Redeeming IPC as a Performance Metric for Multithreaded Programs

Kevin M. Lepak, Harold W. Cain, and Mikko H. Lipasti
Electrical and Computer Engineering and Computer Sciences Department
University of Wisconsin
Madison, WI 53706
{lepak,mikko}@ece.wisc.edu, cain@cs.wisc.edu

Abstract

Recent work has shown that multithreaded workloads running in execution-driven, full-system simulation environments cannot use instructions per cycle (IPC) as a valid performance metric due to non-deterministic program behavior. Unfortunately, invalidating IPC as a performance metric introduces its own host of difficulties: special workload setup, consideration of cold-start and end-effects, statistical methodologies leading to increased simulation bandwidth, and workload-specific, higher-level metrics to measure performance. This paper explores the non-determinism problem in multithreaded programs, describes a method to eliminate non-determinism across simulations of different experimental machine models, and demonstrates the suitability of this methodology for performing architectural performance analysis, thus redeeming IPC as a performance metric for multithreaded programs.

1 Introduction

For many years, simulation at various levels of abstraction has played a key role in the design of computer systems. In particular, detailed, cycle-accurate simulators are widely used to estimate performance and make design trade-offs for both real and proposed designs by academics and industry practitioners alike. There appears to be broad consensus that such simulators ought to be execution-driven, in order to capture the effects of wrong-path instructions. However, the introduction of wrong-path instructions in execution-driven simulation invalidates the naive use of the *instructions-per-cycle* (IPC) metric, since changes in branch predictor configuration or branch resolution latency can increase or decrease the number of instructions fetched and executed. A similar problem exists for ISAs with compiler controlled speculation (for example delayed exception semantics or predicated instruction execution in IA-64 [14].) Fortunately, there is a simple solution: namely, only counting committed instructions, or instructions guaranteed to be present across all machine configurations in this metric. As long as the simulated program has a well-defined beginning and ending, the committed instruction count should be the same across machine models, hence enabling the continued use of IPC as a valid and intuitively appealing performance metric. More recently, advances in simulation technology have enabled execution-driven simulators that are also capable of simulating both supervisor and user code in order to exercise all aspects of the system design. Full-system simulators such as SimOS [18] and Simics [13] are capable of booting and running full-fledged multitasking operating systems. Hence, execution-driven simulation can be used to faithfully model not just the behavior of user code, but also kernel-mode paths through the operating system, eliminating errors in accuracy that can exceed 100% [6].

However, as pointed out in [2], even minute changes in the simulated configuration can cause dramatic spatial variation in the executed instruction stream in a full-system simulator. In a uniprocessor system, the catalyst for this variation is the arrival time of asynchronous interrupts: if the changes in machine model either accelerate or decelerate instruction commit rate, an interrupt occurring at a fixed point in simulated time will be serviced at differing instruction boundaries across the different simulations. The operating system task dispatcher which is invoked by the interrupt handler may then respond to these differences by choosing to schedule a different ready task, leading to dramatic spatial variation [3]. In a multiprocessor system, this problem is compounded by data and/or synchronization races in multithreaded programs. If changes in the simulated machine model accelerate processors at nonuniform rates, a race that was won by processor A in one simulation may now be won by processor B instead. Once again, dramatic spatial variation in program execution can result. Since most current and future general-purpose processors are designed to operate in multiprocessor systems, or are multithreaded [5, 12], there is an urgent need for a tractable simulation methodology for such designs.

The fundamental problem created by this nondeterministic behavior is that accurate performance comparisons can no longer be made across simulations with varying machine models, since there is no longer any guarantee that a comparable type and amount of work was performed in both simulations. In other words, instruction count is no longer a reliable measure of work, and hence, IPC loses its validity as a meaningful performance metric. Therefore, higher-level metrics are needed to measure the amount of work performed in each simulation. For example, standalone programs may need to run end to end, eliminating the attractive option of time-domain sampling, or transaction-based workloads may need to commit a certain number of transactions. Ultimately, performance must be measured in transactions per cycle, queries per cycle, or some other high-level metric instead of instructions per cycle. Such coarse-grained simulations can also suffer from cold-start and end effects, since there is no easy way to guarantee that the set of transactions completed in each simulation were in comparable stages at the beginning of the simulations. In other words, a database may have 1000 in-flight transactions at the beginning of a simulation, and 100 of those transactions may be 90% complete, while the remaining 900 are only 10% complete. One simulation may complete the 100 nearly-ready transactions, and then exit, while another may complete 100 transactions that had barely started to execute. As a result, there can be dramatic variation in the amount of work actually performed, even though the high-level metric of completed transactions is the same. Figure 1 illustrates this variation for multiple simulations of the SPECjbb2000 [19] benchmark running on 16 processors and completing 400 transactions. This

