RICE UNIVERSITY

Improving the Speed vs. Accuracy Tradeoff for Simulating Shared-Memory Multiprocessors with ILP Processors

by

S.N. Murthy Durbhakula

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Sarita V. Adve, Chair
Assistant Professor in Electrical and
Computer Engineering

James B. Sinclair
Associate Professor of Electrical and
Computer Engineering

Alan L. Cox
Associate Professor of Computer Science

Houston, Texas

April, 1998

Improving the Speed vs. Accuracy Tradeoff for Simulating Shared-Memory Multiprocessors with ILP Processors

S.N. Murthy Durbhakula

Abstract

Current simulators for shared-memory multiprocessor architectures involve a large tradeoff between simulation speed and accuracy. Most simulators assume much simpler processors than the current generation of processors that aggressively exploit instruction-level parallelism (ILP). This can result in large simulation inaccuracies. A few newer simulators model current ILP processors more accurately, but are about ten times slower.

This study proposes and evaluates a new simulation technique that requires almost no compromise in accuracy and far less compromise in speed compared to the state-of-the-art. This technique uses a novel adaptation of direct execution, a methodology used widely for simulation of multiprocessors with simple processors. We develop a new simulator based on this technique, called DirectRSIM.

We compare the performance and accuracy of DirectRSIM with three other simulators – two current direct execution simulators that use a simple processor model, and RSIM, a state-of-the-art detailed simulator for multiprocessors with ILP processors. For various combinations of applications and system configurations, we find that DirectRSIM is an average of 4 times faster than RSIM with an average relative error of 1.6%. In contrast, the current direct execution simulators see large and variable errors relative to RSIM, with an average of around 40% with the best methodology and 130% for the most commonly used methodology. Despite its superior accuracy, DirectRSIM achieves a speed within a factor of 2.7 of that achieved by the current direct execution simulators with simple processors. Although the performance advantage of simple processor based simulators is still significant, it may no longer be enough to justify the errors that such simulators see in modeling the performance of shared-memory systems with state-of-the-art processors.

Acknowledgments

I would like to thank my advisor, Sarita Adve, for her guidance throughout this work. Without her motivation and support this work would not have been possible. I thank my group members Partha, Vijay and Hazim for their encouragement and advice over the last two years. I would also like to thank my other committee members, Alan Cox and Bart Sinclair, for their valuable feedback.

This thesis originated from my joint work with Vijay Pai. Many thanks are due to him for contributing heavily to this project both through substantial coding and enriching discussions over the course of our research.

Special thanks to Mahesh for working with me on the initial phase of this project.

I would also like to thank all my friends at Rice who made working at school an enjoyable and rewarding experience.

Finally, I would like to thank my parents, my brother and my sister for their unwavering encouragement and support while I spent time far away from home.

Contents

Abstract ii
Acknowledgments iii
List of Illustrations vi

1 Introduction 1
1.1 Motivation 1
1.2 Contributions 2
1.3 Organization 5

2 Background 6
2.1 Overview of Current Simulation Techniques 6
2.2 Direct Execution Simulation for Shared-Memory Architectures with Simple Processors 8
2.3 Simulators for Shared-Memory Architectures with ILP Processors . . 9
2.3.1 ILP Processors 9
2.3.2 Detailed Simulation of Shared-Memory Architectures with ILP Processors 10
2.3.3 Current approximations for Simulating Shared-Memory Architectures with ILP Processors 10

3 Direct Execution for Simulation of Multiprocessors with ILP Processors 12
3.1 Handling Non-Blocking Loads 13
3.2 Timing Simulation of other ILP Features and Mispredicted Paths . . 14
3.3 Implementation of the DirectRSIM Simulator 17
3.3.1 Application Code Instrumentation 17
3.3.2 Architectural Timing Simulator Implementation 18

4 Evaluation Methodology 23

4.1 Simulated Architectures 23
4.2 Simulators Used in the Study 24
4.3 Metrics 26
4.4 Applications 27

5 Results **29**
5.1 Simulator Accuracy 30
5.1.1 Base Configuration 30
5.1.2 Other system configurations 32
5.2 Simulator Performance 33
5.3 Detailed analysis of DirectRSIM's performance 35
5.3.1 Comparing DirectRSIM with RSIM 35
5.3.2 Comparing DirectRSIM with Simple 39

6 Related Work **42**

7 Conclusions **45**
7.1 Thesis Summary 45
7.2 Future Work 47

Bibliography **49**

A Performance Comparison of Simple Simulators **53**

B Absolute Performance of Simulators in Instructions per Second **55**

Illustrations

2.1 Classification of Simulation Techniques 7

4.1 Base system parameters 25
4.2 Application input sizes and configurations 28

5.1 Comparison of simulator accuracy and performance for the base system configuration 31
5.2 Execution time errors of simulation models. (Averages are over the absolute values of the errors.) 34
5.3 Performance analysis of the simulators. 36
5.4 Components of RSIM's simulation time and impact of DirectRSIM . 38
5.5 DirectRSIM simulation overhead and sources of slowdown relative to Simple 40

A.1 Simulation time for SPLASH-2 applications 54

B.1 Absolute performance of the simulators in kilo-instructions per second. 55

Chapter 1

Introduction

1.1 Motivation

Shared-memory multiprocessors offer significant performance increases over uniprocessors, and are a growing segment of the high performance computing market. Simulation has been the most widely used technique for evaluating new shared-memory architectures. Recent advances in processor architecture, however, force a re-evaluation of shared-memory simulation methodology. Specifically, recent processors (referred to as *ILP processors* in this thesis) exploit instruction-level parallelism (ILP) through aggressive techniques such as multiple issue, out-of-order issue, non-blocking loads, and speculative execution. Most shared-memory simulation studies reported in the literature, however, use a much simpler model of the processor, assuming single-issue, in-order issue, blocking loads, and no speculative execution (referred to as *simple processors* in this thesis) [10, 11].

Pai et al. showed that using simple-processor based simulation models to approximate ILP processor based systems can result in large inaccuracies [17]. For the most commonly used simple-processor based simulation models, Pai et al. report errors in execution time ranging from 26% to 192% for their applications and simulated systems. A less commonly used approximation based on simple processor models reduced these errors, but the errors remained large and application dependent (-8% to 73%).

In spite of the potential for large errors, the practice of using simple-processor based simulators continues. This is largely because although the more detailed ILP

simulators are more accurate, they are generally much slower than the simple processor simulators. Studies with slow simulators are also potentially subject to erroneous conclusions as they may be restricted to smaller application input sizes or fewer applications. Confronted with this dilemma of speed vs. accuracy, the shared-memory architecture community so far appears to be reconciled to accepting the errors of simple processor based simulators. For example, of the five shared-memory simulation studies in ISCA '97[1] that reported results for full applications, four studies used a processor model with in-order issue, blocking loads, and no speculative execution, and three of these studies assumed a single-issue processor.

Intuitively, simulators that model ILP processors in detail can be expected to be much slower than simple-processor based simulators because simulating an ILP processor is inherently more complex (and therefore slower) than simulating a simple processor. Additionally, over the last decade, several speed enhancing techniques have been developed for simple-processor simulators that are currently not applicable to ILP processor simulators. One such widely used technique is direct execution, which decouples functional and timing simulation. Functional simulation is done at the speed of the simulation host machine by directly executing application instructions on the host machine. For the timing simulation, typically only memory accesses are fully simulated; the timing for other instructions is computed through static instrumentation of the application program. We hypothesize that speed enhancing techniques for simple-processor simulators can also be applied to ILP processor simulators, and test our hypothesis for direct execution.

1.2 Contributions

This thesis develops and evaluates a new simulation technique based on direct execution that significantly improves the speed vs. accuracy tradeoff for simulating

[1]Proceedings of the 24th Annual International Symposium on Computer Architecture

shared-memory multiprocessors with ILP processors. We compare four simulators equivalent in every respect other than the processor simulation:

- **RSIM** – State-of-the-art detailed simulator for shared-memory multiprocessors with ILP processors.
- **DirectRSIM** – Uses our new technique for simulating ILP-based multiprocessors using direct execution.
- **Simple** – Direct execution based simulator for shared-memory multiprocessors with simple processors
- **Simple-ix** – This is identical to Simple except that to model an ILP processor of issue width i, it simulates a simple processor whose clock speed and L1 cache access time are sped by a factor of i compared to the ILP processor. This approximation has been used in a few studies, and was shown by Pai et al. to be the best current approximation when modeling ILP processors with a simple processor based model [17].

We perform our analysis for different system configurations and application programs. For these configurations and programs, our key findings are:

- **RSIM vs. simple-processor simulators** – The simple processor simulators are an average of 10 times faster than RSIM. The speed advantage, however, comes at the cost of large errors (as also reported by Pai et al. [17]). We find an error in execution time of 2% to 128% (average of 42%) with Simple-ix and 9% to 438% (average of 131%) with Simple. The errors are relative to the results reported by RSIM.
- **New simulator vs. RSIM** – DirectRSIM is an average of 4 times faster than RSIM with 1.6% error in execution time on the average (within 2% for most cases and a maximum of 6.3%).

- **Simple processor simulators vs. New simulator** – Simple and Simple-*i*x are an average of 2.7 times faster than DirectRSIM. This speed, however, again has a high cost – up to 128% error in execution time for Simple-*i*x and up to 438% error for Simple.

The above results clearly illustrate the value of the new simulation technique. DirectRSIM is much faster than, and almost as accurate as, the detailed ILP simulator. Comparisons with the more common simple simulators yield an even more striking result. The 10x speed advantage of the simple simulators over RSIM made a compelling argument for their use in spite of their potential for large errors. However, the simple simulators only have a 2.7x speed advantage relative to DirectRSIM. While this factor is still significant, it is not clear that this advantage is always worth the cost of the potential errors (up to 128% error in execution time for Simple-*i*x). Thus, our results suggest a reconsideration of the appropriate simulation methodology for shared-memory systems.

This thesis focuses on using direct execution as an optimization technique to enhance simulation speed. However, most of the discussion in this thesis also applies to simulators based on dynamic binary translation [24]. Such simulators dynamically translate the application code into optimized sequences of the native host machine code, thereby also applying a form of direct execution and sharing most of the problems and solutions posed in this thesis. Other simulation optimizations such as parallelization [18] and sampling [4] are orthogonal to the processor model, and can be used to further enhance the performance of ILP-based simulations in a manner similar to their use for simple-processor based simulations. We also consider simulations for only user-level code. The core ideas presented here should apply to system-level simulations as well.

