads: speculation and synchronization. Speculation is suitable for satisfying infrequently occurring data dependences, while synchronization is suitable for satisfying frequently occurring data dependences. Most data dependences are bi-modal, i.e., they either occur very frequently or rarely, and thus these two value communication mechanisms suffice. However, there are some cases in which data dependences cause enough speculation failure to degrade performance if speculated upon, but stall the program more than necessary if synchronized. For these cases, we can efficiently communicate values to satisfy these data dependences using a recovery mechanism that execute only the minimal set of instructions necessary, identified by the compiler, when speculation fails. This approach is especially beneficial for large threads, and thus it would encourage extracting parallelism at coarser granularity.

Several techniques presented in this dissertation rely on profiling information to identify optimization opportunities, however, collecting profiling information, especially data dependence profiling information, is a very time consuming process. One potential solution for speeding up this process is to utilize the hardware. Chen and Olukotun [10] has proposed hardware support in order to dynamically tracks the data dependence pairs that would serialize parallel execution the most (a.k.a. the critical forwarding path). Similar hardware support can be used for fast profiling: for each loop, the compiler marks the beginning of each iteration and load/store instruction to be profiled; while the hardware records all data dependences between the marked instructions in a buffer. The other possible approach is to statically determine which loops to parallelize by first identifying dependent instructions using pointer analysis [4, 11, 42, 46, 50, 69], and then estimating the potential performance gain with this information.

Section 5.9 has shown that accurate profiling information leads to better performance, however, it is unrealistic to always expect perfect profiling information.

One potential solution is to dynamically adjust the inter-thread value communication strategy at runtime. In the current infrastructure, once the compiler has decided to synchronize a certain data dependence, it does so for the entire duration of execution. However, data dependence patterns of a parallelized loop may change depending on the input data and/or the phase of the program. The compiler could insert synchronization instructions for all possible data dependences and let the hardware decide which synchronization instructions to use at runtime.

Overall, we believe that the compiler, being able to analyze the entire program, and the hardware, being able to adjust to the dynamic behavior of the program, should work in tandem to optimize the efficiency of inter-thread value communication mechanisms for TLS.

Appendix A

Profiling Methodology

The compiler optimization passes make extensive use of profiling information. In this appendix, we present a detailed description of how to collect such information. Our approach is to instrument the source code by inserting calls to profiling routines next to each object that we are interested in profiling. When the instrumented program is executed, it maintains a hash table to collect profiling information and prints out the contents of this data structure at the end of the execution.

Profiling information is kept separately for each parallel region. A unique identification number is assigned to each region and the beginning and the end of the region is marked using the two profiling routines in Table A.1. When the `enter_region` routine is invoked with a region identification number, the data structure associated with the region is initialized; when the `exit_region` routine is invoked, the data structure is summarized if necessary.

Table A.1: Profiling routines that mark the beginning and the end of a region.

Profiling Routine	Usage	Parameters
`enter_region`	Marks the beginning of a region	Region identification number
`exit_region`	Marks the end of a region	Region identification number

A.1 Control Dependences

This dissertation uses control dependence profiling information to facilitate speculative instruction scheduling across conditional branches and indirect jumps. For this purpose, we assign a unique identification number to each branch/indirect jump instruction and count the number of times each branch/indirect jump instruction is executed. Two profiling routines, shown in Table A.2, are used to collect control dependence profiling information. A `branch` routine is inserted with a unique branch identification number before each branch instruction indicating that a branch is about to be executed; a `branch_target` routine is inserted at all possible targets with the branch identification number and the target identification number. A single target can be the destination of several branch instructions; however, whenever control reaches this target, it should only be counted towards the last branch taken, if there is one. Thus, at every destination, a call to the `branch_target` routine is inserted for every branch instruction that can potentially jump to this destination. To resolve this issue, we keep track of the identification number of the last branch instruction executed. A `branch_target` instruction only records a branch if the last unmatched branch identification number matches the branch identification number in its parameter.

The data structure maintained for collecting control dependence information is illustrated in Figure A.1. In this data structure, branches are kept in a hash table indexed by the branch identification number. Every entry in the table corresponds

Table A.2: Profiling routines to collect control dependence information.

Profiling Routine	**Usage**	**Parameters**
`branch`	Marks an upcoming branch	Branch Identification number
`branch_target`	Marks a target of a branch	Branch Identification number
		Target Identification number

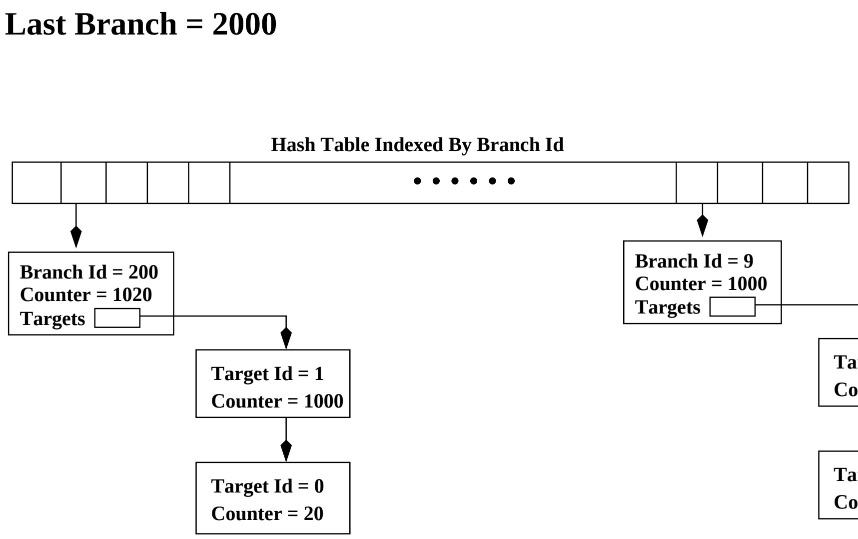


Figure A.1: Data structure maintained at runtime to keep track of branch behavior.

to a branch. It also contains a list of targets and the number of times each target is reached from that particular branch. For instance, branch #200 is executed 1020 times and has two targets. Assuming branch #200 is a conditional branch instruction and target #1 corresponds to branch taken, then this branch is a biased branch; it is taken 1000 times and falls through 20 times.

A.2 Data Dependences

This dissertation uses data dependence profiling information to identify and synchronize frequently occurring inter-thread data dependences and to facilitate speculative instruction scheduling across potential intra-thread data dependences. To collect this information, all loads and stores are instrumented to match up dependent pairs at runtime. My approach is to assign a unique identification number to each load and store instruction and insert a call to the profiling routines before the actual memory access. The profiling routines, `prof_load` and `prof_store`, are described in detail in Table A.3. Each routine takes two parameters, the address accessed and the unique identification number associated with the load/store instruction. During execution, a data structure is maintained, as illustrated in Figure A.2(a). For each memory location, all the store instructions that have stored to this address are kept track of. For each unique store, the load instructions that are dependent on it are also kept track of. Dependence distance between a load and a store instruction is calculated using an induction variable that is unique for each region. The induction variable is initialized upon entering a region and incremented at the end of each iteration (assuming every iteration of a loop corresponds to one thread of execution). When a store occurs, its corresponding entry is moved to the head of the link list, its counter is incremented, and the value of the induction variable at the time is assigned to `Last Access Iteration`. When a load accesses a memory location, the last store that has modified this location is at the head of the link list. Thus, we can update the dependence table with the newly discovered dependent pair. The distance of the dependence is the difference between the value of the current induction variable and the value of `Last Access Iteration` in the store entry.

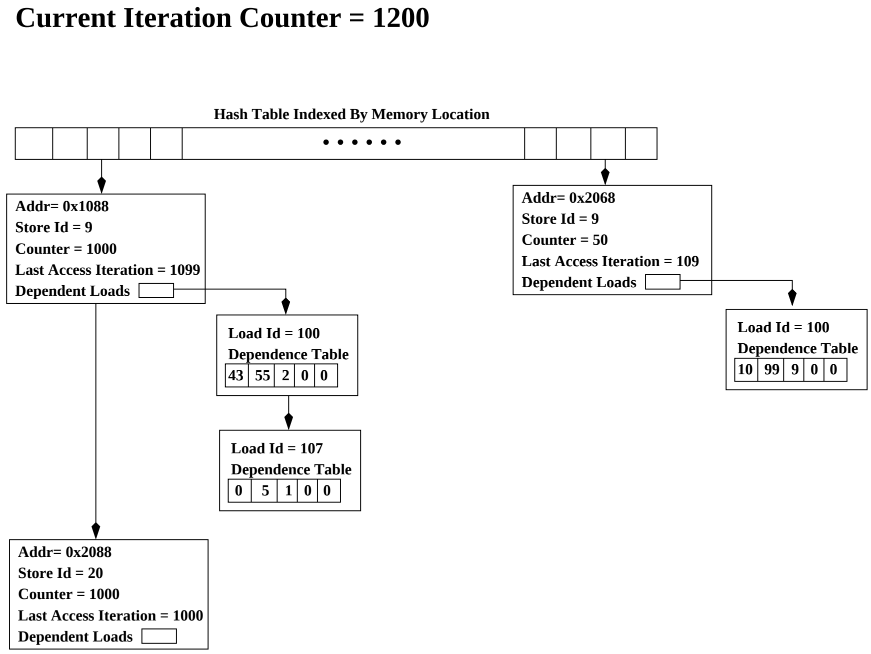
This data structure can become large and impractical to maintain during execution. Thus, the table is compressed at the end of each region. This is done by

Table A.3: Profiling routines to collect data dependence information.