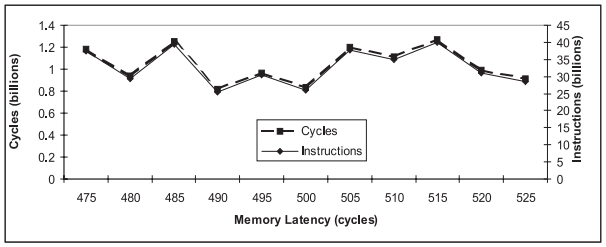

FIGURE 1. End-to-end simulated cycles (SPECjbb2000) for varying main memory latency. Main memory latency is varied from 475 to 525 cycles for a 16 processor run of SPECjbb2000 for 400 transactions starting from the same checkpoint. The measured number of cycles and instructions to complete the run is indicated, showing substantial variation in performance measurement for a minor architectural change.

effect can be reduced by counting a large number of transactions and relying on the law of large numbers to even out the variations. Since these higher-level metrics can usually only be measured in a coarse-grained manner (e.g. hundreds of transactions at several million instructions per transaction for typical database applications), the end-to-end simulation time for each data point becomes quite large.

Alameldeen and Wood suggest that statistical analysis of multiple runs with random perturbations can be used to regain statistical confidence in the measured results, but this requires one or more orders of magnitude of additional simulation time to generate a single relatively comparable data point [3]. This dramatically complicates engineering trade-off analysis, since minute machine model changes can result in performance variations of only a few percent, yet the spatial variation can often result in 5-10% or even more variation in measured performance. Using statistical confidence can require a very large number of randomized simulation measurements to bound the error in such cases (the case study in Section 4.2 would require 8100 simulation runs per data point).

In this paper, we describe an alternative and complementary approach to simulation of inherently nondeterministic systems. Instead of relying on statistical methods to bound error, we systematically remove the sources of nondeterminism in a controlled and defensible fashion, quantify the performance effect of this removal, and count on the measured performance, once again expressed in IPC, as a reliable measure of relative performance. In Section 2, we provide a general overview of multithreaded simulation methodology, in order to place deterministic simulation in the proper context. In Section 3 we describe our deterministic simulation infrastructure, including details on the composition of the determinism trace used to control different simulators. We also describe several obstacles to deterministic simulation which were encountered in this work, and their solutions. In Section 4, we evaluate the usefulness of deterministic simulation and show that it is an effective means of achieving comparability among different simulation models without sacrificing fidelity with respect to the modeled system. Section 5 concludes the paper and suggests several directions for future work.

2 Simulation methodologies for multithreaded programs

In this section we discuss the advantages and disadvantages of using deterministic execution-driven simulation relative to traditional execution-driven simulation and trace-driven simulation. We begin with a review of the strengths and weaknesses of traditional trace-driven and execution-driven multiprocessor simulation, in order to properly compare the strengths and weaknesses of our method.

2.1 Trace-driven simulation

The constraints placed on a multithreaded execution by a fixed trace can have an effect on the outcome of the comparison of two simulations, and the magnitude of this effect is dependent upon the type of optimization being evaluated. When evaluating an optimization which changes the timing of a program's inter-thread dependences, the optimization's impact may be over or under-estimated because the optimized execution is constrained by a fixed trace. For example, suppose a designer is evaluating an optimization which decreases the latency of inter-processor cache block transfers in a coherence protocol for those blocks containing lock variables. In an actual system, such an optimization may decrease the number of spin iterations for processors waiting to acquire the lock. If one were to evaluate this optimization using trace-driven simulation, the spinning processors would continue to execute the same number of spin instructions, even though the lock release should have been observed earlier, and these additional spin instructions would offset any gains obtained through the optimization. Using a fixed trace artificially forces a timing simulator to follow that trace, whereas in a real system the lock transfer optimization would have caused fewer spin loop iterations in the application's execution.

