Appears in Proceedings of the Fifth International Symposium on High Performance Computer Architecture. January 1999.

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

Murthy Durbhakula, Vijay S. Pai, Sarita Adve
Department of Electrical and Computer Engineering
Rice University
Houston, Texas 77005
{murthy|vijaypai|sarita}@rice.edu

Abstract

Previous simulators for shared-memory architectures have imposed a large tradeoff between simulation accuracy and speed. Most such simulators model simple processors that do not exploit common instruction-level parallelism (ILP) features, consequently exhibiting large errors when used to model current systems. A few newer simulators model current ILP processors in detail, but we find them to be about ten times slower. We propose a new simulation technique, based on a novel adaptation of direct execution, that alleviates this accuracy vs. speed tradeoff.

We compare the speed and accuracy of our new simulator, DirectRSIM, with three other simulators – RSIM (a detailed simulator for multiprocessors with ILP processors) and two representative simple-processor based simulators. Compared to RSIM, on average, DirectRSIM is 3.6 times faster and exhibits a relative error of only 1.3% in total execution time. Compared to the simple-processor based simulators, DirectRSIM is far superior in accuracy, and yet is only 2.7 times slower.

1. Introduction

Shared-memory multiprocessors are a fast growing segment of the high performance computing and server market. Simulation is the most widely used technique to evaluate new shared-memory architectures. Recent advances in processor architecture, however, force a re-evaluation of current shared-memory simulation methodology. Current processors aggressively exploit instruction-level parallelism (ILP) through techniques such as multiple issue, out-of-order issue, non-blocking loads, and speculative execution. Most shared-memory simulation studies, however, use a much simpler model of the processor, assuming single-issue, in-order issue, blocking loads, and no speculative execution. We refer to the two types of processors as *ILP processors* and *simple processors* respectively.

Pai et al. showed that using current simple-processor based simulators to model ILP-processor based systems can give large and application-dependent errors (over 100% error in execution time in some cases) [9]. Unfortunately, the more accurate previous ILP-processor based simulators are much slower: we find an average slowdown of 9.7X.

The higher speed of simple-processor based simulators comes from the inherent benefits of a less complex processor, as well as from several speed enhancing techniques developed for such simulators. Direct execution is one such widely used technique that has previously relied on simple-processor features such as blocking loads, in-order issue, and no speculation [2, 3, 7].

This paper presents a novel adaptation of direct execution to substantially speed up simulation of shared-memory multiprocessors with ILP processors, without much loss of accuracy. We have developed a new simulator, DirectRSIM, based on our new technique. We evaluate the accuracy and speed of DirectRSIM by comparing it with RSIM, a state-of-the-art detailed ILP-processor based shared-memory simulator, as well as two representative simple-processor based direct execution simulators. For a variety of system configurations and applications, and using RSIM as the baseline for accuracy, we find:

- DirectRSIM, on average, is 3.6X faster than RSIM with an error in execution time of only 1.3% (range of -3.9% to 2.2%).
- Simple-processor based simulators remain an average of 2.7X faster than DirectRSIM. However, this addi-

*This work is supported in part by an IBM Partnership award, Intel Corporation, the National Science Foundation under Grant No. CCR-9410457, CCR-9502500, CDA-9502791, and CDA-9617383, and the Texas Advanced Technology Program under Grant No. 003604-025. Sarita Adve is also supported by an Alfred P. Sloan Research Fellowship and Vijay S. Pai by a Fannie and John Hertz Foundation Fellowship.

tional speed comes at a high cost, with average error in execution time of 46% (range of 0% to 128%) with the best simple-processor model, and average error of 137% (range of 9% to 438%) with the most common model.

Our results suggest a reconsideration of the appropriate simulation methodology for shared-memory systems. Earlier, the order-of-magnitude performance advantage of the simple-processor based simulators over RSIM made a compelling argument for their use in spite of their potential for large errors. It is not clear that those errors are still justifiable given only a 2.7X performance advantage relative to DirectRSIM.

2. Background

2.1. Direct execution with simple processors

Direct execution is a widely used form of execution-driven simulation, and has been shown to be accurate and fast for modeling shared-memory systems with simple processors [2, 3, 7].

Direct execution decouples functional and timing simulation. Functional simulation generates values (for registers and memory) and control flow, while timing simulation determines the number of cycles taken by the simulated execution. Direct execution achieves high speed in two ways. First, for functional simulation, it directly executes the application on the host. Second, timing for non-memory instructions is determined mostly by static analysis. The application is instrumented to convey this analysis to the memory timing simulator. Previous direct execution shared-memory simulators assume in-order issue and no speculation since they cannot model the effects of out-of-order issue statically and they view only one basic block at a time. With the exception of the Wisconsin Wind Tunnel II [7], these simulators also assume single-issue processors. Timing for memory references is modeled in detail, and is the most expensive part of the simulation.

For memory simulation, the application is usually instrumented to invoke the timing simulator on each memory reference, as these are the only points of interaction between the processors. When an application process invokes the timing simulator on a load, its functional simulation is suspended until the timing simulator completes the entire simulation of the load, thereby modeling only blocking loads. Stores are either modeled as blocking or non-blocking. In the non-blocking case, direct execution of the store's process may be resumed as soon as the appropriate simulation events for the store are scheduled (but not necessarily completed). The timing simulator can process these events asynchronously with respect to the store's process because, unlike a load, later instructions of the process do not depend on the completion of the store.