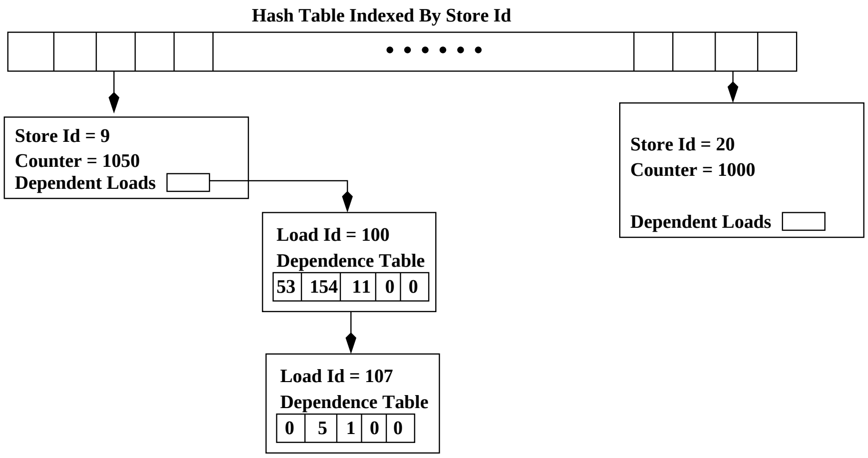
Profiling Routine	Usage	Parameters
`init_ind`	Initializes induction variable	Induction Variable Name
`inc_ind`	Increments induction variable	Induction Variable Name
`load`	Marks an upcoming load instruction	Load Identification
		Load Address
`store`	Marks an upcoming store instruction	Store Identification
		Store Address

merging all entries into a new table indexed by store identification number, as shown in Figure A.2(b). The contents of the compressed table are printed at the end of the execution.

So far, only unique identification numbers have been used to represent static branches, load and store instructions. However, there are occasions where it is desirable to collect context-sensitive profiling information, e.g., the call path leads us to the branch or memory access instruction. In such cases, each profiled object is identified not only by an identification number, but also by the call path that leads to it. To incorporate this information into the profiling framework, we assign a unique identification number to each procedure, instrument the beginning and the end of each procedure, and then maintain a calling stack during execution. When a profiled object is encountered, it can be identified not only by the unique identification number and but also by the contents of the calling stack at the time.



(a) Dependence Table Indexed by Memory Locations.



(b) Dependence Table Indexed By Store Id.

Figure A.2: Data structured maintained at runtime to keep track of data dependences.

Appendix B

Selected High Coverage Loops

This section provides a detailed description of the set of loops that are selected by our region selection algorithm and demonstrate significant speedup under TLS. For each loop that speeds up under TLS after optimization and has a greater than 5% coverage, we provide the following information:

- corresponding source file and line number;
- the coverage of the loop;
- best speedup achieved when register-resident value communication is optimized;
- best speedup achieved when memory-resident value communication is optimized.

The information is displayed separately for different benchmarks, and we display only loops that speed up by more than 5%.

Table B.1: Speedup for High Coverage Loops in GO.

Loop	**Coverage**	**Initial Speedup**	**Register-Resident Value Optimization**	**Memory-Resident Value Optimization**
g23.c:3565	5.7%	0.91	1.25	1.24

Table B.2: Speedup for High Coverage Loops in IJPEG.

Loop	**Coverage**	**Initial Speedup**	**Register-Resident Value Optimization**	**Memory-Resident Value Optimization**
jdhuff.c:576	6.2%	1.00	2.92	2.92
jidctint.c:373	7.6%	0.94	2.74	2.74
jccolor.c:138	10.3%	1.19	3.08	3.08
jchuff.c:468	15.2%	1.00	3.05	3.05
jdmerge.c:304	5.4%	1.13	2.59	2.59
jcdctmgr.c:207	22.4%	0.99	2.17	2.17

Table B.3: Speedup for High Coverage Loops in GZIP_DECOMP.

Loop	**Coverage**	**Initial Speedup**	**Register-Resident Value Optimization**	**Memory-Resident Value Optimization**
inflate.c:513	99%	0.60	0.66	1.09

Table B.4: Speedup for High Coverage Loops in VPR.

Loop	**Coverage**	**Initial Speedup**	**Register-Resident Value Optimization**	**Memory-Resident Value Optimization**
place.c:902	11.6%	1.03	1.17	1.10
place.c:951	62.2%	0.86	1.52	1.44
place.c:501	99.2%	0.84	0.84	1.10

Table B.5: Speedup for High Coverage Loops in MCF.

Loop	**Coverage**	**Initial Speedup**	**Register-Resident Value Optimization**	**Memory-Resident Value Optimization**
mcfutil.c:80	72.2%	1.17	1.20	1.05
pbeampp.c:186	14.9%	1.94	2.47	2.20
mcfutil.c:80	7.9%	0.63	1.12	1.13

Table B.6: Speedup for High Coverage Loops in PARSER.

Loop	**Coverage**	**Initial Speedup**	**Register-Resident Value Optimization**	**Memory-Resident Value Optimization**
parse.c:290	43.1%	1.03	1.01	1.03

Table B.7: Speedup for High Coverage Loops in PERLBMK.

Loop	**Coverage**	**Initial Speedup**	**Register-Resident Value Optimization**	**Memory-Resident Value Optimization**
toke.c:5597	10.4%	1.03	1.39	0.98

Table B.8: Speedup for High Coverage Loops in BZIP_COMP.

Loop	**Coverage**	**Initial Speedup**	**Register-Resident Value Optimization**	**Memory-Resident Value Optimization**
bzip2.c:1253	6.4%	0.91	1.31	1.41
bzip2.c:2296	5.7%	1.11	2.55	2.55
bzip2.c:2186	5.1%	0.95	1.10	0.99

Table B.9: Speedup for High Coverage Loops in BZIP_DECOMP.

Loop	**Coverage**	**Initial Speedup**	**Register-Resident Value Optimization**	**Memory-Resident Value Optimization**
bzip2.c:2637	13.3%	1.45	2.48	2.48

Table B.10: Speedup for High Coverage Loops in TWOLF.

Loop	**Coverage**	**Initial Speedup**	**Register-Resident Value Optimization**	**Memory-Resident Value Optimization**
dimbox.c:82	6.9%	0.93	1.15	1.15

Table B.11: Speedup for High Coverage Loops in SWIM.

Loop	**Coverage**	**Initial Speedup**	**Register-Resident Value Optimization**	**Memory-Resident Value Optimization**
swim.f:316	41.3%	1.06	2.87	2.87
swim.f:115	29.8%	0.99	3.30	3.30
swim.f:262	28.8%	1.05	2.36	2.36

Table B.12: Speedup for High Coverage Loops in MGRID.

Loop	**Coverage**	**Initial Speedup**	**Register-Resident Value Optimization**	**Memory-Resident Value Optimization**
mgrid.f:364	17.2%	1.00	2.31	2.31
mgrid.f:191	62.7%	1.49	1.53	1.67
mgrid.f:234	9.6%	1.04	2.79	2.79

Table B.13: Speedup for High Coverage Loops in MESA.

Loop	**Coverage**	**Initial Speedup**	**Register-Resident Value Optimization**	**Memory-Resident Value Optimization**
osmesa.c:693	5.6%	1.73	1.86	1.86
vbrender.c:905	85.9%	1.01	1.21	1.44

Table B.14: Speedup for High Coverage Loops in ART.

Loop	**Coverage**	**Initial Speedup**	**Register-Resident Value Optimization**	**Memory-Resident Value Optimization**
scanner.c:409	5.1%	1.06	1.23	1.23
scanner.c:471	20.6%	1.01	2.44	2.41
scanner.c:479	54.5%	0.77	0.80	1.42

Table B.15: Speedup for High Coverage Loops in EQUAKE.

Loop	**Coverage**	**Initial Speedup**	**Register-Resident Value Optimization**	**Memory-Resident Value Optimization**
quake.c:317	96%	1.04	1.75	1.78

Table B.16: Speedup for High Coverage Loops in AMMP.

Loop	**Coverage**	**Initial Speedup**	**Register-Resident Value Optimization**	**Memory-Resident Value Optimization**
eval.c:347	6.8%	0.94	1.94	1.94
atoms.c:178	83.2%	0.97	0.95	1.75

Appendix C

Estimating the Performance Upper Bounds for Value Communication Optimizations on All Loops

This section determines various optimization parameters using the evaluation techniques described in Section 5.3.

C.1 Avoiding Frequently Occurring Speculation Failures

Since it is neither realistic nor desirable to eliminate all speculation failures, we evaluate the impact of predicting values only for frequently occurring data dependences. The results are shown in Figure C.1. In each graph, the x-axis is the speedup with a perfect value predictor for register-resident values and loads that frequently depend on previous threads,[1] and the y-axis is the speedup when all register-resident values and memory-resident values are perfectly predicted. The benchmarks respond very differently: (i) for some benchmarks, such as MCF, SWIM and ART, only perfectly predicting values for frequently occurring dependences is as good as perfectly predicting values for all dependences; (ii) for some other benchmarks, such as GO, GCC, and EQUAKE, perfectly predicting values for all dependences performs better than only perfectly predicting values for frequently occurring dependences; (iii) for still other benchmarks, such as GZIP and BZIP2 (both compression and decompression phases), only perfectly predicting values for frequently occurring loads performs better than predicting values for all dependences. This counter-intuitive behavior is the result of changing data cache access patterns when violations are avoided. Unfortunately, this phenomenon cannot be explored within the scope of this dissertation. Overall, with a large number of loops located close to the 45-degree reference line, we are able to conclude that being able to capture only frequently occurring data dependences still offers a significant enough performance improvement for all benchmarks.

C.2 Avoiding Speculation Failures on the First Occurrences of Loads

Since the synchronization mechanisms supported by the underlying hardware allow for only one value to be communicated between two designated program points, load instructions that reside in inner loops and access multiple memory locations in a single thread cannot be synchronized on all occurrences. In other words, the underlying hardware is unable to forward a stream of values between two threads. To evaluate the impact of this disability, a perfect value predictor that predicts values only for

[1] Loads that depend on previous threads in more than 1% of all threads are considered frequent in this experiment.