Despite this drawback, trace-driven simulation offers advantages over execution-driven simulation. Because the simulator does not need to contain logic for executing the semantics of an architecture, trace-driven simulators are inherently more efficient and require less development effort. This is true of trace-driven simulators modeling both uniprocessor and multiprocessor systems. The other major advantage to trace-driven simulation is that because the simulator's execution path is dictated by the trace, one is guaranteed an identical execution across simulations of multiple machine models, despite potential sources of non-determinism present in most multi-threaded software. Consequently, the non-determinism problem does not exist for trace-driven simulators, and one can compare one machine model to another machine model using a single comparison of the results of running the trace on each machine model, rather than using many runs and gaining confidence through statistical methods [3].

2.2 Traditional execution-driven simulation

Mauer et. al. describe a useful taxonomy of execution-driven simulators [17]. In this section, we summarize their taxonomy, which we will expand upon in Section 2.3 to include deterministic execution-driven simulation. Figure 2 illustrates four simulator organizations, each different from one another in terms of the coupling of functional component and timing component. The functional component of a simulator consists of the logic necessary to implement the semantics of the computer's architecture, ranging from simple user-level instruction set architecture simulators to complex full-system simulators which implement the complete architecture including functional models of various I/O devices. The timing component is responsible for implementing a cycle-accurate simulator of a certain system.

For the sake of simulator flexibility and maintainability, it is useful to isolate each component from one another. Should a designer wish to evaluate an new experimental feature, he or she can modify the timing simulator without the risk of accidentally introducing an error into the functional model. However, if the

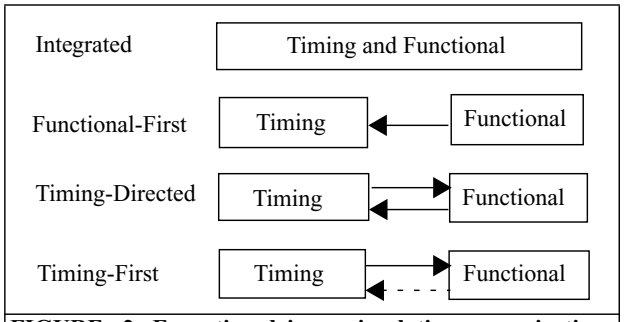

FIGURE 2. Execution-driven simulation organizations (from Mauer et. al. [17]).

separation between the two components is not carefully chosen, their interaction can also fundamentally affect the fidelity of the timing simulator with respect to the actual machine it simulates. Each arrow in Figure 2 represents per-instruction communication between the two simulator components. In the functional-first simulation model, a functional simulator feeds a fixed instruction stream to the timing simulator, which simulates the timing associated with this particular stream. Assuming a simple functional simulator which does not augment the instruction trace with wrong-path speculatively executed instructions, this organization results in a timing simulator which is unable to speculatively execute wrong-path instructions. This organization also prevents multiprocessor timing simulators from resolving timing-dependent inter-thread races (data and synchronization) and interrupts differently for different timing models. Due to this strict adherence to a fixed execution path, the timing simulator may only approximate the timing of the actual system.

To alleviate these drawbacks, some simulators allow the timing simulator to direct the execution path followed by the functional simulator. Such an organization is called a timing-directed simulator [11]. This organization allows the timing simulator to react to timing-dependent inter-processor events or speculatively execute wrong-path instructions by redirecting the functional simulator appropriately. However, support for this organization requires a more complex timing simulator because it must include some functional support to choose wrong-paths and detect inter-processor races, and a more complex functional simulator because it must be able to checkpoint state in order to perform wrong path execution.

Another alternative is the timing-first organization [17], in which a near-functionally correct timing simulator is checked by a simple functional simulator. The timing simulator must contain enough functionality to correctly execute most software, with the functional simulator serving as a safety net in case of error. When an error is detected, the timing simulator reloads its state from the functional simulator, and restarts execution at the next PC. This organization can be advantageous because it reduces the complexity of the functional simulator and communication mechanism between the two simulators, at the expense of greater functional modeling within the timing simulator. The drawback of such an organization is that the presence of races in multi-threaded workloads may cause load instructions to return different values in the timing simulator and functional simulator, incurring an unnecessary squash/restart of the timing simulator's simulated pipeline. Depending on the frequency of errors in the timing simulator's functional model or frequency of races causing different values to be returned between the two models, the the timing simulator's fidelity may be compromised.