1.3 Organization

The rest of the thesis is organized as follows. Chapter 2 provides background material on simulation methods. Chapter 3 describes the technique for using direct execution for ILP-based shared-memory multiprocessors and the DirectRSIM implementation. Chapter 4 describes the evaluation methodology. Chapter 5 presents some results. Chapter 6 describes related work. Chapter 7 concludes the thesis and discusses future work. Appendix A compares the performance of the baseline Simple simulator with Wisconsin Wind Tunnel-II, a state-of-the-art direct-execution simulator for shared memory multiprocessors with models of simple processors. Appendix B provides performance data for all the simulators studied in terms of simulated instructions per second.

Chapter 2

Background

This chapter provides a brief background on existing simulation methodologies for shared-memory systems. Section 2.1 presents a classification of current simulation techniques for shared-memory systems, Section 2.2 discusses the current use of direct execution in simulating shared-memory architectures with simple processors, and Section 2.3 discusses both detailed and approximate models for simulating shared-memory systems with state-of-the-art ILP processors.

2.1 Overview of Current Simulation Techniques

Figure 2.1 summarizes a classification of simulation techniques. Existing simulation techniques can be broadly classified as execution-driven and trace-driven simulation techniques. Execution-driven simulation refers to techniques that execute the application during the course of simulation and are influenced by the execution path taken by the application. The input to an execution-driven simulator is the executable of the application to be simulated. With trace-driven simulation, the input to the simulator is a trace of instructions or memory references of a given execution of the simulated application. Execution-driven simulation is widely considered to be more accurate than trace-driven simulation because it enables dynamic effects such as the order of synchronization and contention in the simulation to affect the simulated application's execution path.

Execution-driven simulation can further be classified into two categories depending on whether all instructions are interpreted. Simulators in the first category interpret

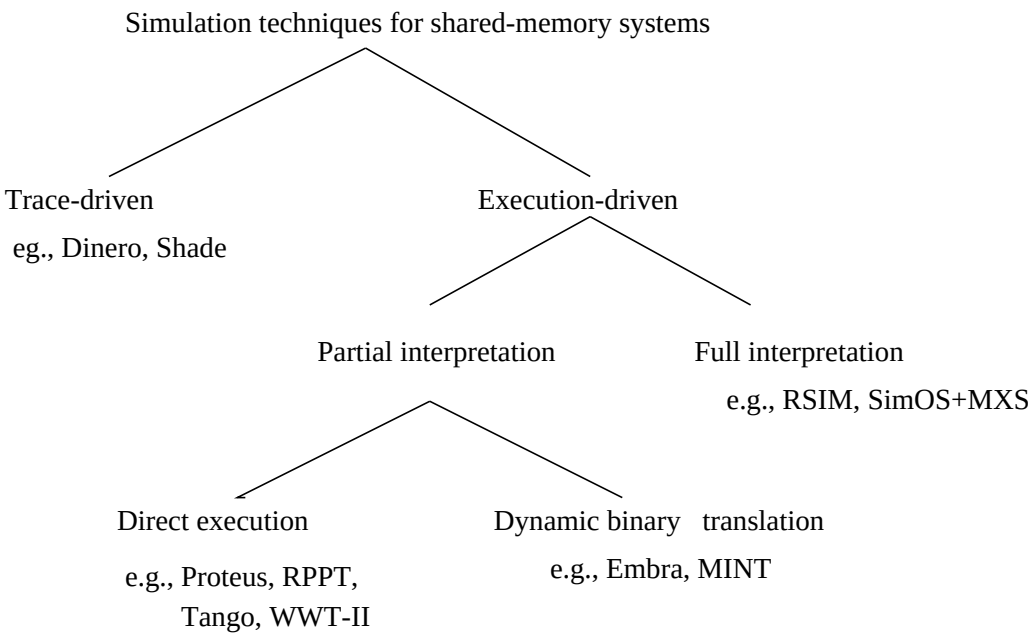


Figure 2.1 Classification of Simulation Techniques

each instruction in software, emulating the effects of the instruction on a simulated machine state and measuring its impact on system timing. RSIM, SimOS+MXS are examples of such simulators [16, 19]. These simulators are significantly slower than trace-driven simulation[2], as they simulate not only *timing* but also the *function* of each instruction.

Decoupling functional and timing simulation can improve the speed of an execution-driven simulator. This concept forms the basis of both direct-execution and dynamic binary translation based simulators. RPPT, Proteus, WWT, Embra are all examples of such simulators [6, 3, 18, 24]. Henceforth , only direct-execution simulators are considered, but most of our discussion applies to simulators based on dynamic binary translation as well. The next section describes direct execution simulators.

[2] We only consider simulation of user-level instructions here

2.2 Direct Execution Simulation for Shared-Memory Architectures with Simple Processors

As discussed in the previous section, in direct execution based simulators functional and timing simulation are decoupled from each other. Functional simulation is done by directly executing the application on the host simulation machine and therefore is very fast. Additionally, timing simulation is sped up by statically assessing the execution time for non-memory instructions and simulating timing in detail only for memory instructions. Many direct execution simulators have been built to simulate shared-memory multiprocessors with simple processors [3, 6, 7, 18].

Components of a direct execution simulator

Common direct execution simulators for shared-memory multiprocessors consist of a mechanism to instrument the application to be simulated, and a timing simulator. The instrumentation mechanism may instrument either assembly code or the executable of the application. It inserts calls to the timing simulator at each memory reference, as these are the only points of interaction among the processors. Some simulators improve performance further by calling the timing simulator only on cache misses [18]. The instrumentation mechanism also statically determines cycle counts for the non-memory portion of each basic block according to the latencies of individual instructions, possibly accounting for pipelining within the basic block. Most current simulators assume single cycle functional units when accounting for this execution time. The Wisconsin Wind Tunnel II simulator assumes a more sophisticated statically scheduled pipeline modeled after the Ross HyperSPARC processor [14]. However, current simulators based on direct execution cannot model the effects of out-of-order issue since they determine the cycle-counts for the non-memory portion statically and they only view one basic block at a time.

The timing simulation in most reported direct execution simulators is based on a discrete event simulation system, and is responsible for the simulation of memory references. When a process invokes the timing simulator on a store, the actions taken depend on whether the simulated target hardware supports blocking or non-blocking stores. If the target machine blocks on a store, then control is not returned to the direct execution of the store's process until the timing simulator completes the simulation of the store. For a non-blocking store, direct execution of the store's process may be resumed as soon as the appropriate events for the store's execution are scheduled by the timing simulator. The resumed process proceeds correctly in spite of the incomplete store because the execution of the rest of the instructions is not dependent on the completion of the store.

The timing simulator treats a load similar to a blocking store. Thus, the direct execution of the load's process is suspended until the simulator completes the simulation of the load and the load returns a value. Current direct execution simulators do not model non-blocking loads.

In this thesis, we use a direct execution simulator, called Simple, as a baseline simple processor based simulator.

2.3 Simulators for Shared-Memory Architectures with ILP Processors

2.3.1 ILP Processors

The base ILP processor considered in this thesis incorporates features such as multiple issue, out-of-order issue, non-blocking loads and stores, register renaming, speculative execution, and aggressive forwarding of values from a store to a later load to the same location, as supported by many current processors (e.g., Digital Alpha 21264, HP-PA8000, Intel Pentium Pro, MIPS R10000). A central data structure in such a processor is a reorder buffer which keeps track of all currently active instructions.

The processor fetches and decodes instructions in program order and inserts them in the reorder buffer. Instruction issue and completion may occur out-of-order. An instruction other than a store is issued to its functional unit as soon as its operands are ready. As in most processors that implement precise exceptions, a store is not issued to the memory system until it reaches the top of the instruction window. Our base processor also implements register renaming and speculative execution as in the MIPS R10000 and Digital Alpha 21264. Specifically, on a branch prediction, it copies the register file mappings into the shadow mappers. On a misprediction, these shadow mappers are used to restore the register mappings in a single cycle. Other important processor structures are the load and store queues, which also serve as the memory disambiguation units in our base processor.

2.3.2 Detailed Simulation of Shared-Memory Architectures with ILP Processors

To the best of our knowledge, only two detailed simulators developed explicitly for shared-memory architectures with state-of-the-art ILP processors have been reported in the literature. These are RSIM [16] and SimOS with the MXS processor simulator[19]. These simulators use straightforward execution-driven simulation, interpreting every instruction and simulating its effects in software. The complete processor pipeline and memory system are faithfully simulated.

2.3.3 Current approximations for Simulating Shared-Memory Architectures with ILP Processors

Researchers have also used simple-processor based simulators to model shared-memory systems with state-of-the-art ILP processors, using certain simple approximations. An implicit approximation made by many studies is that a simulation of a system with a simple processor approximates a system with an ILP processor with the same clock speed. We refer to this approximation as Simple. Other studies have

sped up the clock rate of the simulated processor to model the benefits of ILP [10, 11]. A previous study by Pai et al. showed that the best approximation currently used is to speed up the simple processor's clock cycle and L1 cache access time by a factor equal to the ILP processor's peak instruction issue rate (i) [17]. We refer to this approximation as Simple-ix.

Pai et al. studied the errors produced by the Simple and Simple-ix approximations for 6 applications drawn primarily from the SPLASH and SPLASH-2 suites [17]. They showed that Simple sees errors in execution time ranging from 26% to 192%. Simple-ix is reasonable for some applications, but continues to see large errors in other applications (up to 73%). Their study shows that the key source of inaccuracy is the inability of simple-processor based simulators to model the impact of non-blocking reads. Specifically, the impact of overlapping read misses was found to be contributing to a large portion of inaccuracy. Applications with a significant number of overlapped read misses can experience dramatic reductions in memory stall time at the processor; such overlap is not modeled by current simple-processor based simulators. This work provided the key motivation for this thesis.