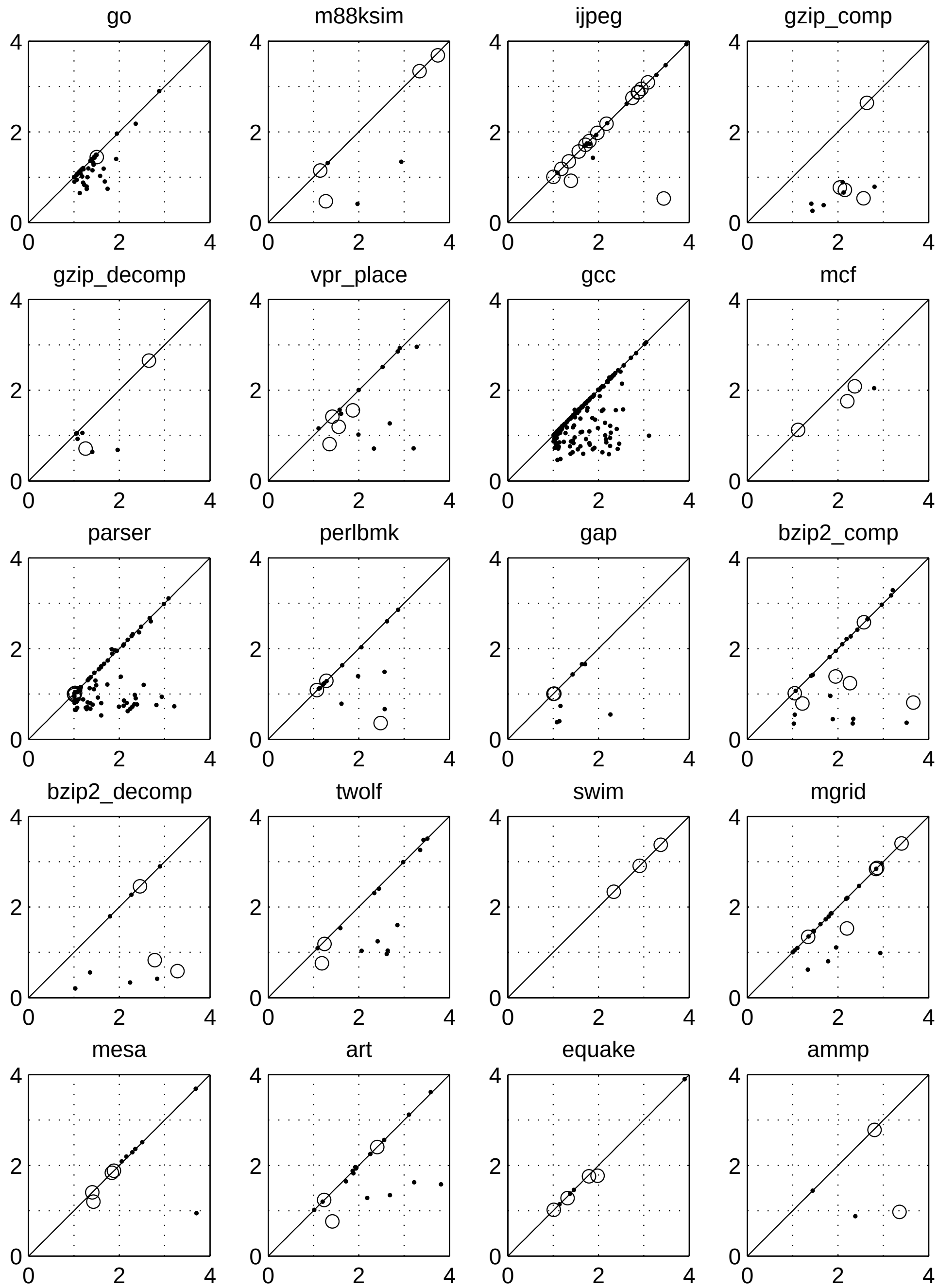


Figure C.1: **Impact of only avoiding speculation failures caused by loads of frequently occurring inter-thread data dependences.** In each graph, the x-axis is the speedup with a perfect value prediction for register-resident values and loads that depend on previous threads in more than 1% of all threads, and the y-axis is the speedup with a prefect value predictor for all register-resident values and memory-resident values. Loops that do not speed up in both cases are omitted from the graph for clarity.

loads on their first occurrence in a thread is implemented. The results of this study are shown in Figure C.2. In each graph, the x-axis is the speedup with a perfect value predictor for register-resident values and loads of frequently occurring data dependences on the first occurrences of these loads; and the y-axis is the speedup with a perfect value predictor for register-resident values and loads of frequently occurring data dependences on all occurrences. The results show that although a few loops are located above the 45-degree reference line, for a large majority of loops, this disability does not reduce parallelism.

C.3 Impact of Optimizing Data Dependences in Callee Procedures

Although all instructions that could cause dependences between register-resident values are located in the body of the parallelized loop, instructions that could cause dependences between memory-resident values could be located in either the parallelized loop or in the procedures called from the loop. This experiment tries to determine how deep into to the call tree we have to search to find the loads of all frequently occurring dependences. This experiment uses a prefect values predictor that predicts only the value for a load if that load can be reached from the parallelized loop with at most ten, five or zero levels of procedural calls. The results are shown in Figure C.3, Figure C.3 and Figure C.3. We observe that if loads of frequently occurring dependences in the callee procedures are not taken into consideration, the performance degrades for a significant number of loops, as shown in figure C.3, however, we also observe that only optimizing loads that can be reached with five levels of procedural calls is as good as optimizing all loads, as shown in Figure C.3. Thus, we conclude, although it is necessary to search for loads of frequently occurring dependences in the callee

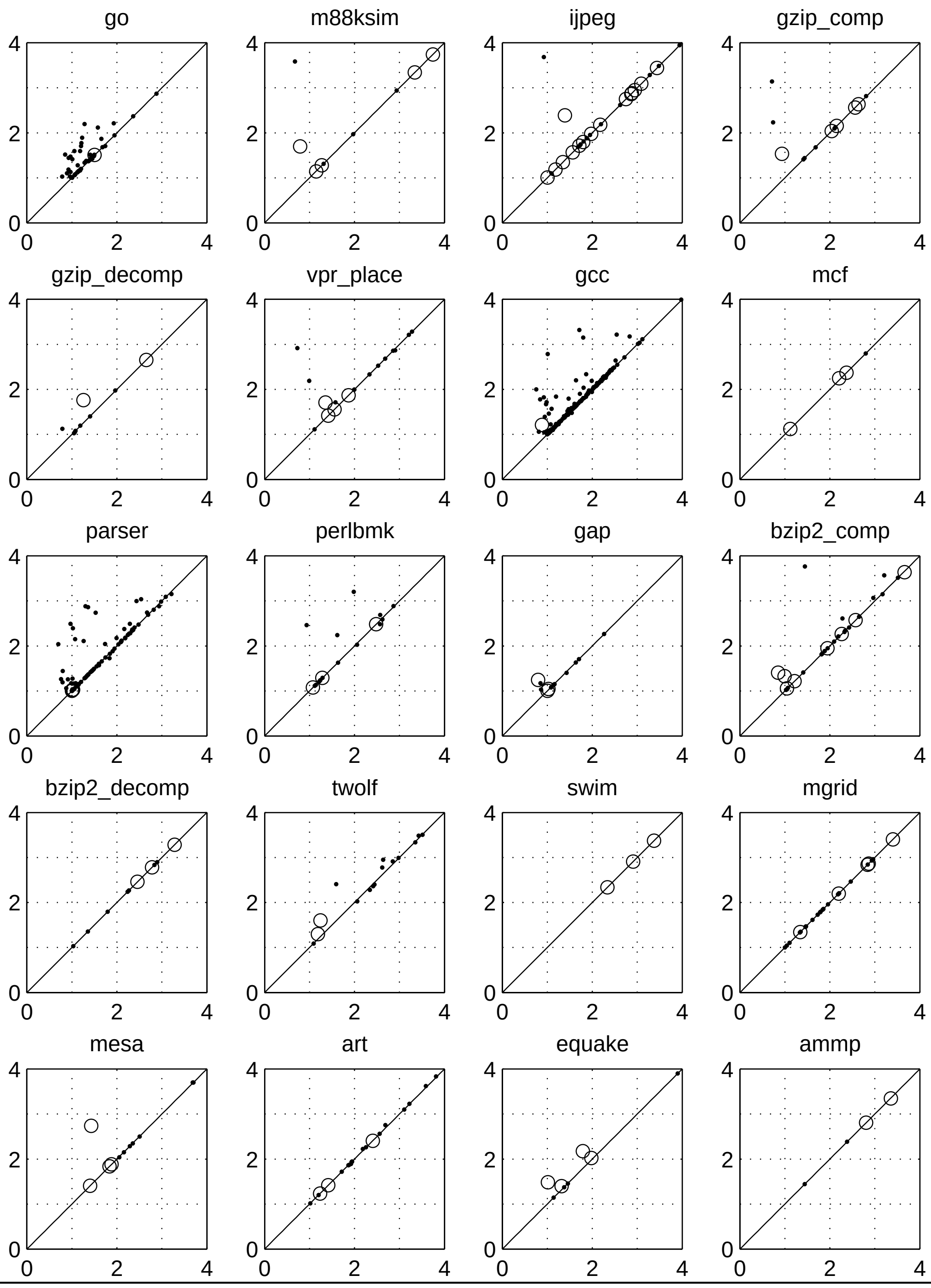


Figure C.2: **Impact of only perfectly predicting values for loads on its first occurrence within a thread.** In each graph, the x-axis is the speedup with a perfect value predictor for register-resident values and loads of frequently occurring data dependences on the first occurrences of these loads, and the y-axis is the speedup with a perfect value predictor for register-resident values and loads of frequently occurring data dependences on all occurrences. Loops that do not speed up in both cases are omitted from the graph for clarity.

procedures, search only five levels into the call tree is enough for most benchmarks.

C.4 Impact of Optimizing Distance One Dependences

The compiler-based synchronization insertion pass described in Section 3.5 forwards and synchronizes data dependences only between two consecutive threads, which are referred to as data dependences of distance one. The assumption that this optimization is able to capture almost all data dependence violations is verified by the results shown in Figure C.4. In each graph, the x-axis is the speedup with a perfect value predictor for register-resident values and loads that frequently depend on its immediate predecessor (e.g. distance one dependences); and the y-axis is the speedup with a perfect value predictor for register-resident values and loads that frequently depend on any of its predecessors. Predicting only loads that frequently suffer distance one violations is found to be as good as predicting loads that frequently suffer violations of all distances.