The integrated simulator combines the functional model and timing model into a monolithic entity. Like the timing-directed simulator, it can faithfully model all aspects of a real system, including the timing dependent execution path behavior of inter-processor races and interrupts. The main drawback of integrated simulators is the additional complexity from combining the two components, which often results in a lack of simulator flexibility.

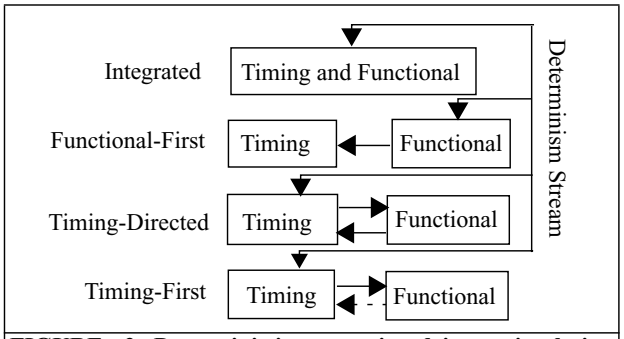

FIGURE 3. Deterministic execution-driven simulation organizations.

Each of the simulation methodologies discussed thus far suffers from the determinism problem (with the exception of trace-driven/functional-first), which prevents the direct comparison of a single execution from two different timing models. Although the trace-driven/functional-first methodology does not have a determinism problem, the strict adherence to a fixed trace can skew results, without any indication of the level of skew. In the next subsection, we will discuss how deterministic multiprocessor simulation solves these problems using the determinism-delay metric.

2.3 Deterministic MP simulation

In deterministic multiprocessor simulation, a trace of "determinism events" is fed to the simulator. The simulator uses this trace to determine when it is "safe" to perform operations, to ensure that the path followed by this execution will match the path of any other execution which uses the same determinism-trace. Using this deterministic simulation approach, we can directly compare results from different timing models because exactly the same work was performed in each.

It is possible to construct a deterministic simulator from any of the simulator types discussed above. Figure 3 illustrates the augmentation of each type with the determinism trace, which is an input to the simulator component which controls the path of execution. Details on the construction of the determinism trace and how it is used to control the simulator are found in Section 3 and Section 4.

Figure 4 presents a qualitative comparison of the fundamental differences among each simulation methodology, in terms of the fidelity of the simulator with respect to the modeled system, and ability to yield comparable results when simulating different machine models using the methodology. Of course, the fidelity of the simulator with respect to the modeled system depends on the level of detail used when implementing the timing simulator. For the comparison in Figure 4, we assume that each type of timing simulator perfectly models all aspects of the system, and thus the losses in fidelity are due to the inherent nature of the simulation methodology, not due to the lack of a detailed timing model.

Because the integrated, timing-directed, and timing-first simulators suffer from the non-determinism problem, it is not possible to yield directly comparable results without running simulations for a prohibitively long time. Consequently, these methodologies fall on the lesser side of the comparability spectrum. In terms of

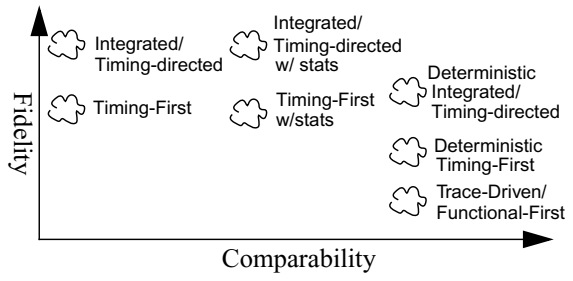

FIGURE 4. The fidelity/comparability trade-off in multiprocessor simulation.

the fidelity of these methodologies with respect to one another, timing-first simulators are inherently less faithful to the real system than timing-directed and integrated simulators, due to the errors between the timing simulator and the functional safety-net caused by different race resolutions.

By augmenting each simulator with support for deterministic replay, one can collect timing results which are comparable because determinism guarantees that the same work is being performed in each execution. This level of comparability is also possible in trace-driven/functional first simulation, at the expense of fidelity because the trace-driven timing simulator is restricted to a more rigid trace. Using deterministic simulation, we sacrifice some fidelity compared to traditional execution-driven simulation, because we introduce delays to enforce determinism, in order to increase comparability. We explain our metric for quantifying the loss in fidelity (determinism-delay) in Section 4.1.