2.2. Simulators for ILP shared-memory systems

RSIM [8] and SimOS with the MXS processor simulator [10] are two previous shared-memory simulators that model ILP processors explicitly and in detail. They use straightforward execution-driven simulation, interpreting every instruction and simulating its effects on the complete processor pipeline and memory system in software.

Researchers have also used simple-processor based simulators to model ILP-processor based shared-memory systems using certain approximations. The most common approximation is to simply simulate a system with a simple processor to approximate a system with an ILP processor with the same clock speed (referred to as *Simple*). Other studies have sped up the clock rate of the simulated simple processor to model the benefits of ILP [5]. Pai et al. showed that the best previously used approximation is to speed up the processor clock cycle and L1 cache access time by a factor equal to the ILP processor's peak instruction issue rate (i) [9] (referred to as *Simple*-i*x*). They found that Simple-*ix* was reasonably accurate for some applications, but exhibited large errors in others. The key source of inaccuracy was that simple-processor based simulators do not model the impact of non-blocking loads (specifically, the overlapping of multiple load misses with each other).

3. Direct execution with ILP shared-memory multiprocessors

There are two problems with using previous direct execution techniques for ILP-processor based shared-memory systems:

Values for non-blocking loads. After a non-blocking load invokes the timing simulator, its direct execution process must be allowed to proceed before its timing simulation completes. This is required so that the direct execution process can generate later instructions for the timing simulator to execute in parallel with the load. However, the value that the load will return in the simulated architecture is unknown until the load's timing simulation is complete; this value depends on writes to the same location by other processors before the load reaches memory in the simulated architecture. Thus, the first problem is that the simulator must decide what value to return when a load occurs in the direct execution while it is incomplete in the timing simulation, and what action to take when the direct execution reaches a later instruction dependent on such a load.

Timing simulation of ILP features. The second problem is that a simple static analysis is insufficient to determine the impact of ILP features (such as out-of-order issue, speculative execution, and non-blocking loads) on the execution time of CPU instructions and on the time at which a memory instruction can be issued (or when it stalls the processor). Previous direct execution techniques do not directly

provide a way to account for these features.

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

3.1. Values for non-blocking loads

We focus on a release consistent architecture. For ease of explanation, we assume that synchronization accesses are identified to the simulator.

When a synchronization load invokes the timing simulator, it is treated as a blocking load as in previous direct execution simulators. When invoked by a data load, the timing simulator starts processing all instructions executed since its last invocation as described in Section 3.2. The timing simulator may return control to the direct execution before the load completes at (or even issues to) the memory hierarchy. The load returns the current value for the accessed memory location at the time of the direct execution, based on the following insight.

If the load does not form a data 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. A load that is not part of a data race (a non-race load) must be separated from any conflicting store by a chain of synchronization releases and acquires; these synchronization accesses are ordered as in previous direct execution simulators and enforce the necessary orderings among non-race accesses. Thus, for a non-race load, the value at the time of the direct execution can be safely returned and used by dependent instructions.

For a race load, the value returned may be different from the one that would be returned in the simulated architecture. This value would be legal for release consistency, but may not be possible on the simulated architecture. Since data races are generally rare in parallel programs, we expect this issue to not have a significant impact. Further, a system that obeys the data-race-free consistency model (which requires identifying data races for a guarantee of sequential consistency) and blocks on race loads can naturally be simulated with our technique without any error (by simply blocking on the race loads).

3.2. Timing simulation of ILP features

Like previous direct execution simulators, our technique performs the functional simulation directly on the host machine and invokes the timing simulator only on memory references. Unlike previous uses of direct execution, the application is instrumented to record the path taken by the direct execution since the previous invocation of the timing simulator by the same process. The timing simulator simulates the timing for this path with the goal of providing the best accuracy and performance possible.

A naive timing simulator would simply replicate the features of detailed simulators such as RSIM, modeling the register state, pipeline stages, and all instruction effects in detail. Instead, our timing simulator improves performance relative to RSIM in three ways. First, direct execution allows the timing simulator not only to avoid instruction emulation, but also to make use of the values determined in direct execution to speed up several parts of simulation (e.g., register renaming and memory disambiguation).

Second, we approximate some parts of the processor simulation, motivated by previous work that shows that the key characteristic in determining shared-memory multiprocessor performance is the behavior of the memory system and its interaction with the processor [9]. Our most significant approximation is that we do not simulate speculated execution paths that are mispredicted. This approximation does not preclude modeling other effects of speculation; e.g., we keep track of branch prediction tables and stall instruction fetch on a mispredicted branch as the processor waits for the branch to be resolved. The simulation speed benefits of this approximation cannot be exploited by detailed simulators such as RSIM since RSIM does not know if a prediction is correct until the prediction is actually resolved in the simulated execution. The timing simulator of DirectRSIM has this information at the time the prediction is made, based on the values generated by the direction execution.