C.5 Search for the Threshold of Frequently Occurring Data Dependences

In all the above experiments, a data dependence is defined as frequently occurring if it occurs in more than 1% of all threads. To determine if 1% is a good threshold, a set of experiments were conducted to quantify the threshold for frequently occurring data dependence. In these experiments, a prefect value predictor was used to predict only loads that depend on previous threads in more than a certain percentage of threads. The performance was measured with the threshold set to 2%, 4%, 8%, and 16% against

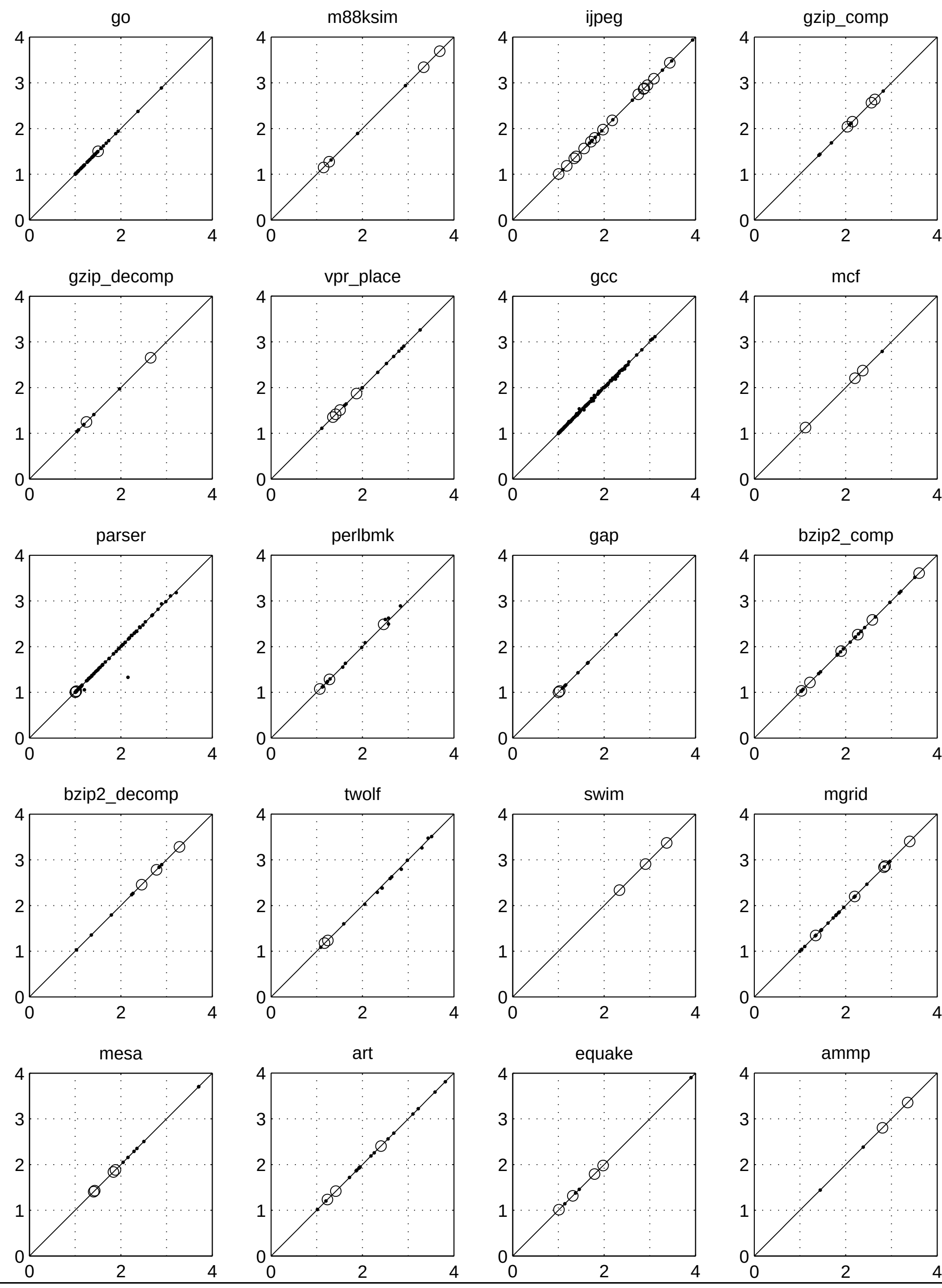


Figure C.3: **Impact of optimizing loads in the callee procedures.** In each graph, the x-axis is the speedup with a perfect value predictor for all register-resident values and for loads of frequently occurring dependences regardless of their location, and the y-axis is the speedup with a perfect value predictor for all register-resident values and loads of frequently occurring dependences that can be reached from the parallelized loop with at most ten levels of procedural calls. Loops that do not speed up in both cases are omitted from the graph for clarity.

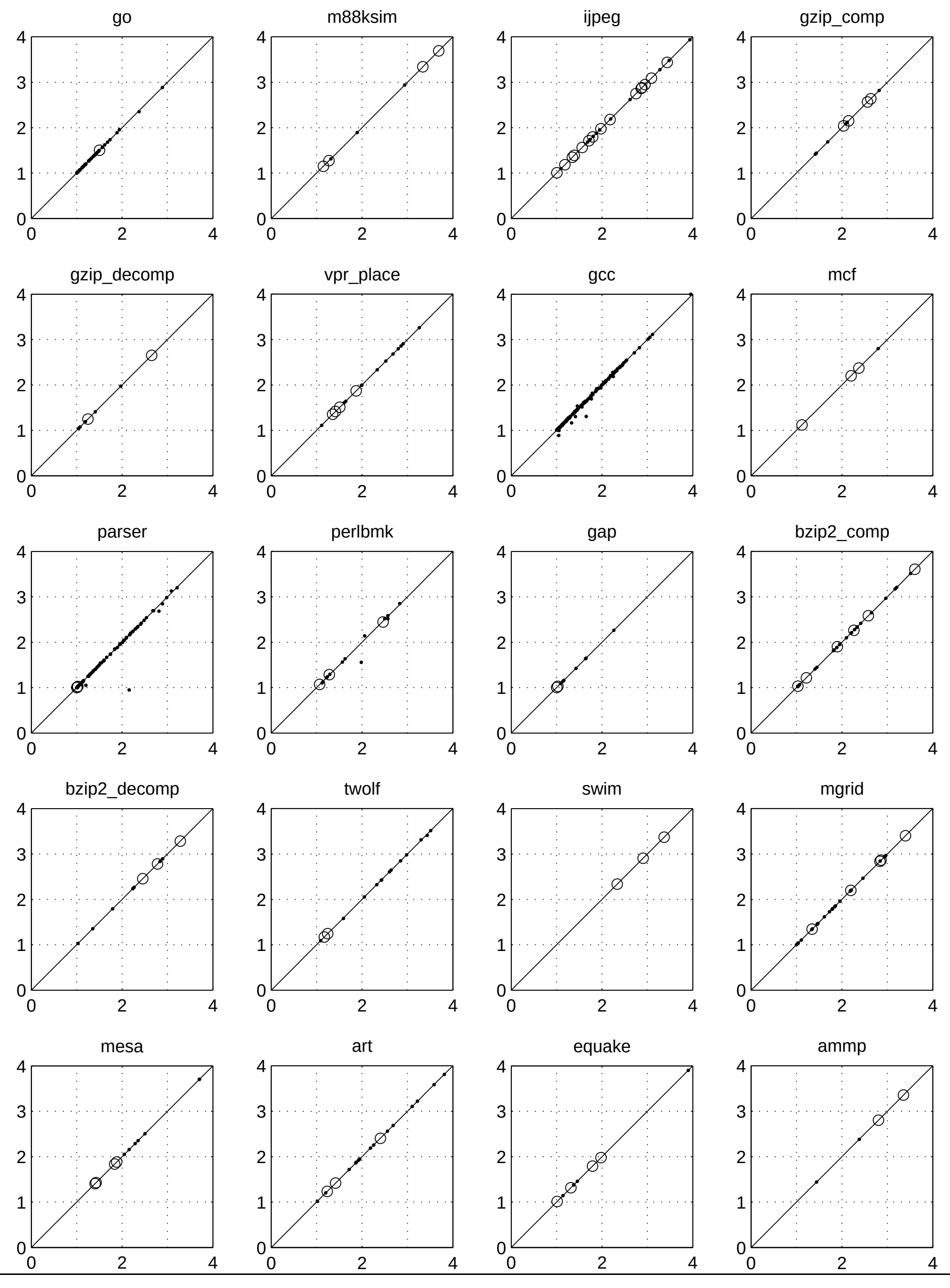


Figure C.4: **Impact of optimizing loads in the callee procedures.** In each graph, the x-axis is the speedup with a perfect value predictor for all register-resident values and for loads of frequently occurring dependences regardless of their location, and the y-axis is the speedup with a perfect value predictor for all register-resident values and loads of frequently occurring dependences that can be reached from the parallelized loop with at most five levels of procedural calls. Loops that do not speed up in both cases are omitted from the graph for clarity.