Statistical methods are a complementary approach to deterministic simulation. One can perform a set of simulations for each machine configuration, and using statistics estimate the level of confidence associated with the outcome of a comparison between two simulated machine models. Should this level of confidence be too low, confidence can be increased by performing additional runs (and using additional CPU bandwidth), thus increasing the sample size. Unfortunately, when the simulated machine configurations being compared to one another are very close in terms of performance, one must have a large sample size in order to gain statistical confidence that one is better than the other. Using deterministic simulation, one can make this comparison using a sample size of one. As the performance of the simulated machines diverge, the sample size needed to gain statistical confidence shrinks, and the determinism-delay metric grows, indicating that deterministic simulation is not suitable for such disparate machine configurations (as we will show in Section 4). Consequently, deterministic simulation and traditional simulation with statistical methods are very complementary in terms of the trade-off between simulation time and comparability.

3 Implementing a deterministic multiprocessor simulator

In this section we discuss fundamental issues which must be considered to construct a deterministic multiprocessor simulator. We then discuss more subtle complications, some of which are specific to our target architecture (PowerPC) and also general complications with synchronization primitives and how such primitives affect performance modeling in a deterministic environment.

3.1 Handling sources of non-determinism

To implement a deterministic multiprocessor simulator, we must provide mechanisms to deterministically resolve both races and interrupts. In this section we describe the mechanisms implemented in our execution-driven, full-system simulator PHARMsim [6].

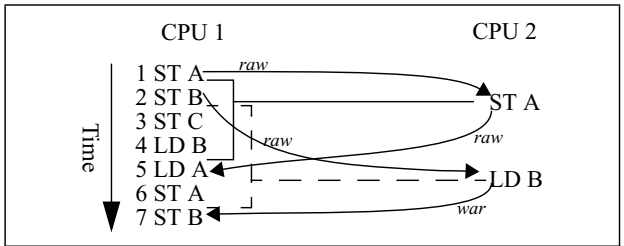

FIGURE 5. Race resolution for deterministic execution. The figure illustrates the logical time at which references from CPU 2 must appear to occur in order to obtain consistent race resolution between executions..

3.1.1 Data and synchronization races

In a deterministic multiprocessor simulator, we must consider data and synchronization races. We use the general term *race* to refer to either type. A race occurs when two or more processors perform unsynchronized conflicting accesses to a memory location. Two memory operations are said to be conflicting if they are executed by different processors, they both reference the same memory location, and at least one is a write, resulting in the occurrence of a RAW, WAW, or WAR dependence.

We illustrate the problem for all three types of dependences in Figure 5. The figure also shows the window of opportunity for the ST A and LD B by CPU 2 to be performed with respect to memory references by CPU 1. For example, CPU 2's ST A must be performed after ST A at time 1 to maintain write-after-write order, and also before LD A at time 5 to maintain read-after-write order. A similar window exists for CPU 2's LD B.

In order to handle such races, we must express the relative ordering of memory references to shared data in both a meaningful and concise way. The description should be "meaningful" in that it allows for re-creation of the same committed instruction stream across all processors; "concise" in that we effectively limit the amount of both processing and storage required. We discuss the method used in PHARMsim in Section 3.2.

3.1.2 External interrupts

External interrupts can come from many sources: DMA transfers (i.e. disk requests, network interfaces, and other I/O devices), system timers, other processors, etc., which must be modeled in full-system simulators. Our approach for ensuring the deterministic delivery of interrupts is to align them in a logical timebase within the simulation environment. Because we are interested in creating a deterministic instruction commit stream among all cpus, the logical timebase we use is committed instruction count, as used in other work (many references included in [4], [23]). For example, instead of signaling timer interrupts every N cycles, our deterministic simulator signals them every n committed instructions. Similarly, I/O interrupts are forced to occur m instructions after a request, rather than M cycles. We have observed experimentally in our simulation environment no meaningful performance difference between configurations with interrupts aligned and unaligned.

Our approach works well for many interrupt scenarios, but unfortunately not all; as an example, consider a multiprocessor system configured with all external interrupts routed to a single processor in the system, or a certain number of fixed processors [20]. This can be desirable to localize interrupt handling on a sin-

Table 1: Interrupt alignment difficulties arising in multiprocessor systems. The table shows an I/O (disk) request initiated by CPU 2, which is handled upon completion by CPU 1.