Third, with direct execution, the different application processes execute asynchronously in the simulation. In contrast, RSIM's processor and cache simulation, due to its detailed nature, is inherently a cycle-by-cycle simulation in which all processors and caches proceed in lockstep (Section 4.3). We improve performance of our timing simulation by further increasing the asynchrony in our system, partly by exploiting the features described above. The next section provides further details.

3.3. Implementation of DirectRSIM

DirectRSIM implements the direct execution methodology described in Sections 3.1 and 3.2. 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

The instrumentation code calls the timing simulator on each memory reference and provides it with the execution path to be processed. The path is represented as ranges of contiguous program-counter values traversed by the direct execution since the last invocation of the timing simulator. For this purpose, the instrumentation code marks each unconditional branch or taken path of a conditional branch as ending a program-counter range and starting a

new range. We currently instrument the application assembly code, but could also use the more general methods of executable-editing or dynamic binary translation.

3.3.2 Timing simulator

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 simulator are common to all our simulators, and are described in more detail in Section 4. The processor simulator is the key feature that sets DirectRSIM apart. Upon entry, the DirectRSIM processor simulator processes the execution path provided by the instrumentation code, attempting to bring each instruction from the path into its instruction window.

Key functionality, data structures, and simulation clocks. The key work 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[1], (2) retiring instructions from the instruction window at appropriate times based on the above completion times, (3) maintaining branch prediction tables, and (4) memory 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).

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 track memory accesses that need to be issued, (3) a structure to track outstanding stores, hashed on their addresses for efficient forwarding (4) the branch prediction table, and (5) a structure for tracking structural hazards for functional units.

The memory system and event-driven simulation engine of DirectRSIM provide a global view of time in the system. However, unlike RSIM, the processors are not required to be in lockstep with the global clock when performing internal actions. Each 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 (as further explained below).

Instruction issue and completion. As the processor simulator brings instructions into its simulated instruction window, 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 directly or indirectly data dependent on any incomplete loads. For such an instruction, the completion timestamp depends on its latency and the availability of a functional unit. The latter is approximated by tracking the future use of functional units by instructions whose completion times are already known; it is possible that some instructions with unknown completion times may interfere with the current instruction, but this effect is not modeled. If an instruction's completion time is not immediately known, it 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. If the issue time is not known (due to dependencies on incomplete loads), then the load is attached to the instructions on which it is dependent and inserted into the load queue on completion of these instructions. 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. This check is efficient since addresses for all previous stores are immediately known through direct execution, and can be stored and matched through a hash table. If there is no forwarding, an event is scheduled for issuing the load to the memory system. On forwarding, a completion time is marked for the load.

As with most current processor simulators, to ensure 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, an event for the issue of the store is scheduled.

Instruction fetch and retirement. Instruction fetching continues until either the instruction window or the load queue or the store queue fills up, or all instructions executed by the functional simulator since the last timing simulator invocation are processed, or there is a misspeculation. In the misspeculation case, instruction fetching continues once the misspeculation penalty is determined. In the case that the instruction window is full, the processor simulator tries to retire the first set of instructions. Retirement is an entirely local action; the head of the instruction window can always be retired unless it is an incomplete load. The processor's retire clock is possibly updated based on the com-

[1] These dependences are the primary reason for DirectRSIM's processor model. Although previous work has shown that the Simple-*i*x model (Section 2.2) can predict the CPU component of execution time reasonably well for the applications studied [9], we do not use its processor model because its policy of simply speeding the CPU clock based on issue rate would allow multiple non-blocking loads to be issued even if there was a dependence from one to the next.

pletion time of the retiring instruction and the number of instructions that have already retired at that time relative to the processor's peak retire rate.

Suspending and resuming processor simulation. A processor's simulation (and its corresponding direct execution process) is suspended when its instruction window is full, it cannot retire any further instructions, 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 so far have been entered in its instruction window.

4. Evaluation Methodology

4.1. Simulated architectures

We model CC-NUMA shared-memory multiprocessors. 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, 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 multiple issue, out-of-order issue, non-blocking loads and stores, speculative execution, and register renaming. Since most previous direct execution simulators model only single cycle functional unit latencies, we assume the same. Both caches are non-blocking and use miss status holding registers (MSHRs) to store state for outstanding accesses. The L1 and L2 cache sizes follow the methodology of Woo et al. [13] for our application input sizes (described in Section 4.2). 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 instruction cache and TLB are modeled since the application suite is known to have a small instruction cache and TLB miss ratio. Figure 1 summarizes the key system parameters for our base system. Results for five variations of the base system are also reported, as described in Section 5.

ILP processor and cache parameters	
Processor speed	500MHz
Maximum fetch/decode/retire rate	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 line size	64 bytes
L1 cache (on-chip)	Direct mapped, 16 K
L2 cache (off-chip)	4-way associative, 64 K
L1 request ports	2
L2 request ports	1
Number of MSHRs at L1 and L2	8
Representative contentionless latencies	
L1 cache hit	1 cycle
L2 cache hit	10 cycles
Local memory	85 cycles
Nearest remote memory	182 cycles
Farthest remote memory	262 cycles
Nearest cache to cache transfer	210 cycles
Farthest cache to cache transfer	309 cycles

Figure 1. Base system parameters. The number of processors varies by application, as described in Section 4.2 and Figure 2.