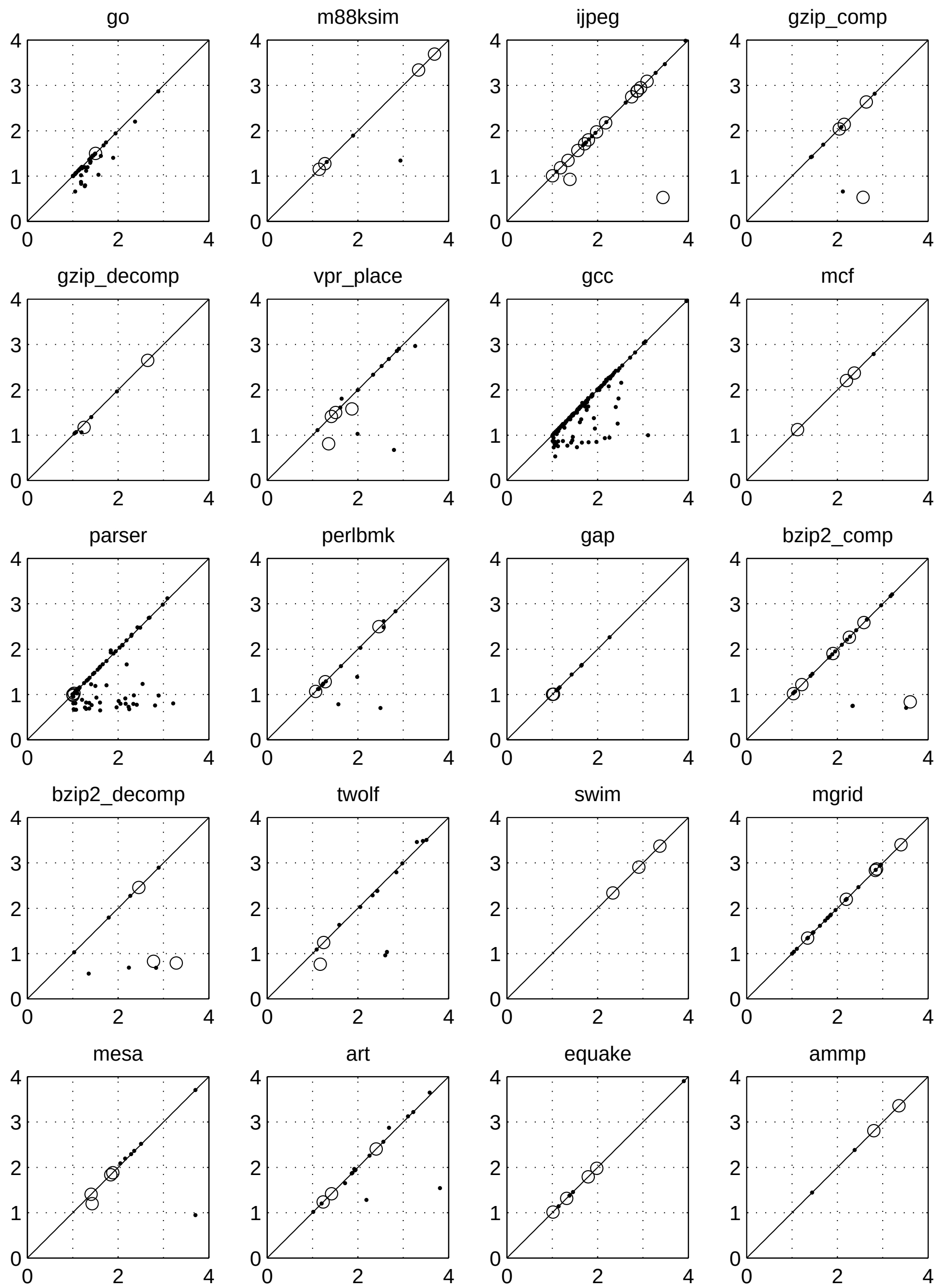


Figure C.5: **Impact of optimizing loads in the callee procedures.** In each graph, the x-axis is the speedup with a perfect value predictor for all register-resident values and for loads of frequently occurring dependences regardless of their location, and the y-axis is the speedup with a perfect value predictor for all register-resident values and loads of frequently occurring dependences that can be reached from the parallelized loop without going through any procedural calls. Loops that do not speed up in both cases are omitted from the graph for clarity.

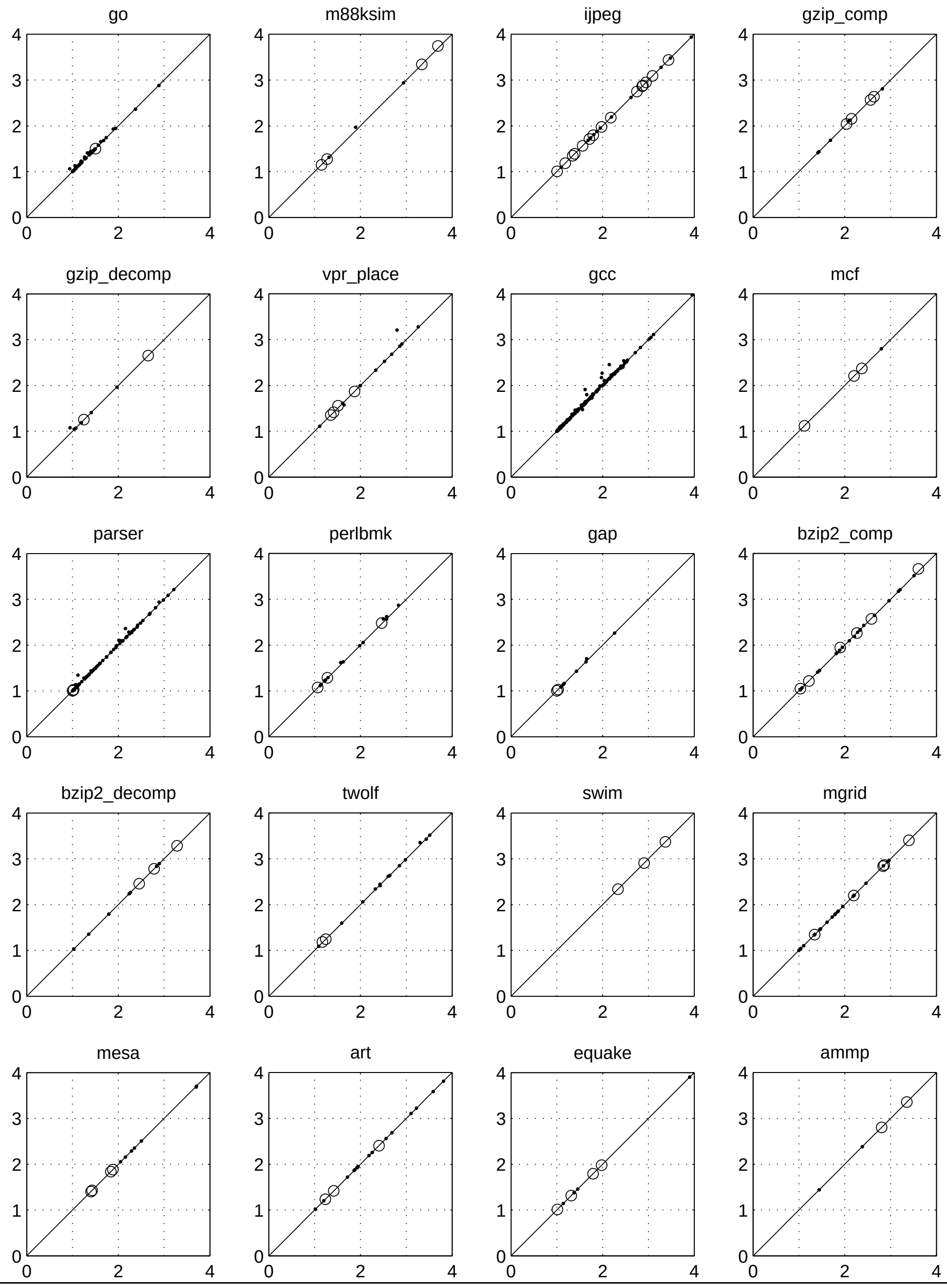


Figure C.6: **Impact of only optimizing loads of distance one.** In each graph, the x-axis is the speedup with a perfect value predictor for register-resident values and loads that frequently depend on its immediate predecessor (e.g., distance one dependences), and the y-axis is the speedup with a perfect value predictor for register-resident values and loads that frequently depend on all of its predecessors. Loops that do not speed up in both cases are omitted from the graph for clarity.

the case when the threshold is set to 1%. The results are shown in Figures C.5, C.5, C.5 and C.5. Across the four graphs, we see that for most benchmarks, only perfectly predicting loads that cause violations in 16% of all threads is as good as perfectly predicting loads that occur in more than 1% of all threads. This indicates that data dependence behavior follows a bi-model pattern—they either occur very frequently or very infrequently. However, when the threshold is increased from 4% and 8%, the performance for a few high coverage loops starts to degrade, as shown in Figures C.5 and C.5. Taking all these observations into account, 4% is shown to be a good compromise.

C.6 Impact of False Sharing

The underlying hardware used in this study tracks inter-thread data dependences at cache line granularity, and thus false sharing could cause speculation to fail even when there is no real data dependence between two threads. The impact of false sharing is shown in Figure C.6. In this figure, the x-axis is the speedup with a perfect value predictor for loads of all real frequently occurring dependences; the y-axis is the speedup with a perfect value predictor for loads of all real frequently occurring dependences and false sharing. The results show that although most benchmarks are not affected by false sharing, some benchmarks, such as GCC, BZIP2_COMP and BZIP2_DECOMP, could lose speedup loops if false sharing is not handled properly. Although the compiler-based techniques cannot handle such cases, hardware-based techniques are able to capture such dependences (see detailed evaluation in Section 5.8).