Chapter 3

Direct Execution for Simulation of Multiprocessors with ILP Processors

This section investigates the use of direct execution to enhance the speed of simulating shared-memory systems with ILP processors without significant loss of accuracy. There are three key problems with using current direct execution techniques for ILP processor-based shared-memory systems:

- **Handling non-blocking loads**: With non-blocking loads, the timing simulator needs to transfer control back into the direct execution of the load's process before the simulation of the load is complete and before the value that the load will return (in the simulation) is known. This will enable the direct execution process to generate later instructions that the timing simulator needs to execute in parallel with the load. Since at this point, the value to be returned by the simulated load is not known, it is unclear what action to take when the direct execution reaches a later instruction dependent on such a load, and when (and if) to block for this load.

- **Timing simulation of other ILP features**: A simple static analysis is insufficient to model the impact of ILP features such as out-of-order issue and speculative execution on the execution time of CPU instructions and on the time at which a memory instruction can be issued. Therefore, additional support is required for this purpose.

- **Accounting for instructions in mispredicted paths**: Speculative execution in ILP processors implies that occasionally a processor will execute instructions

from an execution path that is later rolled back. Current direct execution techniques do not directly provide a way to account for the impact of such instructions.

Sections 3.1 and 3.2 discuss our solutions to the above problems. Section 3.3 describes the detailed implementation of our approach in DirectRSIM.

3.1 Handling Non-Blocking Loads

We focus on a release consistent shared-memory architecture[3]. We assume that synchronization loads and stores are identified to the simulator.

When a synchronization load invokes the timing simulator, it is treated as a blocking load as in current direct execution simulators. For data loads, the timing simulator attempts to bring the load into the simulated processor in order to send the load to the memory hierarchy. However, the timing simulator does not block the direct execution waiting for the load to complete at (or even issue to) the memory hierarchy. Instead, the timing simulator may return control to direct execution as soon as the load is brought into the simulated processor.

Upon return to direct execution, the simulation executes the load and returns the value currently in the accessed memory location, based on the following insight. If the load does not form a race with a store from another process in the simulated execution, the load and store will be executed in the same order in the direct execution as in the simulated execution. This is because the load and store will be separated by a chain of synchronization releases and acquires; these synchronization operations are executed as in a direct execution simulator and enforce the necessary orderings

[3]Our approach is also applicable to other popular memory consistency models such as sequential consistency and processor consistency. Straightforward implementations of these models use blocking loads, and so direct execution is straightforward to implement for them. Optimized implementations of these models allow non-blocking loads using a speculation mechanism [9]. As will be clear from the rest of the discussion, our approach applies to such implementations as well if these speculations usually succeed; previous work has shown that to be the case [15].

among operations that do not race with each other. Thus, for a load that is not involved in a data race, the value in the memory at the time of the direct execution can be safely returned by the load and used by dependent instructions[4].

On the other hand, a load that is involved in a data race may return a different value in the direct execution than what would be returned in the simulated execution. The value returned would be a legal value given the release consistency model; however, the simulator may not model the timing for this load correctly. Since data races are generally rare in parallel programs, we expect that this will not have a significant impact on accuracy.

3.2 Timing Simulation of other ILP Features and Mispredicted Paths

Like current direct execution simulators, our overall approach also separates the functional and timing simulation, executing the functional part directly on the host machine. Also, like current simulators, our base methodology invokes the timing simulator just before the application executes a memory instruction. However, the application is not instrumented to account for the execution time of any instructions. Instead, the application is instrumented to communicate the path taken by the direct execution since the previous invocation of the timing simulator by the same process. The timing simulator processes this path to perform a timing simulation with the goal of providing the best accuracy and performance possible.

A naive timing simulator would simply replicate the simulation features of RSIM: modeling the register state, instruction effects, and pipeline stages in detail. However, RSIM performs much of its work for the purpose of functional simulation itself. Our timing simulator improves performance relative to RSIM through features based on two insights. First, the functional simulation of an instruction is already performed in

[4]This is always valid in uniprocessor configurations, as there are no data races in such systems.

the direct execution. Second, some parts of the processor simulation can be approximated, as the most important characteristic in determining shared-memory multiprocessor performance is the behavior of the memory system and its interaction with the processor [17]. The following explains how each of the above insights enables performance improvements in our timing simulator.

Our timing simulator can avoid some of the overhead of simulation functions performed by simulators such as RSIM by the following means:

- The simulator avoids the overhead of instruction emulation, or the calculation of the effects of each instruction. The simulator also does not need to maintain the simulated processor state, as this is maintained by the host processor itself.

- The simulator does not perform a detailed cycle-by-cycle simulation of all the stages in the processor pipeline. Instead, on arrival, each instruction gets a timestamp for when it is expected to complete (when known); for a load, we also determine a timestamp for when it can be issued. The details of the implementation of these timestamps are described in Section 3.3.

- Details of register renaming need not be modelled. Register renaming is used to alleviate write after write and write after read hazards. However, the timing simulator does not actually write values into the simulated registers. Thus, the actual process of mapping logical registers to physical registers is not needed, and the simulator can model the benefits of register renaming without paying the overhead of this task.

- For memory instructions, address disambiguation and detection of forwarding (of a value from a previous pending store to a later load to the same location) can be made more efficient, since the load and store addresses are known to the timing simulator as soon as it sees these instructions.

- At branches, the register file mappings need not be stored in shadow mappers since the timing simulator does not need to restore register mappings at mispredictions.

- Other optimizations are artifacts of the above. For example, the amount of state that must tracked along with an instruction is reduced (e.g., renamed register names). This results in less overhead when initializing the data structures for each dynamic instruction instance.

None of the above optimizations sacrifices simulation accuracy. However, we do include some approximations in the timing simulator based on previous results. We observe that shared-memory architecture studies are often most concerned with memory system performance; therefore, we need to be concerned about the accurate modeling of memory instructions and their interaction with the processor [17]. We may be able to approximate processor features that do not impact memory instructions.

Based on the above observation, the most potentially profitable approximation we make is that we do not simulate speculated execution paths that were mispredicted. This assumption greatly reduces the complexity of modeling an ILP processor with direct execution. Note that this assumption does not preclude modeling other effects of speculation. Thus, on branches, we can still keep track of prediction tables and determine whether the branch would be mispredicted in the simulated execution. Since the correct direction for the branch is provided by the direct execution, the timing simulator can determine if the branch is predicted correctly as soon as a value is returned from the branch predictor. On a misprediction, we can simulate the actual delay in instruction fetch from the correct path as the processor waits for the branch to be resolved. The only approximation made is that we do not actually simulate the instructions from the mispredicted path. Non-memory instructions from the mispredicted path should have little impact on system performance. Memory instructions allowed to issue from the mispredicted path can help or hurt overall

system performance by increasing contention in the memory hierarchy, by interfering with the cached data of other processors, or by bringing useful data into the cache early. However, the approximation of not simulating mispredicted paths is not likely to be a significant source of error as long as speculation usually succeeds. This solution also addresses the third problem raised at the beginning of this section. Section 7.2 discusses the possibility of including mispredicted-path simulation in DirectRSIM. Note that the optimization of not simulating the mispredicted path is difficult (if not impossible) to incorporate within simulators such as RSIM since RSIM does not know whether the branch is mispredicted or not until the branch is actually resolved.

Contemporary processors also perform speculative load execution, which is analogous in many ways to branch speculation. With this technique, hardware issues a load while the address of a previous store is not yet disambiguated. If the store later turns out to be to the same address as the load, then the load and the execution following it needs to be rolled back. We assume perfect memory disambiguation in our timing simulator; i.e., neither an incorrectly speculated load nor its following execution are simulated. Again, this is not likely to be a major source of error if speculation usually succeeds. As with branch prediction, this approximation is difficult to include in simulators such as RSIM.

3.3 Implementation of the DirectRSIM Simulator

DirectRSIM implements the direct execution methodology described in this section. It consists of an application *instrumentation mechanism* (Section 3.3.1), and a *timing simulator* (Section 3.3.2).

3.3.1 Application Code Instrumentation

Like current direct execution simulators, the instrumentation mechanism for DirectRSIM inserts calls to the timing simulator before each memory instruction in the application. However, the instrumentation code does not attempt to stati-

cally determine cycle counts for the ILP processor. Instead, it records the ranges of contiguous program-counter values traversed by the direct execution since the last memory instruction was passed to the timing simulator. For this purpose, the code instrumentation marks all unconditional branches and the taken paths of all conditional branches as ending a program-counter range and starting a new range. Each contiguous program-counter range can span multiple basic blocks, and multiple ranges can be passed to the timing simulator on each call.

Our current implementation of the instrumentation system works on the assembly code of a program. However, the techniques used here can be extended to the more general methods of executable-editing or dynamic compilation.

3.3.2 Architectural Timing Simulator Implementation

The timing simulator consists of three main parts: the event-driven simulation engine, the multiprocessor memory system simulator, and the processor simulator. The event-driven simulation engine and multiprocessor memory system simulation are common to all our simulators and are described in more detail in Chapter 4. The processor simulator is the key feature that sets DirectRSIM apart.

Processor simulator

As discussed in Section 3.3.1, the instrumented application code calls the processor simulator at each memory reference. The code maintains a list of program counter ranges traversed since the last simulator call. Upon entry, the processor simulator processes this list and attempts to bring each instruction from this list into the instruction window.

In processing these instructions, the key work that needs to be done by the processor simulator is: (1) keeping track of true dependences and structural hazards, and determining when instructions complete or when loads and stores can be issued based on these dependences, (2) retiring instructions from the instruction window at appropriate times based on the above determination of completion times, (3) maintaining

branch prediction tables, and (4) implementation of forwarding (i.e., if a load is ready to issue while a previous store to the same location is pending, then the store's value is forwarded to the load without issuing the load).

The key data structures in the processor simulator are (1) a structure analogous to the reorder buffer or instruction window of an ILP processor, (2) a load queue and a store queue to keep track of memory operations that need to be issued, (3) a data structure that keeps track of outstanding stores, hashed on their addresses, to enable easy store value forwarding to later loads to the same location, (4) the branch prediction table, and (5) a structure for simulating structural hazards for functional units.