Cycle	CPU 1	CPU 2	I/O (Disk)
100		I/O read (blocking)	
200		kernel starts I/O	
300			request initiated
400			request completes
500	disk interrupt		
600	kernel finishes I/O		
700		I/O read (completes)	

gle processor, both improving locality of interrupt handling code and minimizing perturbation of other processors for the relatively infrequent event of external interrupts. An example using disk I/O requests is illustrated in Table 1.

CPU 2 initiates an I/O request at cycle 100. The disk controller reads the data from disk and transfers it into memory using DMA, which completes at cycle 400. The disk controller then raises an external interrupt which is observed by CPU 1 at cycle 500, vectoring it into the interrupt handler so it can complete the kernel's tracking of the I/O request. This processing leads to CPU 2's completion of the I/O request at cycle 700.

The key point of the example is that a request performed by CPU 2 (initiating the I/O read) impacts the instruction stream on CPU 1 (completing the I/O read). Therefore, even if we align the timing of the disk request completion with CPU 2 in logical time, this will not guarantee alignment in CPU 1's logical time. Therefore, I/O requests must be treated as synchronization events, and handled accordingly, to maintain deterministic execution.

3.2 PHARMsim deterministic execution traces

The approach taken in PHARMsim to re-create multithreaded executions is to record a *trace* of the observed execution. This trace contains both interrupt information (the interrupt vector PC, processor servicing the interrupt, and instruction boundary at which it was observed) and race information (relative logical order of memory references to shared locations). After the execution is properly recorded, it is replicated on subsequent executions of the same workload (discussed in Section 3.3).

Throughout the following discussion, keep in mind that we use the trace to express inter-processor dependencies in the logical time-base of committed instructions on each processor. All aspects of deterministic simulation which can be handled on a per-processor basis (by converting to the local logical time-base) need not be part of the trace. We discuss and justify the specific format used throughout the following sections. We use the term *identity* liberally throughout the following discussion to describe the coordinates of an instruction in logical time; this is uniquely determined by CPU number and instruction from the start of simulation, e.g. (Instruction x, CPU y). No single (system-wide) logical timebase is used for recording trace events.

3.2.1 Interrupt handling

Interrupts which can be converted into the local logical time-base (in PowerPC, the decrementer/timebase interrupts which control context-switching) cause no synchronization issues between processors and hence need not be part of the trace. Inter-processor interrupts (as illustrated by example in Table 1) constitute synchronization between processors and must have their relative order recorded in the trace. To record the order, we place an I (Interrupt) record into the trace, as shown in Table 2.

3.2.2 Race resolution

To record races, we track accesses to shared memory locations by all processors. Accesses are tracked at cache line granularity to reduce trace size (exploiting spatial locality) and also for proper handling of store-conditional operations (discussed in Section 3.4.4). Furthermore, local references (references which do not contribute to a dynamic instance of sharing between processors) are combined to further reduce trace size.

To handle shared loads, we add a trace record with the identity of the first reader and the last writer, indicated with the L (Load) record in Table 2.

Handling shared stores requires maintaining additional identities in the trace (compared to shared loads) because stores can create both WAW and WAR dependences (as shown in Figure 5). The WAW dependence is handled similarly to RAW; we simply track the identity of the current writer and the previous writer. The WAR dependence further requires we track the identities of all previous readers. Therefore, shared stores are recorded indicating all such dependences, indicated with the S (Store) record in Table 2.

Finally, since PowerPC implements load-locked, store-conditional atomic primitives, we create additional records for store-conditional operations, indicated with the C (Conditional-store) record in Table 2. Records are placed into the trace for all store-conditionals, whether they succeed or fail. Successful store-conditionals additionally create shared store records, if necessary.

Table 2: Trace record types, data fields, and description. The information contained within each determinism-trace record is indicated. "LT" indicates "logical time" which is determined at architected instruction boundaries for each CPU.

Type	Dependence Information	Description
I	{CPU_ID, LT, Int. Vector PC, Int. Type}	***I***nterrupt record, CPU servicing the interrupt, the instruction boundary, vector PC, and interrupt type
L	{Physical Address, Load CPU_ID, Load LT, Previous Store CPU_ID, Previous Store LT}	***L***oad record, physical memory address referenced, load identity, and previous store identity
S	{Physical Address, Store CPU_ID, Store LT, Previous Store CPU_ID, Previous Store LT, Previous Load LT[Numprocs]}	***S***tore record, physical memory address referenced, current store identity, previous store identity, and previous load identities for each processor in the system (Numprocs)
C	{Physical Address, Store CPU_ID, Store LT, Success/Failure}	***C***onditional store record, physical memory address referenced, and success/failure status of the conditional store

Table 3: Observed execution and corresponding trace format. An execution involving three processors is shown with trace entries for shared references. Cells with double outlines indicate inter-processor dependencies (WAR, RAW, RAW, and WAR/WAW from top to bottom) requiring trace storage. The logical time (instruction count) is shown for each CPU adjacent to the memory reference.