Application	Input Size	Processors
Erlebacher	64x64x64 cube, block 8	16
FFT	65536 points	16
LU	256x256 matrix, block 8	8
Radix	512K keys, max: 512K, 1024	8
Mp3d	50000 particles	8

Figure 2. Application input sizes and number of simulated processors.

4.2. Applications

We study 5 applications – FFT, LU, and Radix from SPLASH-2 [13], MP3D from SPLASH [12], and Erlebacher from the Rice parallel compiler group [1]. A few changes have been made to the original SPLASH-2 codes for better performance. In LU, one loop nest is interchanged to cluster read misses closer together, thereby increasing their overlap with each other and improving performance in a system with ILP processors [9]. A similar change is applied to two loop nests in FFT. For better load balance, we use flags instead of barriers for synchronization in LU.

Figure 2 lists the input data sizes (chosen so that the simulations complete in reasonable time) and the number of processors simulated for each application (based on the scalability of the application for the input size used).

4.3. Simulators

We compare DirectRSIM with RSIM (the only publicly available detailed ILP-processor based shared-memory simulator), and Simple and Simple-*i*x (two representative simple-processor based direct execution simulators). RSIM and DirectRSIM directly model the ILP processor described in Section 4.1. Simple and Simple-*i*x use a simple-processor model to approximate the ILP processor, using previous direct execution methodology (Section 2). We chose these two simple-processor approximations since

they are the most widely used and the best reported such approximations respectively [9]. Recall that to model the 500 MHz 4-way base ILP processor, Simple-4x models a 2 GHz single-issue processor. To ensure that the performance of our simple-processor based simulators is representative of the state-of-the-art, we compared Simple to the recently released Wisconsin Wind Tunnel-II (a simple-processor based direct execution simulator) and found the speed of the two simulators to be comparable [4].

The differences between our four simulators are limited to the processor model and its interaction with the cache hierarchy. The memory system simulation in all simulators uses nearly identical code. It is based on an event-driven simulation engine [2], where events for the simulated system modules are scheduled by inserting them on a central event queue, and are triggered by a central driver routine.

A few differences between RSIM and the other simulators arise because of the inherent differences between detailed and direct execution based simulation. The direct execution simulators use user-level lightweight processes to provide the register and stack state needed by each simulated processor for direct execution. Each activation of a process incurs the overhead of a lightweight context-switch. RSIM does not use lightweight processes, as it simulates all register and stack state in software. Instead, it uses a special event that 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. Effectively, RSIM simulates the processors and caches on a cycle-by-cycle basis, since in a detailed ILP processor simulation it can be expected that some processor or cache will have some event scheduled every cycle.

Additionally, the direct execution simulators optimize L1 cache hits whenever the cache is guaranteed to have ports available and not be stalled for resources such as MSHRs. In these cases, the processor simulator itself accounts for the impact of the hit on execution time without forwarding the request to the L1 cache module. The processor may, however, still have to stall to allow the global simulation clock to catch up with the issue time of such an access. RSIM issues all hits to the caches, consistent with its cycle-by-cycle detailed simulation policy.

4.4. Metrics

The accuracy of a direct execution simulator is evaluated based on the execution time it reports for the simulated application (excluding initialization), relative to the time reported by RSIM. Since all simulators use nearly identical code for the memory system, the discrepancy in simulated execution times occurs solely due to the level of detail in the processor models. To gain further insight, we also report three components of the execution time – CPU time, memory stall time, and synchronization stall time – calculated as in previous work [9, 10].

To determine simulator performance, the elapsed (wall-clock) time is measured for each simulation when run on an unloaded single 250MHz UltraSPARC-II processor of a Sun Ultra Enterprise 4000 server with 1GB memory and 1MB L2 cache. The simulators were all compiled using the Sun C 4.2 compiler with the highest practical level of optimization. The time spent in the initialization phase of the application is not included, since this time is not reported in the simulated execution time and can be sped up in various ways orthogonal to the rest of the simulation methodology.

5. Results on simulator accuracy

5.1. Base system configuration

Figure 3 shows the simulated execution time and its components reported by each simulator for each application on the base system configuration, normalized to that for RSIM. The number above each bar in the figure gives the percentage error in total execution time relative to RSIM. Numbers shown at the side of a bar represent the breakup of the total error among the three components of execution time.

Figure 3 shows that DirectRSIM reports overall simulated execution time very close to RSIM on all our applications, with a maximum error of 2.2%. This is a striking improvement over the best previous approximation of Simple-4x, which sees an execution time error of 87% for LU and 25% to 33% on three other applications studied. The Simple simulator sees much larger errors, ranging from 47% to 271%.

The differences between the four simulators arise from their abilities to capture the benefits that ILP provides to the various components of execution time. As discussed in [9], ILP reduces the CPU component of execution time by issuing multiple instructions at a time and by issuing instructions out of order. ILP reduces the memory component primarily by overlapping multiple long latency memory operations with each other, or also by overlapping memory latency with CPU instructions. ILP can also increase the memory component by increasing contention for resources or by changing an access pattern. Synchronization time is negligible for all our applications, and is not discussed further.