The memory system and simulation engine of DirectRSIM provide a global view of time shared by all the nodes in the system. However, unlike RSIM, the processors are not required to act in lockstep with the global clock when handling internal processor actions. The processor is allowed to maintain local views of the clock that run ahead of the global clock, as long as it synchronizes with the global clock before issuing any instruction to the memory system. The completion timestamps of individual instructions are one type of localized clock. Additionally, each processor simulator has two other views of time: a fetch time and a retire time. Instructions are marked with the value of the fetch time when they are fetched into the window, and the processor retires instructions from the head of its instruction window according to the value of the retire time. Both of these clocks can run ahead of the global simulation clock.

As the processor simulator brings instructions in, it tags non-memory instructions with their completion times, if known. The completion time for an instruction is known as long as it is not data dependent on any incomplete loads (either directly or through other instructions dependent on the load). The completion timestamp also depends on the latency of the instruction and the availability of a functional unit. If the completion time is not immediately known, the instruction is attached to

the instructions on which it is dependent; its completion timestamp will be set upon completion of these instructions.

For a load instruction, the processor simulator calculates a timestamp for the time when the load is ready to issue (if known), and inserts it in the load queue in issue time order. When the global simulation time catches up with the issue time of a load, the processor simulator checks to see if the load can be forwarded from a previous store. DirectRSIM uses the addresses passed in for memory instructions from the direct-execution to efficiently check for any instances of a load matching the address of a previous store in the instruction window. If the load does match an earlier store, the processor simulator considers the load forwarded and marks a completion time for the access. Otherwise, the processor simulator schedules an event for issuing the load to the memory system.

As with most current processor simulators, to insure precise interrupts, a store instruction is marked ready for issue only when it reaches the top of the instruction window. At this time, the store will be inserted in the store queue with an issue timestamp equal to the current retire time. When the global time catches up with the issue time, the event-driven simulator schedules an event for the issue of the store.

Instruction fetching continues until either the instruction window fills up, or all instructions that were executed in the direct execution are exhausted, or a branch is mispredicted. In the latter case, the fetch continues once the branch miss penalty is determined. If the instruction window is full, the processor simulator tries to retire the first set of instructions. This is an entirely local action, and takes place according to the retire time, possibly setting a new value for this clock.

The processor simulation (and the corresponding direct execution) is suspended when the instruction window of the processor is full, it cannot retire any further instructions (this can only happen if there is an incomplete load at the head of the window), and no other loads or stores can be issued (either because they are dependent on other loads, or because the cache ports are full, or because the global

time has not caught up with their issue time yet). At this point, the processor stalls in a state waiting for an action that will allow progress on any of the above situations. A processor's direct execution may be resumed once all of its directly executed instructions have been entered in its instruction window.

Effectively, the timing simulator acts like a trace-driven simulator, operating on the trace of instructions executed since its last invocation by the same process. The methodology, however, is still execution-driven and differs from conventional trace-driven simulation because the execution path of the simulated application can still be affected by the dynamic ordering of synchronization accesses and contention in the system. In the uniprocessor case, however, DirectRSIM effectively becomes a trace-driven simulator for ILP processors.

Other differences between DirectRSIM and RSIM

In addition to the differences described above, DirectRSIM also uses user-level lightweight processes to provide the register and stack state needed by each simulated processor for direct execution. Each activation or deactivation of a process requires the overhead of a lightweight context-switch. Simple and Simple-*ix* also use the same lightweight process structure for direct execution. RSIM does not use lightweight processes, as all register and stack state for the RSIM CPUs is simulated in software.

DirectRSIM also avoids simulating L1 cache hits whenever possible (whenever the cache is guaranteed to have ports available and not be stalled for resources such as MSHRs). The processor simulator may still have to stall to allow the global simulation clock to catch up with the issue time of an access, but if it determines (when the global clock has caught up) that the access is a hit and that the cache is available, the processor does not send the access to the caches to be simulated. Instead, the processor lets a store that hits in the L1 cache leave the system and assigns a load that hits in the L1 cache a completion time stamp based on the issue time and the L1 cache access time. This can save overhead in invocations of the L1

cache simulation module, as well as the memory allocation and deallocation of the data structures used to track accesses in the memory system. Simple and Simple-*ix* also use this optimization; RSIM actually issues all hits to the caches, consistent with its cycle-by-cycle detailed simulation philosophy.

Chapter 4

Evaluation Methodology

This study compares the accuracy and performance of four shared-memory multiprocessor simulators. The following sections describe the architectural parameters of the systems modeled, the simulators used in this study and their differences, the metrics used to evaluate the simulators, and the applications studied.

4.1 Simulated Architectures

We model the performance of cache-coherent, non-uniform memory access (CC-NUMA) shared memory multiprocessor systems in this study. Cache coherence is maintained through an invalidation-based MESI directory coherence protocol. Each system node includes one processor, a two-level write-back cache hierarchy, part of the system's distributed physical memory and directory structure, a network interface, and a split-transaction bus connecting the different components of the node. All nodes are connected by a two-dimensional mesh network. Contention is modeled at all resources in the processor, memory hierarchy, bus, and network.

The base processor incorporates aggressive features such as out-of-order issue, multiple instruction issue, non-blocking loads and stores, register renaming, and speculative execution, as described in Section 2.3.1. Both caches are non-blocking and use a write-allocate write-back policy. Each cache supports 8 miss status holding registers (MSHRs) [12] to hold state related to outstanding cache misses and to merge misses to a single cache line into a single external miss. The L1 cache has a size of 16 KB and the L2 cache has a size of 64 KB. These sizes reflect our application input sizes (described in Section 4.4), following the methodology of Woo et al. [25]. All

primary working sets in these applications fit in the L1 cache, while the secondary working sets do not fit in the L2 cache. Currently, a perfect TLB and instruction cache are modelled since the application suite is known to have a small instruction cache and TLB miss ratio. More realistic models of the instruction cache and TLB can be implemented in all the simulators in a straightforward manner.

Figure 4.1 summarizes the key system parameters of interest for our base system. Results from five other variations of the base system parameters are also reported, as described in Section 5.

4.2 Simulators Used in the Study

We compare the performance of four different simulators in this study. The differences among all four simulators are limited to the processor model and its interaction with the cache hierarchy. RSIM and DirectRSIM model a processor with out-of-order issue, multiple issue, and nonblocking loads. RSIM uses detailed simulation (Section 2.3), and DirectRSIM uses our new technique (Section 3.3). Simple and Simple-*ix* model a single-issue, in-order issue processor with blocking loads and no speculation, using current direct execution methodology. The memory system simulation in all these simulators uses nearly identical code and is based on an event-driven simulation engine. Most of the system is divided into different modules that generate events in an obvious manner (example modules are L1 cache, L2 cache, bus, memory modules, network interfaces, etc.).

Events for the simulated modules (including lightweight process activations) are scheduled by inserting them on a central event queue. The scheduled activities are triggered at an appropriate time by a centralized driver routine. One difference in the RSIM's event-driven simulation is a special event called RSIM_EVENT. This event occurs every cycle and examines the state of each processor, L1 cache, and L2 cache, scheduling any external events triggered by these parts of the system in the event queue. In cycles in which no other event is scheduled, RSIM_EVENT repeats

ILP Processor	
Processor speed	500MHz
Maximum fetch/retire rate (instructions per cycle)	4
Instruction issue window	64 entries
Functional units	4 integer arithmetic 4 floating point 4 address generation
Branch speculation depth	8
Memory unit size	32 entries
Cache parameters	
Cache line size	64 bytes
L1 cache (on-chip)	Direct mapped, 16 K
L1 request ports	2
L1 hit time	1 cycle
Number of L1 MSHRs	8
L2 cache (off-chip)	4-way associative, 64 K
L2 request ports	1
L2 hit time	8 cycles, pipelined
Number of L2 MSHRs	8
Memory and bus parameters	
Memory access time	60 cycles
Memory interleaving	4-way
Bus width	32 bytes
Bus frequency	167 MHz
Network parameters	
Network speed	125 MHz
Network width	64 bits
Flit delay (per hop)	1 network cycle
Resulting contentionless memory latencies	
Local memory	85 cycles
Nearest remote memory	182 cycles
Farthest remote memory	262 cycles
Latency for nearest cache to cache transfer	210 cycles
Latency for farthest cache to cache transfer	309 cycles

Figure 4.1 Base system parameters

automatically without the overhead of enqueueing and dequeueing the event. Thus, the processors and caches are simulated on a cycle by cycle basis in RSIM. The motivation for this is that in a detailed ILP processor simulation, it can be expected that some processor or cache will have some event scheduled every cycle. In contrast, the other simulators do no have any events that are always scheduled for every cycle.

4.3 Metrics

This study evaluates the simulators based on their accuracy and performance.

Simulator accuracy

The primary metric used to evaluate the accuracy of a simulator is the execution time it reports for the simulated application. To gain further insight, we also study three components of the execution time reported by the simulator – CPU time, memory stall time, and synchronization stall time. While it is straightforward to determine these components for a simple-processor based system, it is more difficult for a system that models state-of-the-art ILP processors where instructions can be overlapped and reordered. RSIM and DirectRSIM calculate various stall components according to a heuristic used in previous work [17, 19]: any cycle in which the processor can retire instructions at its peak rate is considered a *busy* cycle. All other cycles are charged to the first instruction that could not retire in that cycle, as that instruction is the one that is preventing the processor from retiring at peak rate.

The execution times and components for the various simulators are reported not to suggest any architectural tradeoffs or advantages, but rather to show the ability of each type of simulator to capture the behavior of a shared-memory multiprocessor with state-of-the-art processors. In particular, this study measures the error of each of the direct-execution simulators (Simple, Simple-*i*x, and DirectRSIM) relative to the times reported by the detailed execution-driven simulator (RSIM). RSIM has not been validated against a real machine, as we do not have access to a machine with the architecture modeled by RSIM. However, any errors seen by RSIM outside the

processor would also be seen in the other simulators, as these simulators use nearly identical code in all other modules. Consequently, this error metric represents the additional discrepancy seen by a simulator with an approximate processor model.

Simulator performance

To measure simulator performance, the elapsed (wall-clock) time is used for each simulation when run on a single 250MHz UltraSPARC-II processor of a Sun Ultra Enterprise 4000 server with 1GB memory. The simulators were all compiled using the Sun C 4.0 compiler with the highest practical level of optimization – for most of the source files, this is `-xO4`; however, some of the context-switching libraries in Simple, Simple-*ix*, and DirectRSIM must be compiled with `-xO3`.