CPU 1	CPU 2	CPU 3	Corresponding Trace Entries							
1 LD A			L	A	1	1	-1	-1		
	1 ST A		S	A	2	1	-1	-1	1	1
2 LD A			L	A	1	2	2	1		
3 LD A										
4 ST A										
		1 LD A	L	A	3	1	1	4		
	2 ST A		S	A	2	2	1	4	3	1

In Table 3 we show a three-processor example and the corresponding trace that would be generated for the observed execution. Horizontal rows in the table indicate the observed global order of memory references for ease of presentation. The example shows two RAW dependences (Time 2, CPU 1; Time 1, CPU 3), two WAR dependences (Time 1, CPU 2; Time 2 CPU 2), and a single WAW dependence (Time 2, CPU 2). Note that the combined WAR/WAW dependence (Time 2, CPU 2) is represented with a single shared store record. Also note that only references creating or observing shared values lead to storage in the trace; therefore the LD A at (Time 3, CPU 1) and ST A at (Time 4, CPU 1) do not create trace records.[1] The LD A at (Time 1, CPU 1) record reflects a cold miss, which is also recorded for both loads and stores.

It is obvious from this example that our deterministic execution traces bear little similarity to traditional hardware traces used for architectural evaluation as described in Section 2. Our traces only indicate relative orderings to allow race resolution and basic information for interrupt handling--no memory value or instruction sequencing information is maintained. Note that our traces serve the same purpose as many other proposals related to both multithreaded programming debugging and deterministic replay (e.g. [4], [23]). However, our traces are designed for maximal concurrency in playback, and not to minimize trace size, in contrast to these approaches. Furthermore, to our knowledge, we are the first to propose and evaluate such deterministic playback in the context of multithreaded program performance evaluation.

3.3 Deterministically resolving races by inserting delays

When running PHARMsim in deterministic mode, we load a trace of the execution to be re-created, as described previously. Once the trace is loaded, PHARMsim's execution semantics must be changed to ensure the desired execution is re-created. We ensure this by delaying references until correct memory order or interrupt alignment conditions are satisfied. For races, using the example from Table 3: we must ensure that ST A (Time 1, CPU 2) is not performed until LD A (Time 1, CPU 1) is performed; LD A (Time 2, CPU 1) is not performed until ST A (Time 1, CPU 2) is performed, etc. Note that "performed" in this context implies that a store is visible to all processors in the system (and has been reflected throughout their respective cache hierarchies) or a load has bound its value and is non-speculative.

In the case of interrupts, the processor queries the trace to see if an interrupt is to occur at the current instruction; if so, two possibilities arise: the interrupt is already pending or it has not yet been signaled. If the interrupt is already pending, we service it by vectoring to the interrupt handler as indicated in the trace. If the interrupt has not yet been signaled, we stall the processor at the current instruction to wait for the interrupt. For the example presented in Table 1, the inter-processor interrupt can now be correctly handled by CPU 1 at cycle 500 because it will either wait for the interrupt (if it reaches the interrupt instruction boundary too early) or the interrupt will be deferred until the CPU reaches the correct instruction boundary.

Of course, delaying execution of instructions artificially to maintain deterministic execution affects PHARMsim's fidelity. We discuss our metric for fidelity, *determinism-delay* and the impact on fidelity in detail in Section 4.

3.4 Implementation considerations

We have previously discussed the difficulties of interrupt alignment and race resolution which are common across virtually all modern architectures. In this section, we discuss more subtle implementation issues which may not apply across all architectures, focusing on PowerPC (on which PHARMsim is based) while also commenting on other common ISAs. We then give two examples of potential performance optimizations that present complications within a deterministic simulation environment.

3.4.1 Speculative references

When simulating a modern microprocessor that performs speculative execution, we must correctly deal with speculative references. PHARMsim is a completely integrated timing and functional microprocessor model, in which we do not know a priori whether a given instruction is from the wrong path when it is executed. Because the determinism trace contains only a subset of the committed instruction stream, it cannot be used to distinguish incorrect speculative references from correct ones. Therefore, we force all references to follow the semantics specified by the trace--incorrect speculative references may either be delayed (if a record governing their execution exists in the trace) or allowed to issue with no restrictions. This creates no correctness problem since incorrect references will be squashed anyway.