As reported by Pai et al. [9], the Simple model cannot capture the effects of ILP on either the CPU or memory stall component of execution time. Simple-4x models much of the benefit for the CPU component (because its clock speed is increased by a factor equal to the issue width of the processor). Most of the errors seen by Simple-4x are in the memory stall component, primarily because Simple-4x does not allow multiple read misses to overlap with each other. Thus, this method cannot properly capture

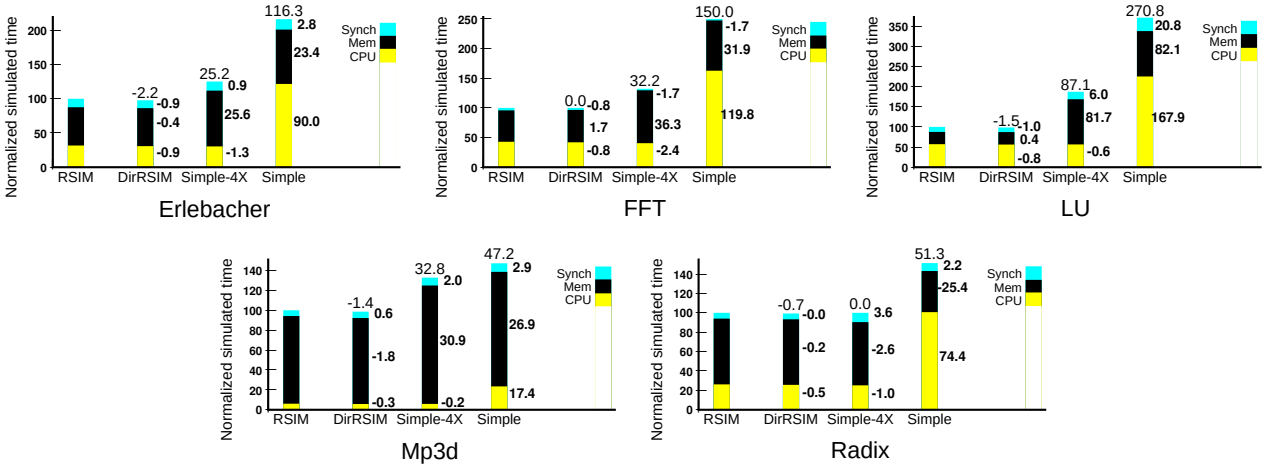

Figure 3. Simulator accuracy for the base system.

Config.	Difference from the base configuration
Lat. x2	Roughly twice the local and remote memory latencies.
Lat. x3	Three times the local memory latency, and a minimum contentionless remote-to-local latency ratio of 3:1.
ILP+	Processor is twice as aggressive, with double the instruction issue width, instruction window size, processor memory unit size, functional units, branch-prediction hardware, cache ports, and MSHRs.
ILP++	Same as ILP+, but with four times the instruction window size, memory unit size, and MSHRs as the base.
C. net	Constant-latency 50-cycle network instead of a 2-D mesh network.

Figure 4. Variations on base configuration.

	Erle.	FFT	LU	Mp3d	Radix	Avg.
Base	-2.2	0.0	-1.5	-1.4	-0.7	1.2
Lat. x2	-1.6	-1.5	-2.0	-0.7	0.0	1.2
Lat. x3	-0.7	2.2	-0.7	0.0	-0.3	0.8
ILP+	-2.8	0.2	-3.5	-3.9	-0.8	2.2
ILP++	0.7	-1.2	-2.4	-0.9	-0.8	1.2
C. net	-1.5	-0.8	-1.6	-0.5	-0.5	1.0
Avg.	1.6	1.0	1.9	1.2	0.5	1.3

(a) % error in execution time for DirectRSIM relative to RSIM

	Erle.	FFT	LU	Mp3d	Radix	Avg.
Base	25.2	32.2	87.1	32.8	0.0	35.5
Lat. x2	27.5	35.0	109.0	31.9	3.6	41.4
Lat. x3	27.6	38.4	90.5	23.3	2.0	36.4
ILP+	31.4	50.0	122.4	58.6	4.0	53.3
ILP++	69.8	58.8	127.8	98.2	10.1	72.9
C. net	23.1	29.7	84.7	28.1	3.6	33.8
Avg.	34.1	40.7	103.6	45.5	3.9	45.5

(b) % error in execution time for Simple-ix relative to RSIM

	Erle.	FFT	LU	Mp3d	Radix	Avg.
Base	116.3	150.0	270.8	47.2	51.3	127.1
Lat. x2	77.7	99.5	232.4	38.8	22.7	94.2
Lat. x3	54.5	73.4	147.6	25.8	9.1	62.1
ILP+	156.3	227.8	425.1	78.2	68.0	191.1
ILP++	231.2	247.1	437.8	122.8	77.8	223.3
C. net	110.6	145.8	264.9	38.3	55.9	123.1
Avg.	124.4	157.3	296.4	58.5	47.5	136.8

(c) % error in execution time for Simple relative to RSIM

Figure 5. Simulator accuracy for all configurations. (Averages are over absolute values of the errors.)