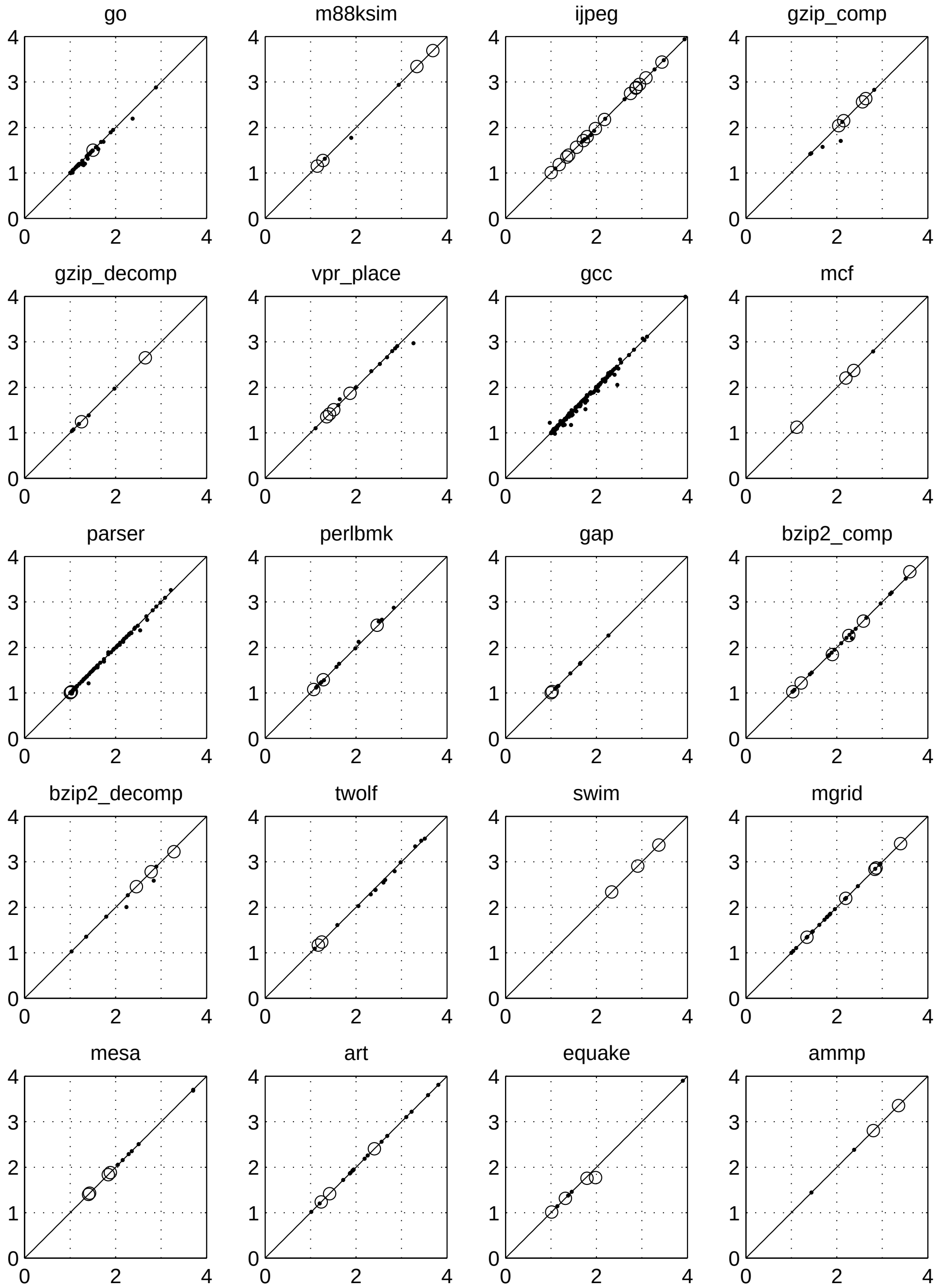


Figure C.7: **Impact of optimizing loads with different dependence frequencies.** In each graph, the x-axis is the speedup with a perfect value predictor for all dependences occurring in more than 1% of all threads, the y-axis is the speedup with a perfect value predictor for all dependences occur in more than 2% of all threads; Loops that do not speed up in both cases are omitted from the graph for clarity.

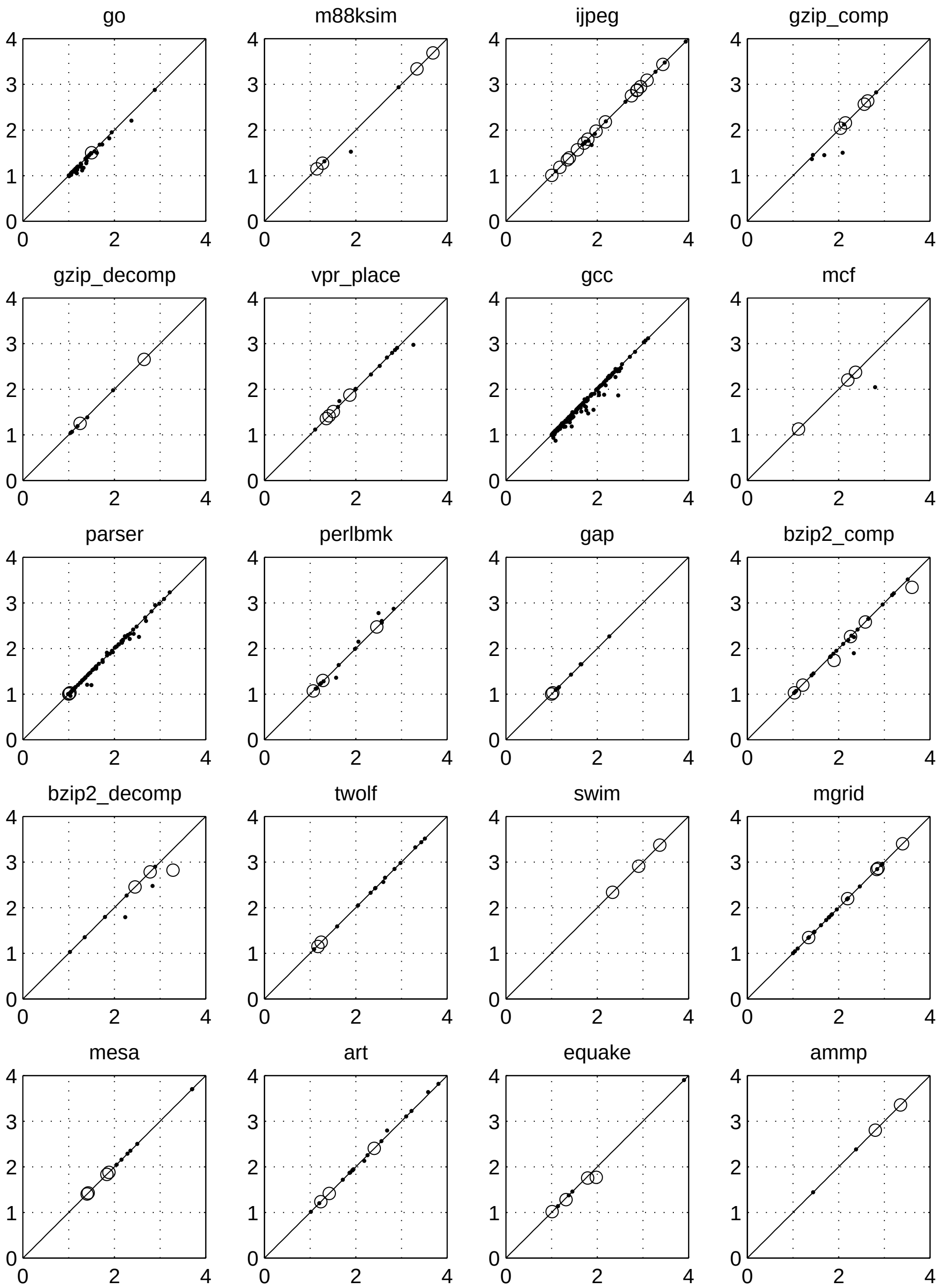


Figure C.8: **Impact of optimizing loads with different dependence frequencies.** In each graph, the x-axis is the speedup with a perfect value predictor for all dependences occurring in more than 1% of all threads, and the y-axis is the speedup with a perfect value predictor for all dependences occurring in more than 4% of all threads. Loops that do not speed up in both cases are omitted from the graph for clarity.

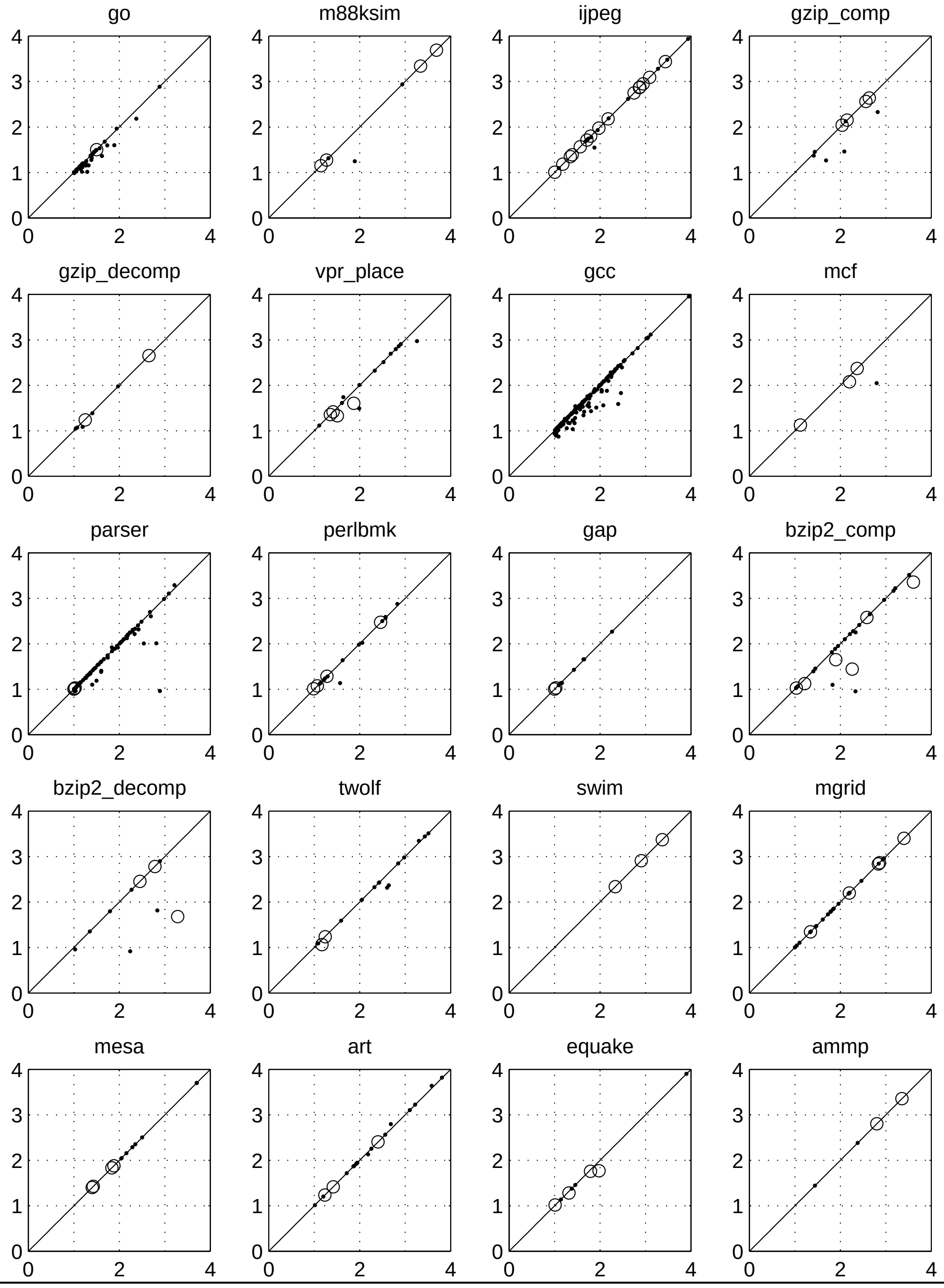


Figure C.9: **Impact of optimizing loads with different dependence frequencies.** In each graph, the x-axis is the speedup with a perfect value predictor for all dependences occurring in more than 1% of all threads, the y-axis is the speedup with a perfect value predictor for all dependences occurring in more than 8% of all threads. Loops that do not speed up in both cases are omitted from the graph for clarity.