Although each of the simulators supports detailed statistics related to all of its main modules, the runs for the simulator performance metrics are measured *without* collecting or reporting any of the most detailed statistics; only the total execution time, the percentage of execution time spent in each component of time, the cache miss behavior, and the cache MSHR occupancy are collected. Thus, these numbers represent the performance of the actual simulation core, without the need for potentially extraneous statistics.

4.4 Applications

This study uses 5 different applications – FFT and Radix from the SPLASH-2 suite [25], LUopt an optimized version of LU from the SPLASH-2 suite, MP3D from the SPLASH suite [21], and Erlebacher from the Rice parallel compiler group [1]. In LUopt when compared to LU one important loop nest has been interchanged to increase the overlap of read misses and thereby improve performance in a system with ILP processors. For better load balance, LUopt has been modified slightly to use flags for synchronization rather than barriers. The number of processors in the simulated configuration for each application is chosen according to the scalability of the application. Using systems with more processors would require an increase in the

input size for reasonable scalability; this would result in a very long running time for RSIM.

Figure 4.2 lists the input sizes for the applications and the number of processors in each multiprocessor configuration. The input sizes are at least as large as the default size specified in the distribution for each application, with the exception of LUopt. In LUopt, a matrix one size smaller than the default has been used because of the large simulation time required with RSIM; however, the number of processors has also been scaled down correspondingly from the maximum recommended configuration.

Application	Input Size	Processors
FFT	65536 points	16
LUopt	256 by 256 matrix, block 8	8
Radix	512K keys, max: 512K, 1024	8
Mp3d	50000 particles	8
Erlebacher	64 by 64 by 64 cube, block 8	16

Figure 4.2 Application input sizes and configurations

Chapter 5

Results

This chapter evaluates the accuracy and performance of the simulator models under investigation. Previous work has explored the differences in accuracy between RSIM and models such as Simple and Simple-*ix*[5] [17]. Thus, this chapter particularly focuses on DirectRSIM.

Section 5.1 examines the accuracy of DirectRSIM, Simple-*ix*, and Simple by comparing the values these simulators report for total execution time and its components with those reported by RSIM. The execution time is divided into CPU, memory, and synchronization components according to the methodology in Section 4.3. Section 5.2 discusses the performance of the four simulators. Section 5.3 provides a detailed analysis of the sources of DirectRSIM's performance benefits relative to RSIM and the sources of any slowdown seen by DirectRSIM relative to Simple.

A discussion of the performance of Simple, relative to the recently released Wisconsin Wind Tunnel-II, is provided in Appendix A. The results in the appendix show that Simple performs comparably to the Wisconsin Wind Tunnel-II and can thus be considered representative of the state-of-the-art in direct-execution simulators for shared-memory multiprocessors with simple processor models.

[5] The numbers reported here differ from those in previous work [17] because this study uses a more aggressive compiler (Sun C compiler) and uses a slightly different memory hierarchy (write-back L1 cache, MESI coherence protocol, slightly longer memory latencies).

5.1 Simulator Accuracy

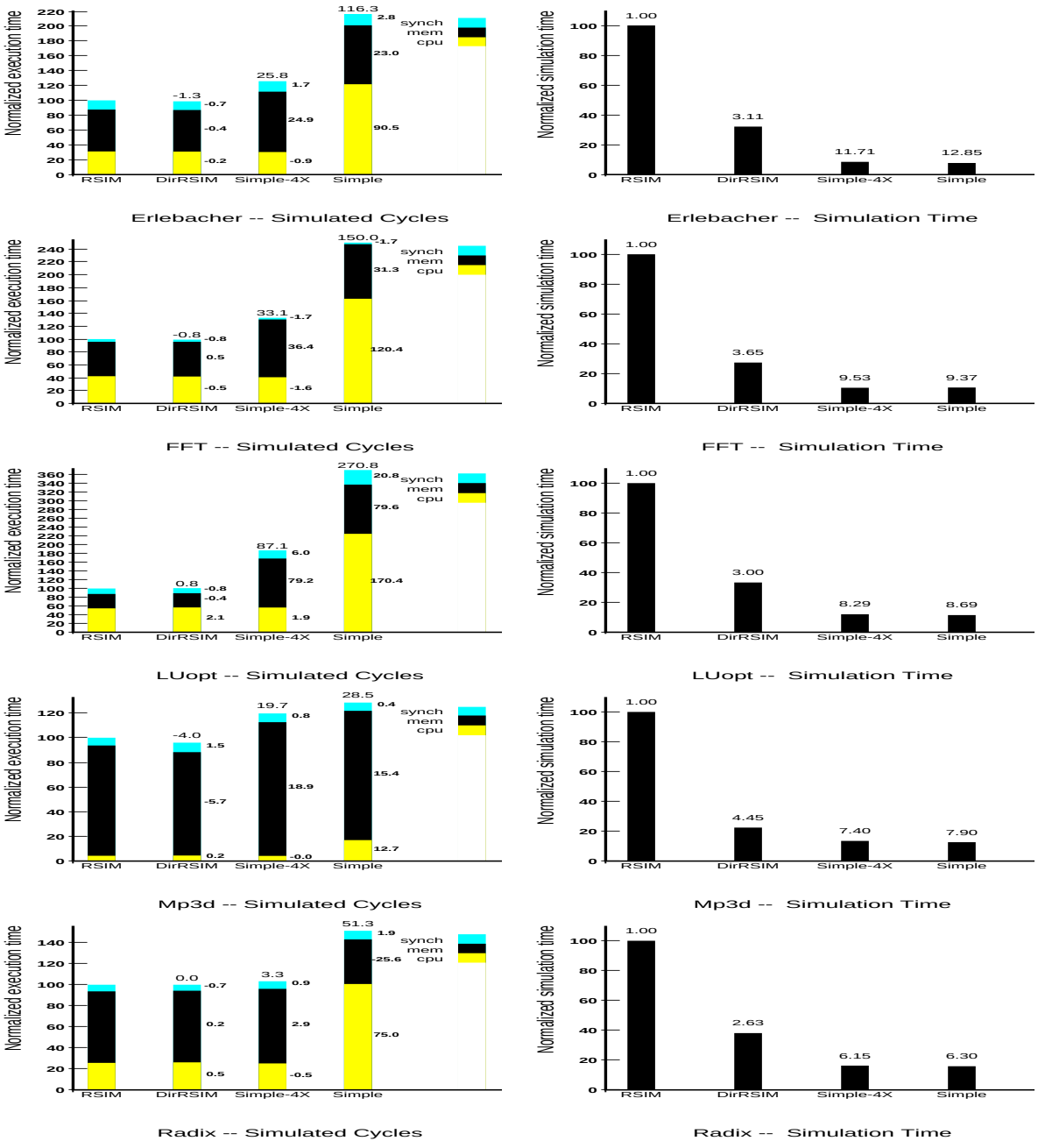
5.1.1 Base Configuration

Figure 5.1(a) shows the execution time reported by each of the simulation models for each of the application programs using the base system configuration described in Section 4.1. Each one of these graphs shows the total execution time reported by RSIM, DirectRSIM, Simple-4x, and Simple. These graphs are split up into three components of execution time (CPU, memory stall, and synchronization stall), and the lengths of the bars are normalized to the length of the execution time reported by RSIM. The number above each bar gives the percentage error in total execution time for the corresponding simulator relative to RSIM. Numbers shown to the side of a bar represent the breakup of the total error among the three components of execution time.

Figure 5.1(a) shows that DirectRSIM reports overall simulated execution time within 5% of RSIM on all of our applications. This is a striking improvement from the best previous approximate simulation model using Simple-4x, which sees an execution time error of 87% for LUopt and 20%-40% on 4 other applications studied. The straightforward version of Simple sees errors ranging from 29% to 271%.

Figure 5.1(a) also shows that most of the errors seen in Simple-4x are in the memory component of execution time. Proper prediction of the fraction of time spent in the memory component is important for a large class of architectural studies that target this execution time component. Figure 5.1(a) shows that Simple-4x often significantly overestimates the importance of memory time, while Simple often greatly underestimates this component. In contrast, DirectRSIM provides a close estimate in all cases.

The differences between these four simulation models arise from their ability to capture the benefits that ILP provides to the various components of execution time. For example, ILP can reduce the CPU component of execution time by issuing multi-



(a) Reported simulated time (b) Elapsed time of simulation

Figure 5.1 Comparison of simulator accuracy and performance for the base system configuration

ple instructions at a time, or by issuing instructions out of order. ILP can reduce the memory component by overlapping multiple long latency memory operations with each other, or by overlapping memory latency with CPU instructions. ILP can also increase a component of execution time by increasing contention for resources or by changing an access pattern.

As reported by Pai et al. [17], the Simple model cannot capture the effects of ILP on either the CPU component of execution time or the memory component. Simple-4x models much of the benefits of multiple instruction issue for the CPU component (because its clock speed is increased by a factor equal to the issue width of the processor). However, it does not allow multiple read misses to be overlapped with each other. As a result, this method is not able to properly capture ILP-specific improvements in the memory component of execution time, which is dominated by read miss penalties.

In contrast, DirectRSIM models the impact of ILP in both CPU and memory components of execution time. Thus, DirectRSIM provides a closer and more consistent approximation to the functionality of detailed execution-driven simulators than either Simple or Simple-ix.

5.1.2 Other system configurations

Figure 5.2 shows the errors relative to RSIM seen by DirectRSIM, Simple-ix, and Simple in a variety of multiprocessor configurations. It shows the error in the total execution time, as well as the component of the error in the memory time. The first row in these tables repeats the data of the base configuration shown in Figure 5.1(a). The second row represents a system with rougly twice the local and remote memory latency of the base configuration. The third row shows a system with three times the local memory latency of the base configuration and a minimum contentionless remote-to-local latency ratio of 3:1; this is chosen to represent potential future configurations with faster processors and relatively higher remote memory latencies that are much

harder to overlap. The fourth row shows a system with the base memory and network parameters, but with a processor that is twice as aggressive as the base. Specifically, we double the instruction issue width, instruction window size, processor memory unit size, functional units, branch-prediction hardware, cache ports, and MSHRs. The fifth row similar to the previous row, except that the instruction window size is four times that of the base[6]. The sixth row shows a system with the base processor, cache, and memory parameters, but with a constant-latency 50-cycle network rather than a 2-D mesh configuration. These configurations are chosen to represent expected future trends toward higher processor clock speeds, aggressive processor microarchitectures, and aggressive network configurations.