ILP-specific improvements in the memory stall component of execution time. DirectRSIM models the impact of ILP in both CPU and memory stall components of execution time, and provides a closer and more consistent approximation to the functionality of detailed execution-driven simulators.

5.2. Other system configurations

Figure 4 summarizes the variations on the base system configuration studied in this section. These configurations are intended to capture future trends towards higher processor clock speeds, larger remote to local memory latency ratios, aggressive processor microarchitectures, and aggressive network configurations. In the ILP+ and ILP++ configurations, Simple-8x is used rather than Simple-4x.

Figures 5(a), (b), and (c) show the percentage errors in total execution time relative to RSIM as seen by DirectRSIM, Simple-ix, and Simple respectively for the various system configurations (the first row in the tables repeats the data of the base configuration shown in Figure 3). DirectRSIM continues to see very low errors, with an average of 1.3% and a maximum of 3.9%. In contrast, the errors with Simple-ix remain high for most of the applications, and continue to vary widely, ranging from 0% to 128%, with an average of 46%. The errors seen with Sim-

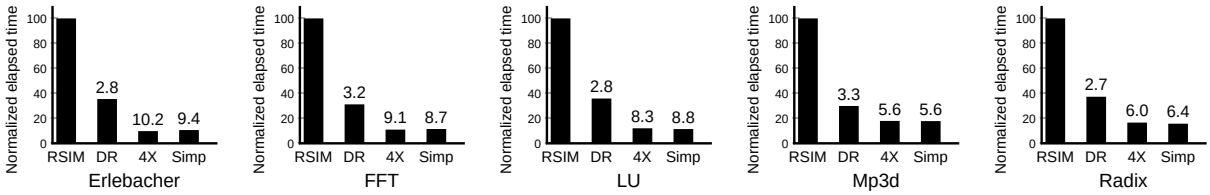

Figure 6. Simulator performance for the base system. DR=DirectRSIM, 4X=Simple-4X, Simp=Simple.

ple are even higher, ranging from 9% to 438%, averaging 137%. As with the base configuration, most of the error with Simple-ix comes from the memory component, while the error with Simple comes from both the CPU and the memory component [4]. As expected, the errors are greatest in the applications with the most read miss overlap. This application characteristic becomes even more important for systems with future aggressive processors (e.g., ILP+ and ILP++), as seen by the increase in error with Simple and Simple-ix for these configurations.

In conclusion, DirectRSIM achieves significantly greater and more reliable accuracy than Simple-ix or Simple in a variety of current and future multiprocessor configurations.

5.3. Applicability to architectural studies

So far, we have evaluated the simulators based on their ability to predict absolute execution times and the fraction of time stalled for memory. The latter is particularly important for a large class of architectural studies that target the memory stall component.

In some architectural studies, accurately modeling relative gains of an optimization may be more important than accurately modeling absolute execution time. We evaluate DirectRSIM and Simple-4x on the base configuration for their ability to predict the benefits of an example optimization. Recall that our version of LU has been optimized with a loop interchange to increase the overlap of read misses with each other. We compare the reduction in execution time due to this optimization reported by the simulators. We find that RSIM reports a reduction of 26%. DirectRSIM closely follows RSIM showing a reduction of 23%. In contrast, Simple-4x reports no reduction in execution time, as it does not model the benefits of read miss overlap. Therefore, unlike DirectRSIM, Simple-4x is unable to predict the benefits of the optimization.

6. Results on simulator performance

6.1. Overall performance

Figure 6 graphically depicts the elapsed times for the four simulators in the base configuration for each application, normalized to the time for RSIM. The number above the bars for DirectRSIM, Simple-ix, and Simple are the

	Erle.	FFT	LU	Mp3d	Radix	Avg.
Base	2.8	3.2	2.8	3.3	2.7	3.0
Lat. x2	3.3	3.9	3.0	4.8	3.2	3.6
Lat. x3	3.7	3.8	3.5	5.9	4.8	4.3
ILP+	3.8	4.4	3.1	3.7	3.2	3.6
ILP++	2.9	4.2	3.2	5.0	3.7	3.8
C. net	3.1	3.3	3.0	4.4	2.7	3.3
Avg.	3.3	3.8	3.1	4.5	3.4	3.6

(a) Speedup of DirectRSIM over RSIM

	Erle.	FFT	LU	Mp3d	Radix	Avg.
Base	3.6	2.8	3.0	1.7	2.2	2.7
Lat. x2	3.3	2.8	2.8	1.6	2.5	2.6
Lat. x3	3.0	3.8	2.6	2.1	2.7	2.8
ILP+	3.2	2.7	2.9	1.6	2.2	2.5
ILP++	4.0	2.8	3.0	1.6	2.9	2.9
C. net	3.6	3.1	3.3	2.0	3.0	3.0
Avg.	3.4	3.0	2.9	1.8	2.6	2.7