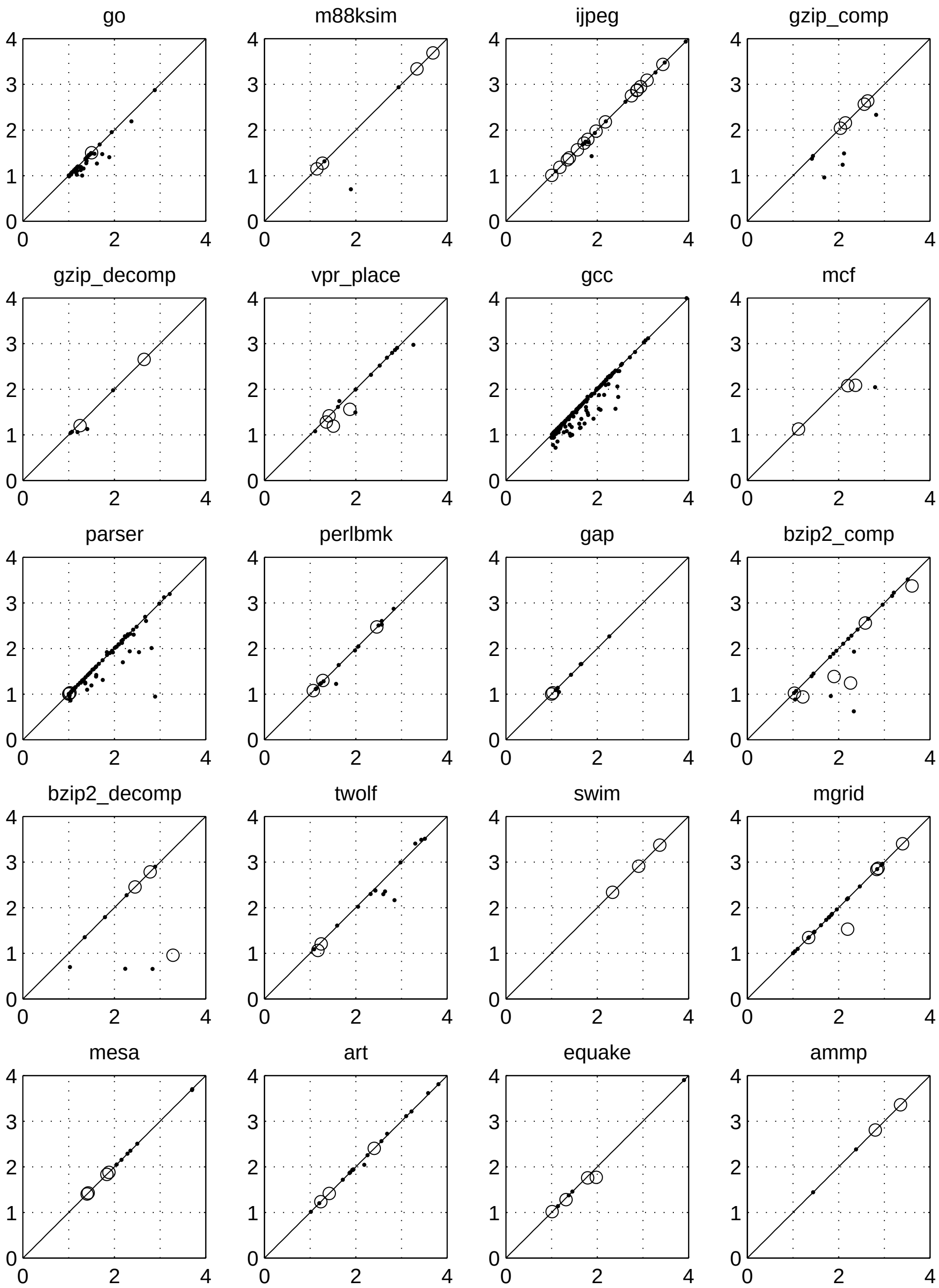


Figure C.10: **Impact of optimizing loads with different dependence frequencies.** In each graph, the x-axis is the speedup with a perfect value predictor for all dependences occurring in more than 1% of all threads, the y-axis is the speedup with a perfect value predictor for all dependences occurring in more than 16% of all threads. Loops that do not speed up in both cases are omitted from the graph for clarity.

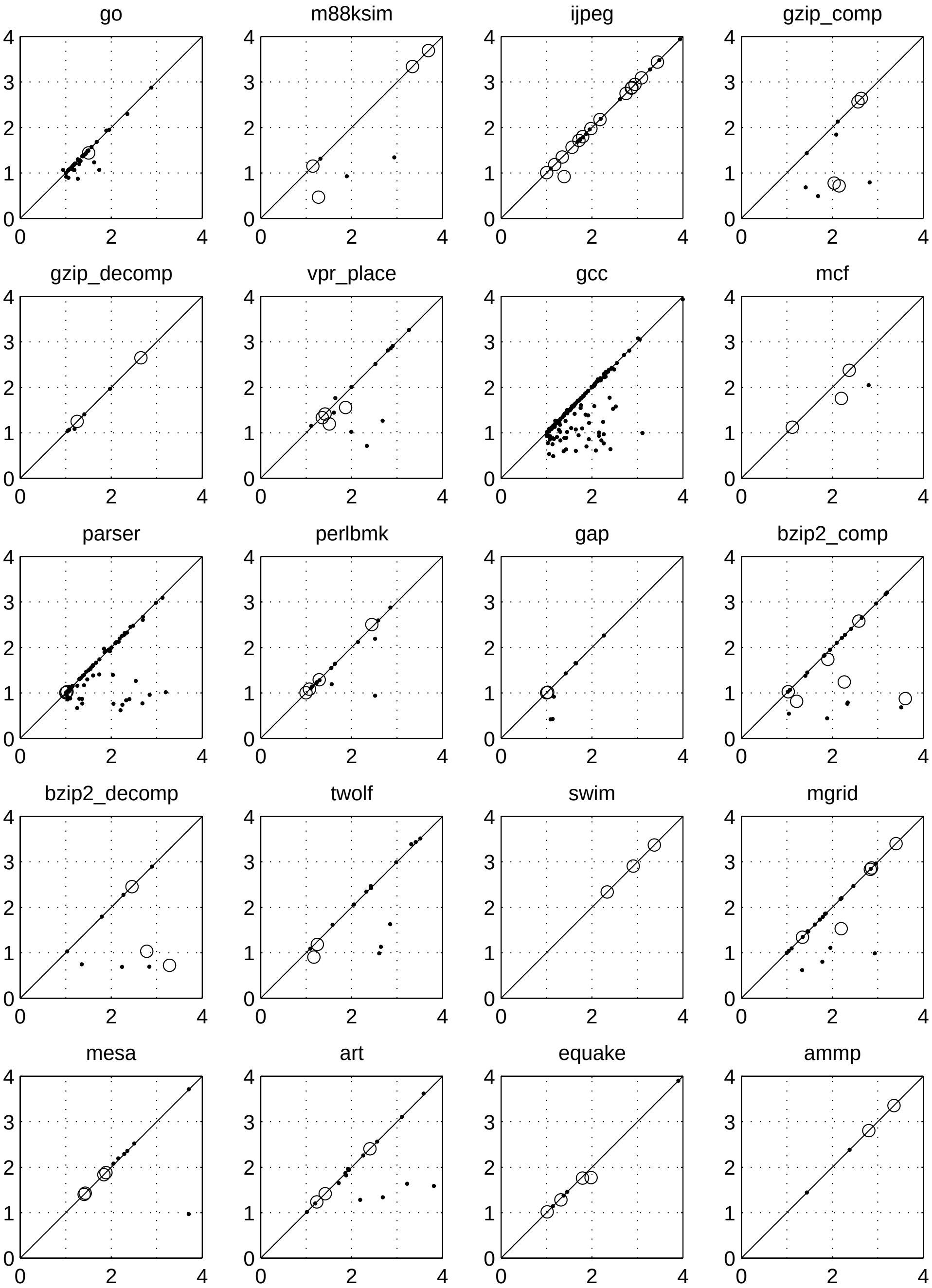


Figure C.11: **Impact of avoiding speculation failures caused by false sharing.** In each graph, the x-axis is the speedup with a perfect value predictor for loads of all real frequently occurring dependence, the y-axis is the speedup with a perfect value predictor for loads of all real frequently occurring dependence and false sharing. Loops that do not speed up in both cases are omitted from the graph for clarity.