As Figure 5.2 shows, DirectRSIM continues to see very low errors in execution time relative to RSIM – an average of 1.6% and a maximum of 6.3%.

In contrast, the errors with Simple-*i*x remain high for most of the applications, and continue to vary widely, with errors in total execution time ranging from 2% to 128%, averaging nearly 40%. The errors seen with Simple range from 9% to 438%, averaging around 130%. As expected, the errors are greatest in the applications with the most read miss overlap. This application characteristic becomes even more important in determining the performance of systems with future aggressive processors (e.g., ILP+ and ILP++ configurations).

In conclusion, DirectRSIM achieves significantly greater and more reliable accuracy than Simple-*i*x or Simple in a variety of current-generation and next-generation multiprocessor configurations.

5.2 Simulator Performance

Figure 5.1(b) graphically depicts the relative simulation times taken by each of the four simulators in the base configuration. The bars in this graph are normalized to the

[6]In the fourth and fifth rows, Simple-8x is used rather than Simple-4x.

	Erle.		FFT		LUopt		Mp3d		Radix		Avg.	
	Tot	Mem	Tot	Mem	Tot	Mem	Tot	Mem	Tot	Mem	Tot	Mem
Base	-1.3	-0.4	-0.8	0.5	0.8	-0.4	-4.0	-5.7	0.0	0.2	1.4	1.4
Lat. x2	-1.2	-1.0	-2.0	0.7	1.5	1.1	-3.8	-5.2	0.0	0.5	1.7	1.7
Lat. x3	-0.7	-1.3	0.5	0.2	0.7	-0.0	-2.3	-4.2	0.0	0.4	0.8	1.2
ILP+	-2.3	-0.6	-0.4	0.2	0.3	2.4	-1.9	-4.6	-0.8	-0.0	1.1	1.6
ILP++	-6.3	-2.6	-2.1	-1.2	1.2	4.2	-0.3	-5.8	-1.4	-0.2	2.3	2.8
C. net	-0.4	-0.4	-0.8	0.3	0.8	-0.2	-3.4	-4.1	-0.2	0.2	1.1	1.0
Avg.	2.0	1.1	1.1	0.5	0.9	1.4	2.6	4.9	0.4	0.2	1.4	1.6

(a) Error of DirectRSIM relative to RSIM

	Erle.		FFT		LUopt		Mp3d		Radix		Avg.	
	Tot	Mem	Tot	Mem	Tot	Mem	Tot	Mem	Tot	Mem	Tot	Mem
Base	25.8	24.9	33.1	36.4	87.1	79.2	19.7	18.9	3.3	2.9	33.8	32.5
Lat. x2	27.6	26.3	35.6	37.0	109.0	98.9	18.0	17.1	3.6	2.2	38.8	36.3
Lat. x3	27.6	23.7	37.2	39.9	90.5	83.0	9.9	10.3	2.0	1.2	33.4	31.6
ILP+	31.4	32.5	50.0	56.0	121.8	115.0	32.7	32.7	4.0	3.4	48.0	47.9
ILP++	69.8	72.0	60.0	66.1	127.8	121.5	47.5	51.7	9.4	15.6	62.9	65.4
C. net	23.1	23.7	29.7	33.7	84.7	77.6	17.0	17.0	3.6	3.1	31.6	31.0
Avg.	34.2	33.9	40.9	44.9	103.5	95.9	24.1	24.6	4.3	4.7	41.4	40.8

(b) Error of Simple-*ix* relative to RSIM

	Erle.		FFT		LUopt		Mp3d		Radix		Avg.	
	Tot	Mem	Tot	Mem	Tot	Mem	Tot	Mem	Tot	Mem	Tot	Mem
Base	116.3	23.0	150.0	31.3	270.8	79.6	28.5	15.4	51.3	-25.6	123.4	35.0
Lat. x2	77.9	24.2	100.5	31.1	232.4	100.3	22.2	14.9	23.6	-18.7	91.3	37.8
Lat. x3	54.5	23.0	72.0	35.6	147.6	81.7	10.6	9.1	8.8	-7.2	58.7	31.3
ILP+	156.3	29.2	227.8	47.1	423.5	115.3	43.2	27.4	68.0	-29.1	183.8	49.6
ILP++	231.2	67.7	249.5	56.6	437.8	121.8	59.2	45.7	76.8	-18.6	210.9	62.1
C. net	110.6	21.4	145.8	27.9	264.9	78.4	21.2	10.6	55.9	-24.1	119.7	32.5
Avg.	124.5	31.4	157.6	38.3	296.2	96.2	30.8	20.5	47.4	20.5	131.3	41.4

(c) Error of Simple relative to RSIM

Figure 5.2 Execution time errors of simulation models.
(Averages are over the absolute values of the errors.)

elapsed time taken by RSIM. The numbers above each bar show the speedup achieved by each simulator relative to RSIM . Figure 5.3 extends the data of Figure 5.1(b) for all the applications and configurations reported in Section 5.1. It gives the speedup of DirectRSIM over RSIM, Simple-*i*x over DirectRSIM, and Simple-*i*x over RSIM. We do not report the speedups for Simple as they are similar to those for Simple-*i*x, and Simple gives much greater errors.

The Simple-*i*x simulator gives the best simulation time in each of the cases, as it avoids the overhead of processor simulation. Consequently, this simulator sees an average speedup of 10 over RSIM. DirectRSIM does have some runtime overheads from processor simulation. Nevertheless, DirectRSIM sees an average speedup of 4 over RSIM. Even more interestingly, the performance advantage of Simple-*i*x is reduced to an average of 2.7 compared to DirectRSIM. The competitive performance of DirectRSIM indicates that the performance benefits of simple-processor based simulator models may no longer be enough to justify their inaccuracies in modeling current and future multiprocessor systems. Further, the relative performance stability of DirectRSIM shows that it is a cost-effective way to accurately simulate many useful configurations.

For reference, Appendix B provides data for absolute performance of all the simulators in simulated instructions per second.

5.3 Detailed analysis of DirectRSIM's performance

5.3.1 Comparing DirectRSIM with RSIM

As described in Chapters 3 and 4, DirectRSIM sees performance benefits by eliminating some of the details of pipeline management, speculative execution, memory disambiguation, and instruction emulation, as well as by replacing the cycle-driven processor and cache hierarchy of RSIM with a purely event-driven system. Figure 5.4 summarizes the average results of `gprof` profiles of RSIM executions for the appli-

	Erle.	FFT	LUopt	Mp3d	Radix	Avg.
Base	3.1	3.6	3.0	4.5	2.6	3.4
Lat. x2	3.5	5.0	3.4	6.1	3.1	4.2
Lat. x3	4.0	4.5	3.8	6.7	4.8	4.7
ILP+	3.6	4.1	2.9	3.4	3.1	3.4
ILP++	3.9	5.0	3.5	7.0	3.9	4.7
C. net	3.1	3.7	3.0	5.3	2.8	3.6
Avg.	3.5	4.3	3.3	5.5	3.4	4.0

(a) Speedup of DirectRSIM over RSIM

	Erle.	FFT	LUopt	Mp3d	Radix	Avg.
Base	3.8	2.6	2.8	1.7	2.3	2.6
Lat. x2	3.7	2.5	2.6	1.8	2.4	2.6
Lat. x3	3.4	3.4	2.6	2.9	2.6	3.0
ILP+	3.3	2.4	2.5	2.1	2.1	2.5
ILP++	3.3	2.4	2.6	1.9	2.6	2.6
C. net	3.6	2.8	3.0	2.0	2.3	2.8
Avg.	3.5	2.7	2.7	2.1	2.4	2.7

(b) Speedup of Simple-*ix* over DirectRSIM

	Erle.	FFT	LUopt	Mp3d	Radix	Avg.
Base	11.7	9.5	8.3	7.4	6.2	8.6
Lat. x2	13.1	12.5	8.6	11.1	7.6	10.6
Lat. x3	13.7	14.9	9.9	19.4	12.4	14.1
ILP+	11.9	9.9	7.4	7.0	6.6	8.6
ILP++	12.8	12.1	9.1	13.2	10.3	11.5
C. net	11.2	10.5	8.9	10.6	6.6	9.6
Avg.	12.4	11.6	8.7	11.4	8.3	10.5

(c) Speedup of Simple-*ix* over RSIM

Figure 5.3 Performance analysis of the simulators.

cations studied on the base configuration, with the function calls of RSIM divided according to the logical tasks they perform. This figure also shows the impact of DirectRSIM on each component of RSIM's execution time as determined by `gprof` profiles of DirectRSIM.

Overhead eliminated by DirectRSIM

As discussed in Section 2, RSIM requires the processors and cache hierarchies of the system to proceed in lockstep with a global clock, effectively using a cycle-driven simulation controller for that part of the system. As described in Section 3, DirectRSIM does not tie internal processor actions (such as instruction fetch and retire) to the global clock; processors must synchronize with the global clock only to issue memory instructions. Thus, DirectRSIM eliminates the cycle-driven simulation controller altogether, using event-driven simulation for the entire system.

DirectRSIM also eliminates the overhead of instruction emulation, address generation, and system call emulation, as these tasks are handled in the course of direct execution.

Overhead reduced by DirectRSIM

We find that DirectRSIM significantly reduces the top five overhead components experienced by RSIM, which collectively account for over 80% of RSIM's execution time.

DirectRSIM's use of timestamps and the provision to allow internal processor actions to proceed ahead of the global clock, as described in Section 3 enable more efficient management of the instruction window, register renaming, data dependences, and functional units. For example, the use of timestamps allows DirectRSIM to eliminate some of the queuing for registers and functional units seen in RSIM; DirectRSIM requires queuing only for instructions data dependent on outstanding loads, and scans the structural hazard list to determine functional unit availability. Additionally, DirectRSIM significantly reduces the cache overhead by avoiding cache invocations on L1 cache hits, as described in Section 3.3.2.