(b) Speedup of Simple-ix over DirectRSIM

	Erle.	FFT	LU	Mp3d	Radix	Avg.
Base	10.2	9.1	8.3	5.6	6.0	7.8
Lat. x2	10.9	10.9	8.3	7.8	8.0	9.2
Lat. x3	11.2	14.5	9.2	12.6	12.8	12.1
ILP+	12.1	11.6	9.1	6.0	7.3	9.2
ILP++	11.7	11.7	9.5	8.2	10.8	10.4
C. net	11.0	10.1	9.8	8.9	8.0	9.6
Avg.	11.2	11.3	9.0	8.2	8.8	9.7

(c) Speedup of Simple-ix over RSIM

Figure 7. Simulator performance for all configurations.

speedups achieved by those simulators over RSIM. Since the elapsed times for Simple and Simple-ix are similar in all cases and since Simple gives much larger errors, we do not discuss the performance of Simple any further. Figure 7 tabulates speedup for each pair of simulators, for all configurations in Figure 4. As reference for absolute performance, RSIM simulates an average of 20,000 instructions per second for the base configuration (more data appears in [4]).

Simple-ix gives the best elapsed time, with an average speedup of 9.7 over RSIM. DirectRSIM has some additional overheads from processor simulation, but still sees an average speedup of 3.6 over RSIM. Of particular interest are the increases in DirectRSIM speedup for the longer-latency configurations, which represent future configura-

tions with faster processor speeds. DirectRSIM profits by switching from a largely cycle-driven simulator to a purely event-driven simulator, and so is less sensitive to future increases in system latencies than RSIM. DirectRSIM also sees higher speedups in ILP+ and ILP++ by effectively targeting the more expensive processor simulation component seen by these aggressive microarchitectures.

Most notably, the performance advantage of Simple-ix is reduced to an average of 2.7X compared to DirectRSIM. The competitive performance of DirectRSIM indicates that the performance benefits of simple-processor based simulators may no longer be enough to justify their large inaccuracies in modeling current and future multiprocessor systems.

6.2. Detailed analysis of DirectRSIM's performance

To further understand the reasons for the performance differences among the simulators, Figure 8 depicts their execution profiles for LU on the base configuration as reported by `prof` (with monitoring turned on only during the parallel phase of the application). The other applications show similar profiles. The function calls of the simulators are divided according to the logical tasks they perform. From the bottom to the top of each bar, these tasks are instruction fetch and decode (including dependence checking), instruction retirement, processor memory unit simulation, functional unit management, cache simulation, cycle-driven simulation management, instruction emulation, direct-execution, event-driven simulation management, context switching among lightweight processes, and other tasks (e.g., memory and network simulation, and branch speculation). Not all tasks are present in all simulators.

DirectRSIM vs. RSIM. DirectRSIM improves performance relative to RSIM primarily by reducing the time spent simulating instruction fetch and decode, instruction retirement, the processor memory unit, and functional unit management. DirectRSIM's knowledge of values and addresses through direct execution enables more efficient register renaming and management of store-to-load forwarding, respectively. The provision to allow internal processor actions to proceed ahead of the global clock enables more efficient instruction fetching and retirement. The instruction dependence checking for issue is sped by the use of timestamps. Functional unit management is sped by the structure to approximately track future functional unit utilizations.

DirectRSIM also spends less time than RSIM in cache simulation by not simulating accesses that are known to hit in the L1 cache without contention. Among the remaining components of elapsed time (accounting for less than 20% of RSIM's total time), DirectRSIM eliminates the cycle-driven controller, but adds a component to handle context-switching and also increases event-driven simulation overhead. DirectRSIM avoids the overhead of instruction emulation (about 4% of RSIM time) and replaces it with a smaller component in direct execution. In the "other" category, DirectRSIM also uses values computed in direct execution to reduce the cost of mispredicted branches.

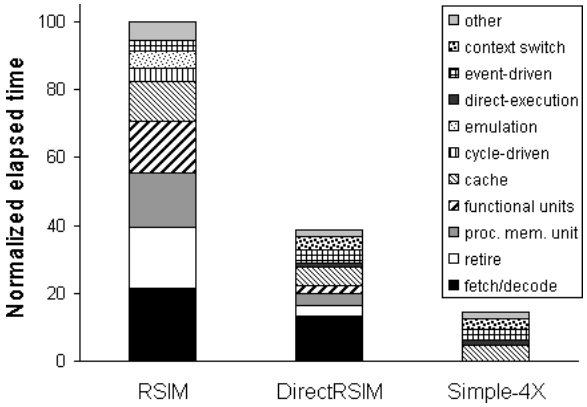

Figure 8. Components of elapsed time.

DirectRSIM vs. Simple-4x. As expected, most of DirectRSIM's overhead relative to Simple-4x stems from its processor simulation features. It also sees slightly more overhead in memory hierarchy simulation (due to increased resource contention from non-blocking reads).

7 Related Work

Section 2 reviewed the previous shared-memory simulation techniques most relevant to this paper. Additionally, sampling [10] and parallelization [7] are used to speed up shared-memory simulation. Both techniques are orthogonal to ours and can be used in conjunction with DirectRSIM.

Dynamic binary translation is sometimes used to speed up simulation [10] (as an alternative to direct execution). For our purposes, this technique can also be seen as a form of direct execution as it also decouples functional and timing simulation and executes most of the translated application directly on the host. Hence, the techniques presented in this paper can also be applied to dynamic binary translation.