Appendix D

Loops with No Inter-Thread Data Dependences for Memory-Resident Values

In this appendix, we attempt to determine the coverage of the set of loops that do not contain any inter-thread dependences; for these loops, the hardware support for recovering from speculative failures is never invoked. The coverage of these loops represents the upper bound of the fraction of loops that can be parallelized without either hardware or compiler support to satisfy inter-thread data dependences for memory-resident values.[1]

To conduct this experiment, we first identify all loops that do not contain any data dependences for memory-resident values of distance 1, 2 or 3 (e.g., data dependences between two consecutive threads is of distance 1). From these loops, we then select a new set of loops using the same algorithm that has been used to select the *register set*. We refer to this set of loops as the *no speculation set*. Table D.1 shows the coverage

[1]We assume inter-thread data dependences for register-resident scalars are handled by the compiler.

of this set of loops as well as the coverages of the *register set*, the *idealistic set* and the *realistic set*. We observe that the coverage of the *no speculation set* is much smaller than that of the other sets, which indicates that by allowing speculative execution, we would be able to parallelize a much larger set of loops without complicated compiler analyses.

Table D.1: Fraction of execution being parallelized.

Benchmark	**Register Set**	**Idealistic Set**	**Realistic Set**	**No Speculation Set**
099.go	17.9%	93.0%	24.5%	4.9%
124.m88ksim	6.1%	97.9%	13.5%	4.8%
132.ijpeg	84.5%	84.5%	97.6%	60.4%
164.gzip_comp	10.3%	99.9%	47.3%	10.3%
164.gzip_decomp	32.0%	99.6%	99.6%	31.6%
175.vpr_place	72.6%	99.7%	73.1%	74.0%
176.gcc	26.6%	88.4%	34.9%	10.1%
181.mcf	96.8%	96.8%	96.8%	9.8%
197.parser	56.7%	89.8%	88.5%	51.7%
253.perlbmk	20.6%	18.9%	18.9%	15.4%
254.gap	51.9%	94.0%	53.1%	54.7%
256.bzip2_comp	68.2%	70.4%	70.5%	7.0%
256.bzip2_decomp	13.5%	99.7%	100.0%	0.1%
300.twolf	7.3%	100.0%	100.0%	0.0%
171.swim	99.8%	99.9%	99.9%	70.0%
172.mgrid	99.1%	88.8%	99.4%	82.6%
177.mesa	87.3%	99.0%	99.0%	7.1%
179.art	46.3%	100.0%	100.0%	45.5%
183.equake	100.0%	100.0%	100.0%	3.8%
188.ammp	6.8%	94.8%	94.8%	0.0%

Bibliography

[1] H. Akkary and M. Driscoll. A dynamic multithreading processor. In *MICRO-31*, December 1998.

[2] Glenn Ammons and James R. Larus. Improving data-flow analysis with path profiling. In *Proc. ACM SIGPLAN 98 Conference on Programming Language Design and Implementation*, 1998.

[3] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Norwell, Mass., 1988.

[4] Anasua Bhowmik and Manoj Franklin. A fast approximate interprocedural analysis for speculative multithreading compiler. In *17th Annual ACM International Conference on Supercomputing*, 2003.

[5] 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.

[6] P. P Chang, N. J. Warter, S. A. Mahlke, W. Y. Chen, and W. W. Hwu. Three superblock scheduling models for superscalar and superpipelined processors. Technical Report CRHC-91-29, Center for Reliable and High-Performance Computing, University of Illinois, Urbana-Champaign, 1991.

[7] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Nancy J. Water, and Wen mei W. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th ISCA*, May 1991.

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

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

[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] P.-S. Chen, M.-Y. Hung, Y.-S. Hwang, R. Ju, and J. K. Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *ACM SIGPLAN 2003 Symposium on Principles and Practice of Parallel Programming*, 2003.

[12] T. Chen, J. Lin, X. Dai, W.-C. Hsu, and P.-C.Yew. Data dependence profiling for speculative optimization. In *13th International Conference on Compiler Construction*, Barcelona, Spain, March 2004.

[13] George Chrysos and Joel Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th ISCA*, June 1998.

[14] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th ISCA*, June 2000.

[15] Marcelo Cintra and Josep Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *Proceedings of the 8th HPCA*, Feb 2002.

[16] C. B. Colohan, J. G. Steffan, A. Zhai, and T. C. Mowry. The impact of thread size and selection on the performance of thread-level speculation, 2004. Ongoing Work.

[17] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth, and Paul K. Rodman. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers*, 37:967 – 979, August 1988.

[18] Broadcom Corporation. The Sibyte SB-1250 Processor. http://www.sibyte.com/mercurian.

[19] Intel Corportation. Hyper-threading technology on the intel xeon processor family for servers. http://www.intel.com/technology/hyperthread/.

[20] Intel Corportation. Intel pentium 4 processor with ht technology. http://www.intel.com/personal/products/pentium4/hyperthreading.htm.

[21] R. Cytron. DOACROSS: Beyond vectorization for multiprocessors. In *International Conference on Parallel Processing*, 1986.

[22] J. Emer. Ev8: The post-ultimate *alpha.*(keynote address). In *International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[23] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 13, June 1981.

[24] M. Frank, C. A. Moritz, B. Greenwald, S. Amarasinghe, and A. Agarwal. Suds: Primitive mechanisms for memory dependence speculation. Technical Report MIT-LCS-TM-591, MIT/LCS, January 1999.

[25] M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin – Madison, 1993.

[26] M. Franklin and G. S. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5), May 1996.

[27] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the 6th ASPLOS*, pages 183–195, October 1994.

[28] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy threads: Implementing a fast parallel call. *journal of Parallel and Distributed Computing*, 37(1):5–20, August 1996.

[29] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative versioning cache. In *Proceedings of the 4th HPCA*, February 1998.

[30] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. Technical Report 1334, Computer Sciences Department, University of Wisconsin-Madison, July 1997.

[31] M. Gupta and R. Nim. Techniques for speculative run-time parallelization of loops. In *Supercomputing '98*, November 1998.

[32] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of ASPLOS-VIII*, October 1998.

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

[34] L. Howard Holley and Barry k. Rosen. Qualified data flow problems. *IEEE Transactions on Software Engineering*, 7(1), January 1981.

[35] J. Kahle. Power4: A Dual-CPU Processor Chip. *Microprocessor Forum '99*, October 1999.

[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] Jens Knoop and Oliver Ruthing. Lazy code motion. In *Proc. ACM SIGPLAN 92 Conference on Programming Language Design and Implementation*, 92.

[38] Kevin Krewell. UltraSPARC IV Mirrors Predecessor. *Microprocessor Report*, November 2003.

[39] V. Krishnan and J. Torrellas. The need for fast communication in hardware-based speculative chip multiprocessors. In *Proceedings of PACT '99*, October 1999.

[40] Venkata Krishnan and Josep Torrellas. A chip multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers, Special Issue on Multithreaded Architecture*, September 1999.

[41] M. Lipasti, C. Wilkerson, and J. Shen. Value locality and load value prediction. In *Proceedings of the 7th ASPLOS*, Boston, MA, October 1996.

[42] V. B. Livshits and M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in c programs. In *11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-11)*, September 2003.

[43] Pedro Marcuello and Antonio Gonzalez. Clustered speculative multithreaded processors. In *15th Annual ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[44] Pedro Marcuello, Jordi Tubella, and Antonio Gonzalez. Value prediction for speculative multithreaded architectures. In *Proceedings of Micro-32*, Haifa, Israel, November 1999.

[45] Samuel P. Midkiff and David A. Padua. Compiler algorithms for synchronization. *IEEE Transactions on Computers*, C-36(12):1485–1495, 1987.

[46] Markus Mock, Manuvir Das, Craig Chambers, and Susan J. Eggers. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *SIGSOFT workshop on on Program analysis for software tools and engineering Snowbird*, June 2001.

[47] Andreas I. Moshovos, Scott E. Breach, T.N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th ISCA*, June 1997.

[48] M.Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[49] A. Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, 38:663–678, May 1989.

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

[51] J. Oplinger, D. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *Proceedings of PACT '99*, October 1999.

[52] D. Padua, D. Kuck, and D. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computing*, September 1980.

[53] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed System*, 10(2):160–172, 1999.

[54] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of Micro 30*, 1997.

[55] Michael Schlansker and Vinod Kathail. Critical path reduction for scalar programs. In *Proceedings of Micro-28*, 1995.

[56] S.Horwiz, T.Reps, and D.Binkley. Interprocedural slicing using dependence graph. *ACM Trans. on Programming Languages and Systems*, 12(1):26–60, January 1990.

[57] Micheal Smith, Mark Horowitz, and Monica Lam. Efficient superscalar performance through boosting. In *Proceedings of the 5th ASPLOS*, Boston, MA, October 1992.

[58] G. S. Sohi, S. Breach, and T.N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd ISCA*, June 1995.

[59] Standard Performance Evaluation Corporation. The SPEC Benchmark Suite. http://www.specbench.org.

[60] J. G. Steffan. *Hardware Support for Thread-Level Speculation*. PhD thesis, Carnegie Mellon University, 2003.

[61] 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.

[62] J. G. Steffan, C. B. Colohan, and T. C. Mowry. Extending cache coherence to support thread-level data speculation on a single chip and beyond. Technical Report CMU-CS-98-171, School of Computer Science, Carnegie Mellon University, November 1998.

[63] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th ISCA*, June 2000.

[64] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *Proceedings of the 8th HPCA*, February 2002.

[65] 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.

[66] M. Tremblay. MAJC: Microprocessor Architecture for Java Computing. *HotChips '99*, August 1999.

[67] 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.

[68] T.N. Vijaykumar. Compiling for the multiscalar architecture. In *Ph.D. Thesis*, January 1998.

[69] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proc. ACM SIGPLAN 95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

[70] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, Massachusetts, 1989.

[71] K. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, April 1996.

[72] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *Proceedings of the 10th ASPLOS*, Oct 2002.

[73] 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*, March 2004.

[74] Chuan-Qi Zhu and Pen-Chung Yew. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Transactions on Software Engineering*, 13(6):726–739, June 1987.

[75] Craig Zilles and G.S. Sohi. Master/slave speculative parallelization. In *Proceedings of Micro-35*, November 2002.

[76] G. Zilles. *Master/Slave Speculative Parallelization and Approximate Code.* PhD thesis, University of Wisconsin – Madison, 2002.