Task	RSIM Overhead	Impact of DirectRSIM
Instruction fetch and decode	21.58%	reduce
Functional unit management	20.35%	reduce
Processor memory unit	17.40%	reduce
Instruction retirement	14.38%	reduce
Cache simulation	7.07%	reduce
Cycle-driven simulation controller	5.17%	eliminate
Branch prediction and speculation	4.53%	reduce
Instruction emulation	3.06%	eliminate
Memory, bus, and network simulation	2.46%	no change
Event-driven simulation controller	1.97%	increase
Statistics	0.92%	no change
Address generation	0.83%	eliminate
System call emulation	0.28%	eliminate

Figure 5.4 Components of RSIM's simulation time and impact of DirectRSIM

DirectRSIM reduces the cost of simulating the processor memory unit by taking advantage of address information provided in direct execution mode. Using this information, DirectRSIM can efficiently check for instances of load addresses matching those of previous stores and can issue a load without any constraints if there is no match; in contrast, RSIM must first check a load against all previous stores in the instruction window before attempting to issue it. At the completion of the load (and possibly even after), RSIM may need to check the load address again to determine if the address of a previous store has been discovered to conflict with the load.

Additionally, DirectRSIM avoids a small percentage of execution time by using the results of the direct-execution to determine if a branch to be simulated is taken or not; if DirectRSIM's prediction does not match the actual direction of the branch, no instructions on the mispredicted path are simulated. Consequently, DirectRSIM avoids both the cost of restoring the register renaming table on a recovery and the cost of flushing instructions from the instruction window and memory unit when a branch is mispredicted. Among the various techniques described in this section to reduce or eliminate simulation overhead, this is the only one that involves a relatively significant approximation.

Overhead increased by DirectRSIM

DirectRSIM can increase overhead related to the simulation engine. First, DirectRSIM needs to enqueue and dequeue more events on the global event list than RSIM does, as all the modules of DirectRSIM are event-driven. Additionally, DirectRSIM introduces context-switch overhead that is not present in RSIM, as the simulated processors in DirectRSIM are implemented as light-weight processes. DirectRSIM aims to minimize the number of context switches by delaying and rescheduling a light-weight process only when the processor must interact with the cache hierarchy (to issue or wait for an instruction), rather than for any internal processor actions.

Overall impact

The net impact of these changes in overhead is that DirectRSIM sees an average speedup of 4 over RSIM for the reported configurations and applications. Of particular interest are the increases in DirectRSIM speedup for the longer-latency configurations, which represent future configurations with faster processor speeds. DirectRSIM profits by switching from a cycle-driven simulator to a purely event-driven simulator. As a result, the simulation time taken by DirectRSIM is less sensitive to future increases in system latencies than the simulation time needed for RSIM.

5.3.2 Comparing DirectRSIM with Simple

As discussed in Section 5.2, DirectRSIM is on average 2.7x slower than Simple (or Simple-*ix*) over all the configurations and applications studied. Hence, the overall overhead seen by DirectRSIM relative to Simple amounts to 63% of its execution time. Figure 5.5 summarizes the average results of `gprof` profiles of DirectRSIM executions for the applications studied with the base configuration.

The function calls of DirectRSIM are split according to logical task. This figure shows the key tasks that contribute directly to the slowdown of DirectRSIM relative to Simple. These tasks are the processor simulation tasks. The processor simulator in

Task	DirectRSIM Overhead	Source of slowdown?
Instruction fetch and decode	29.70%	√
Cache simulation	11.75%	
Lightweight processes	11.12%	
Event-driven simulation controller	10.34%	
Processor memory unit	9.78%	√
Memory, bus, and network simulation	8.86%	
Instruction retirement	6.30%	√
Functional unit management	6.11%	√
Direct execution	4.35%	
Statistics	1.24%	
Branch prediction	0.40%	√

Figure 5.5 DirectRSIM simulation overhead and sources of slowdown relative to Simple

DirectRSIM models an instruction window including fetching and retiring of instructions which consumes 36% of the execution time. The processor simulator also models a memory unit consisting of management of load and store queues which consumes nearly 10% of the execution time. Other features such as modeling of structural hazards and branch prediction hardware consume about 6.5% of the execution time. Thus, the processor simulation in DirectRSIM contributes nearly 53% of the total execution time. This is a large fraction of the overall overhead seen by DirectRSIM relative to Simple. At the same time, our experience shows that simulation of key processor features such as the instruction window and the memory unit is necessary in capturing the read miss overlap seen in various applications. Hence, modeling these features accurately contributes to the superior accuracy of DirectRSIM [17].

Some of the other tasks make minor or indirect contributions to the slowdown. For example, the direct execution portion is slower with DirectRSIM because the instrumentation code is more complex than in Simple. Cache simulation is somewhat slower in DirectRSIM since more MSHRs are likely to be used at any time, and the simulation of the memory, bus, and network is somewhat slower because of an increase in contention caused by the greater frequency of misses in DirectRSIM.

Overall, DirectRSIM is 2.7x slower than Simple-*i*x on average for the configurations and applications studied.

Chapter 6

Related Work

Chapter 2 provided a brief overview of current shared-memory simulation techniques including a discussion of direct execution simulators, detailed execution-driven simulators, and a study showing errors from simple-processor based simulators. This chapter focusses on other existing techniques for fast simulation of shared-memory multiprocessors.

DirectRSIM has sought to use direct-execution as a means to improve simulation performance. Dynamic binary translation is another technique that is commonly used to speedup simulation. Embra, MINT, and Shade are examples of simulators that use dynamic binary translation [24, 23, 5]. Dynamic binary translation can be seen as a form of direct execution as it involves executing large portions of the application directly on the host. Hence techniques presented in this study can also be applied to such simulators.

Sampling is another technique used to improve the speed of shared-memory system simulation. SimOS is an example of a simulation system that uses sampling. By varying the level of detail of simulation dynamically, SimOS avoids simulating the entire application in detail. This method, however, does not have an impact on the speed of accurate simulation of the important phases of the application. Thus, sampling is orthogonal to DirectRSIM and can be combined with DirectRSIM for further improvements.

A simulator can also be parallelized to run on a multiprocessor host for further performance improvements. The Wisconsin Wind Tunnel (WWT) is one such simu-

lator. WWT also makes additional approximations to achieve better parallelism [18]. Parallelization is also orthogonal to DirectRSIM.

Other research has focused on developing analytical models as a way of providing fast performance evaluation for ILP-based multiprocessors [2, 22]. Sorin et al. provide a customized approximate mean-value analysis model for ILP multiprocessors that explicitly accounts for read miss overlap in the workload and the effects of CC-NUMA multiprocessors [22]. This study seeks to tie the parameters of the model closely to the characteristics of given workloads. Simulation studies based on analytical modeling can be used to quickly narrow the architectural design space and account for key constraints. However, a more detailed performance evaluation generally requires simulation.

Concurrent with our work, a fast simulation technique has been developed for uniprocessor simulators which is currently not extendable to multiprocessor simulators in an obvious manner. Specifically, Schnarr et al. proposes two ideas [20]. First, they propose using direct execution in a manner similar to that presented in this thesis, but with a few differences. One difference is that they simulate mispredicted paths, as described in Section 7.2. Second, they propose fast forwarding, a technique that caches the actions of the simulator for a given state of the processor microarchitecture (the latency seen by each memory instruction in the instruction window is also part of the state). When this state repeats, the fast forwarding technique simply replays the actions from the simulator cache, eliminating much of the simulation overhead. They see low benefits from direct execution itself, but substantial performance improvements from fast forwarding. There are two key differences between the goals of the work by Schnarr et al. and this study. First, Schnarr et al. target uniprocessor systems and thus do not have to address the issue related to the value of non-blocking loads with direct execution that we address in Section 3.1. Second, they focus on accurate uniprocessor microarchitectural simulation, while this study focuses on accurate memory simulation in a multiprocessor configuration with only

as much emphasis on microarchitectural simulation as needed for correct memory simulation. This difference in approach allows us to make approximations, and is a reason that direct execution does not provide them the same benefits it provides this work. It is currently not clear that their fast forwarding optimization will benefit multiprocessor systems, since memory latencies in a multiprocessor are more variable and unpredictable than in a uniprocessor system, potentially decreasing the frequency of repeating the microarchitectural state. The speculative execution technique they describe is not directly applicable for multiprocessor simulation, but can be extended as described in Section 7.2. If their techniques are extensible to multiprocessors, then it should be possible to combine them with DirectRSIM as well for further benefits in accuracy and performance.

Chapter 7

Conclusions

7.1 Thesis Summary

Current simulators for shared-memory multiprocessor architectures involve a tradeoff between simulation speed and accuracy. Most simulators assume much simpler processors than the current generation of processors that aggressively exploit instruction-level parallelism (ILP). This can result in large simulation inaccuracies. A few newer shared-memory simulators model current ILP processors more accurately, but are about ten times slower.

This thesis presented a new simulation technique for shared-memory multiprocessors with ILP processors. The technique combines the speed advantages of simple-processor based simulators with accuracy similar to detailed ILP processor based simulation. It is based on a novel adaptation of direct execution, a widely used simulation methodology for shared-memory multiprocessors with simple processors. A simulator, called DirectRSIM, is developed based on the new technique. DirectRSIM extends current direct execution simulators in two important ways. First, DirectRSIM allows a data load that invokes the simulator to proceed in direct execution even before its simulation has completed at the memory system. Second, DirectRSIM provides an efficient timing simulator that accounts for the features of modern processors to aggressively exploit ILP (such as dynamic scheduling, multiple issue, and non-blocking loads). This methodology allows significant performance improvements relative to detailed execution-driven simulators by reducing simulator overhead in modeling instruction decoding, register renaming, memory disambiguation, specu-

lative execution, and instruction emulation. Nevertheless, the processor model of DirectRSIM includes enough detail to account for multiple-issue, out-of-order issue, and non-blocking loads.

An analysis of performance and accuracy in Chapter 5 shows that DirectRSIM sees an average of 1.6% error in simulated execution time relative to RSIM across all studied applications and configurations; at most, it sees an error of 6.3% in simulated execution time. At the same time, DirectRSIM sees a speedup of 4 relative to RSIM. In contrast, the best current simple-processor based simulation methodology sees large and variable errors in total execution time, ranging from 2% to 128%, and averaging 40%. The most commonly used simple-processor based simulation methodology sees errors ranging from 9% to 438%, averaging around 130%. Despite the superior accuracy of DirectRSIM, DirectRSIM sees only a factor of 2.7 slowdown compared to the current simple processor based simulation methodology. Thus, DirectRSIM achieves far greater accuracy than simple-processor based simulators for shared-memory multiprocessors while still maintaining performance competitive with these simulators.