Effectively, DirectRSIM's timing simulator acts as a trace-driven simulator operating on the trace of instructions executed since its last invocation by the same process. DirectRSIM, however, is still execution-driven because the simulated application's execution path is affected by the dynamic ordering of synchronization accesses and contention. In the uniprocessor case, however, DirectRSIM effectively becomes a trace-driven simulator.

Concurrently, Krishnan and Torrellas have proposed a method similar to ours for direct-execution for ILP multiprocessors [6]. They do not discuss the potential for error (or solutions) when using values of non-blocking loads in direct execution. They also do not assess the accuracy of their simulator or compare performance with detailed simulation. Their performance comparison with a previous

simple-processor simulator is done without memory system simulation, and shows slowdowns of 24–29X.

Schnarr and Larus concurrently developed a direct execution simulator for uniprocessors with ILP processors [11]. They simulate mispredicted paths and also propose instruction-window memoization. The use and/or benefits of some of their techniques for shared-memory multiprocessors are unclear (e.g., speculative stores and memoization). Further, their approach focuses on accurate microarchitectural simulation. We allow approximations, since we focus on accurate memory simulation in a multiprocessor with only as much emphasis on microarchitectural simulation as needed for correct memory simulation.

8 Conclusions

This paper presents a new simulation technique for shared-memory multiprocessors with ILP processors that combines the speed advantages of simple-processor based simulators with the accuracy of detailed ILP-processor based simulators. Our technique is based on a novel adaptation of direct execution. First, it allows a data load to proceed in direct execution even before its simulation has completed at the memory system. Second, it provides an efficient timing simulator that accounts for aggressive ILP features such as multiple issue, out-of-order issue, and non-blocking loads.

DirectRSIM, our implementation of the new technique, sees an average of 1.3% error (maximum of only 3.9%) in simulated execution time relative to RSIM for all studied applications and configurations. At the same time, DirectRSIM sees a speedup of 3.6 over RSIM. In contrast, the best current simple-processor based simulation methodology sees large and variable errors in execution time, ranging from 0% to 128%, and averaging 46%. The most commonly used simple-processor based simulation methodology sees errors ranging from 9% to 438%, averaging 137%. Despite its superior accuracy, DirectRSIM sees only a factor of 2.7X slowdown compared to current simple-processor based simulators. Although the performance advantage of simple-processor based simulators is still significant, it may no longer be enough to justify their high errors. Our results, therefore, suggest a reconsideration of simulation methodology for evaluating shared-memory systems.

In the future, several features supported in other simulators can be added to DirectRSIM to further improve its performance and/or functionality. Examples include parallelization, sampling, instrumentation through executable editing or binary translation, instruction cache, TLB, and full simulation of system calls. We are not aware of any fundamental problems in incorporating such support for DirectRSIM. Finally, if desired, we believe that support for simulating mispredicted paths could also be incorporated.

Acknowledgments

We thank Jim Larus and Parthasarathy Ranganathan for valuable comments on earlier versions of this paper.

References

[1] V. S. Adve et al. An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs. In *Proc. Supercomputing '95*, December 1995.

[2] R. G. Covington et al. The Efficient Simulation of Parallel Computer Systems. *Intl. Journal in Computer Simulation*, 1:31–58, January 1991.

[3] H. Davis, S. R. Goldschmidt, and J. Hennessy. Multiprocessor Simulation and Tracing Using Tango. In *Proc. Intl. Conf. on Parallel Processing*, pages II–99–II107, 1991.

[4] M. Durbhakula, V. S. Pai, and S. V. Adve. Improving the Speed vs. Accuracy Tradeoff for Simulating Shared-Memory Multiprocessors with ILP Processors. Technical Report 9802, Department of Electrical and Computer Engineering, Rice University, April 1998. Revised November 1998.

[5] C. Holt, J. P. Singh, and J. Hennessy. Application and Architectural Bottlenecks in Large Scale Distributed Shared Memory Machines. In *Proc. 23rd Intl. Symp. on Computer Architecture*, pages 134–145, May 1996.

[6] V. Krishnan and J. Torrellas. A Direct-Execution Framework for Fast and Accurate Simulation of Superscalar Processors. In *Proc. Parallel Architectures and Compilation Techniques*, October 1998.

[7] 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*, June 1997.

[8] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM Reference Manual version 1.0. Technical Report 9705, Department of Electrical and Computer Engineering, Rice University, August 1997.

[9] V. S. Pai, P. Ranganathan, and S. V. Adve. The Impact of Instruction Level Parallelism on Multiprocessor Performance and Simulation Methodology. In *Proc. Intl. Symp. on High Performance Computer Architecture*, pages 72–83, 1997.

[10] M. Rosenblum et al. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM Transactions on Modeling and Computer Simulation*, 1997.

[11] E. Schnarr and J. Larus. Fast Out-Of-Order Processor Simulation Using Memoization. In *Proc. 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 283–294, October 1998.

[12] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[13] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. 22nd Intl. Symp. on Computer Architecture*, pages 24–36, June 1995.