In the late 1980's and early 1990's, the shared-memory architecture community made a shift from trace-driven simulation to execution-driven simulation based on studies showing inaccuracies in the results generated by trace-driven simulators. Simple-processor based direct-execution simulators became popular because of their speed and accuracy. However, the performance and accuracy results in this study indicate that the shared memory architecture community may again need to reconsider the appropriate simulation methodology for shared-memory systems. To date, the 10x speed advantage of direct execution simulators over detailed simulators like RSIM has been used to justify the potential for errors in using simple simulators to model systems built with state-of-the-art processors. However, the performance advantage of simple simulators drops to 2.7x if they are compared to DirectRSIM. While this factor is still significant, it is not clear that this advantage can offset the potential inaccuracies of such simulators in modeling application performance. Thus,

DirectRSIM substantially improves the speed vs. accuracy tradeoff in the simulation of shared-memory multiprocessors with ILP processors.

7.2 Future Work

Several features supported in other simulators can be added to DirectRSIM to improve performance and/or functionality. As discussed in Chapter 6, DirectRSIM performance can be directly improved with sampling or parallelization. Techniques such as executable editing or optimizing binary code translation could be used to make the instrumentation of application code both more efficient and more generally applicable [13, 24]. All of these techniques are independent of the underlying DirectRSIM methodology.

Additionally, DirectRSIM does not need to wait for the global clock to catch up to issue time on every L1 cache hit access; it should synchronize only if there is a possibility that the data could be replaced or invalidated from the L1 cache between the current value of the global clock and the issue time of the hit.

In DirectRSIM, each load or store executed in the application program currently calls the timing simulator. However, for data-race-free programs, this call overhead can be reduced by invoking the timing simulator from direct execution only at each synchronization, as these are the only logical points of interaction between processors. This would not reduce the total work done to simulate the instructions; but may reduce the function call overhead.

DirectRSIM could be further augmented to support TLB accesses, instruction cache, or operating system calls. We have not yet sought to support any of these because these are known to have little performance impact on our application suite. However, we are not aware of any fundamental problems with incorporating this support for DirectRSIM.

As discussed in Chapter 6, Schnarr et al. discusses how to incorporate support for speculative execution into direct execution simulators for ILP uniprocessor systems.

On a branch misprediction, the simulator saves the register state of the system and allows direct execution to progress on the wrong path. Additionally, each store that issues in direct execution must save aside a copy of the previous value of the memory location so as to restore the old value after the branch is resolved. This allows subsequent loads on the mispredicted path to read the value of the speculative store, while ensuring that the proper value can be returned to the memory after the branch is resolved. This technique is not directly applicable to a multiprocessor simulator like DirectRSIM because allowing a speculative store to write data in direct execution mode may lead to an unexpected data race with or corruption of data at another processor. One possible extension that would allow speculation in a direct-execution multiprocessor simulator would instrument the code to have speculative stores send their values to the timing simulator rather than writing their data into system memory; the timing simulator would then keep these values in a software-managed "write buffer." This "write buffer," rather than the system memory, would be used to service later speculative reads to the same address from the same processor. Reads from other processors would continue to read non-speculative values in the system memory as before.

Bibliography

[1] V. S. Adve et al. An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs. In *Proceedings of Supercomputing*, 1995.

[2] D. H. Albonesi and I. Koren. An Analytical Model of High-Performance Superscalar-Based Multiprocessors. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques*, 1995.

[3] E. A. Brewer et al. PROTEUS: A High-Performance Parallel-Architecture Simulator. Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, September 1991.

[4] E. Bugnion et al. Compiler-Directed Page Coloring for Multiprocessors. In *Proceedings of the 7th International Conference on Architectual Support for Programming Languages and Operating Systems*, 1996.

[5] R. F. Cmelik and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. Technical Report UWCSE 93-06-06, Computer Science Department, University of Washington, 1993.

[6] R. G. Covington et al. The Efficient Simulation of Parallel Computer Systems. *International Journal in Computer Simulation*, January 1991.

[7] H. Davis et al. Multiprocessor Simulation and Tracing Using Tango. In *Proceedings of the International Conference on Parallel Processing*, 1991.

[8] B. Falsafi and D. A. Wood. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[9] K. Gharachorloo et al. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the International Conference on Parallel Processing*, 1991.

[10] M. Heinrich et al. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[11] C. Holt et al. Application and Architectural Bottlenecks in Large Scale Distributed Shared Memory Machines. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, 1996.

[12] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, 1981.

[13] J. R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. In *Proccedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 1995.

[14] S. S. Mukherjee et al. Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulator. In *Workshop on Performance Analysis and Its Impact on Design (PAID)*, 1997.

[15] V. S. Pai et al. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[16] V. S. Pai et al. RSIM: A Simulator for Shared-Memory Multiprocessor and Uniprocessor Systems that Exploit ILP. In *Proceedings of the 3rd Workshop on Computer Architecture Education*, 1997.

[17] V. S. Pai et al. The Impact of Instruction Level Parallelism on Multiprocessor Performance and Simulation Methodology. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, 1997.

[18] S. K. Reinhardt et al. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1993.

[19] M. Rosenblum et al. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM TOMACS, Special Issue on Computer Simulation*, 1997.

[20] E. Schnarr and J. Larus. Personal communication. March 1998.

[21] J. P. Singh et al. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, March 1992.

[22] D. J. Sorin et al. Analytical Evaluation of Shared-Memory Parallel Systems with ILP Processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998. To appear.

[23] J. E. Veenstra and R. J. Fowler. MINT Tutorial and User Manual. Technical Report 452, Computer Science Department, University of Rochester, 1994.

[24] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1996.

[25] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

Appendix A

Performance Comparison of Simple Simulators

This appendix provides evidence that the performance of the baseline Simple and Simple-*i*x simulators used in this study is representative of current state-of-the-art simulators that use simple processor models.

The Simple simulator is compared with the recently released Wisconsin Wind Tunnel-II (WWT-2) [14]. WWT-2 is chosen for comparison since it has been used in many architectural studies and represents the state-of-the-art, it is publicly available, and it simulates SPARC executables similar to Simple. WWT-2 supports parallel simulation. Simple uses sequential simulation, however, since parallel performance is orthogonal to our comparison; parallel simulation technology could also be applied to all our simulators but that is beyond the scope of this work.

The performance of WWT-2 is compared with the performance of Simple for LU, FFT, and Radix with the input sizes used in this study (these are the common applications among those used in this study and in the application suite distributed with WWT-2). The simulators are run on the same hardware as the other simulations reported in this study. Similar parameters were used for both simulators to the extent possible. For example, a constant latency network configuration (100 cycle latency) and direct mapped caches were used in both simulators as these are the only options supported by the released version of WWT-2. Similarly, since WWT-2 only supports a single level cache, both cache levels in Simple were set to be the same size; all caches in both simulators are the L2 cache size used in the rest of this study.

Nevertheless, a completely fair comparison among the two simulators is difficult because of differences in modeled architectures. The most important difference for

our purposes is that the released version of WWT-2 does not support a CC-NUMA protocol. The closest protocol to CC-NUMA in WWT-2 is S-COMA; this protocol was used with the S-COMA hardware stache size of 320K based on a previous WWT-2 based study of coherence protocols [8]. The latter study showed that S-COMA is comparable to CC-NUMA for FFT, CC-NUMA performs worse than S-COMA for LU, and S-COMA performs worse than CC-NUMA for Radix [8].

Our performance measurements, summarized in Figure A.1, closely follow the above trend – WWT-2 and Simple show similar performance for FFT, Simple is significantly slower for LU, and WWT-2 is significantly slower for Radix. An interesting observation is that the total simulation time for all three applications is within 10% for both simulators.

Application	Simple	WWT-2
FFT	248 sec	263 sec
LU	394 sec	112 sec
Radix	395 sec	797 sec

Figure A.1 Simulation time for SPLASH-2 applications

Undoubtedly, there are many factors at play that contribute to the above results. However, our goal is to simply show that the Simple simulator is in the same class as other widely used simulators. The results in this section provide such evidence, and increase confidence that the experimental infrastructure of this study is indeed representative of the state-of-the-art.

Appendix B

Absolute Performance of Simulators in Instructions per Second

Figure B.1 gives the absolute performance in simulated instructions per second for all the simulators used in this study (data for Simple is not separately reported as it is similar to Simple-*i*x). The data reported here is for the same configurations and runs as reported in Figures 5.2 and 5.3.

	Erle.	FFT	LUopt	Mp3d	Radix	Avg.
Base	22.6	23.4	29.6	7.5	17.2	20.1
Lat. x2	20.5	18.4	25.0	6.8	13.6	16.9
Lat. x3	18.6	18.3	23.7	9.0	10.5	16.1
C. net	24.9	25.1	30.0	9.0	18.5	21.5
Avg.	21.6	21.3	27.1	8.1	15.0	18.6

(a) RSIM - Simulated kilo-instructions per second

	Erle.	FFT	LUopt	Mp3d	Radix	Avg.
Base	69.8	85.4	80.9	37.8	45.9	64.0
Lat. x2	73.7	88.5	76.5	46.6	43.9	65.8
Lat. x3	83.4	89.8	80.6	70.7	54.2	75.7
C. net	77.3	93.1	81.2	50.0	53.4	71.0
Avg.	76.0	89.2	79.8	51.3	49.4	69.1

(b) DirectRSIM - Simulated kilo-instructions per second

	Erle.	FFT	LUopt	Mp3d	Radix	Avg.
Base	250.6	205.4	214.8	32.5	93.6	159.4
Lat. x2	240.1	199.6	209.5	32.7	91.5	154.7
Lat. x3	215.1	198.4	196.6	27.7	75.2	142.6
C. net	280.7	243.1	228.5	50.3	126.4	185.8
Avg.	246.6	211.6	212.4	35.8	96.7	160.6

(c) Simple-*ix* - Simulated kilo-instructions per second

Figure B.1 Absolute performance of the simulators in kilo-instructions per second.