3.4.2 Address translation/page table references

Memory references must undergo translation from virtual to physical address to be correctly serviced by the memory system. In order for the translation to be performed when a TLB miss occurs, the page table must be consulted. Fundamentally, this page table walk has an implicit dependence on the data stored within the page table used to correctly translate the memory address.[2]

1. However, the ST A at (Time 4, CPU 1) is reflected in the trace when it is observed at (Time 1, CPU 3) and (Time 2, CPU 2).

Hardware TLBs:

PowerPC specifies an autonomous, hardware-based, page table walker to service TLB misses and TLB misses are not architecturally visible. Therefore, updates to the page table performed by the kernel (and potentially TLB shoot-down) should be tracked to maintain deterministic execution. Since the TLB fill mechanism is completely autonomous, all types of dependences through the page table (WAW[1], RAW, and WAR) must be considered.

Since our traces only reflect architected events, handling this problem in general is very difficult. The solution employed in PHARMsim is to assume that races to page table data will be serialized transitively, i.e. any dependence on page table data will be protected by other architecturally visible synchronization, and therefore will be correctly resolved. In practice, we have empirically observed only one occurrence of an unsynchronized page table update that was not properly reflected under deterministic simulation after many machine-years of simulation; therefore, we believe the current solution is viable for our purposes. Work-arounds for this problem exist, but all are relatively heavy-weight solutions. One method to handle explicit page table modifications involves placing barriers in the trace whenever such an event occurs; this would also require modification to PHARMsim (to implement actually implement the barrier at trace generation) when explicit page table modification occurs. Another solution is to add trace records for page table modification and also reflect the translation dependence in the load and store records (Table 2). Since PHARMsim can detect when it deviates from the trace, we can easily determine when implementing such methods becomes necessary.

Software TLBs:

Many ISAs implement software TLB miss handlers; in these architectures page table WAW dependences require no special handling since the writes are explicit architecturally visible instructions. RAW dependences are also correctly handled, since both the read (TLB miss, assuming appropriate TLB shoot-down) and write (page table update) are explicitly visible. However, WAR dependences must be considered to ensure a translation is not changed before its last use. Solutions similar to those described for hardware TLBs may be applied here.

3.4.3 Self-modifying code

Self-modifying code is not a problem, in general, in our PowerPC simulation environment since the architecture specifies any self-modifying code must be protected with explicit cache control which is architecturally visible. Therefore, self-modifying code is handled through transitive synchronization. However, in ISAs which do not provide such guarantees, additional instruction fetch records, essentially identical to shared load records ("L" Table 2), and appropriately reflecting fetch in shared store records ("S" Table 2) can address this issue.

3.4.4 Load-locked and store-conditional primitives

Many architectures, including PowerPC, provide load-locked (LL) and store-conditional (SC) primitives to allow efficient creation of multiple synchronization constructs. However, LL-SC primitives cause a problem in deterministic simulation. Similar to the page table (Section 3.4.2), such references create an implicit dependence on the reservation register, or reservation granule (rgranule). Therefore, to maintain deterministic execution, we must ensure that this implicit dependence is resolved correctly at trace playback. We can do this by assuring SCs can only fail for one of two reasons within PHARMsim during trace creation: The SC is not paired with a LL (i.e. the reservation is not set), or another store to the rgranule from a remote processor is observed between the LL-SC pair. Practically, this means a reserved cache line can never be replaced due to capacity/conflict misses or other implementation artifacts.

For this reason, we make traces at cache line (rgranule) granularity, as indicated in Section 3.2. Since stores to locations within the same cache line (but not to the rgranule, i.e. *false sharing* [10]) can cause a SC on another processor to fail, and therefore can be observed architecturally, the trace serializes both *true sharing* and *false sharing* references. Work-arounds to mitigate false sharing are possible, but not detailed in this work.

3.4.5 Performance techniques: exclusive prefetching and silent stores

A common optimization in microprocessors is to speculatively prefetch exclusive permission for stores to improve the latency and throughput of store retirement. Also, academic researchers have proposed exploiting stores which to not change memory state, so-called *silent stores*, to improve multiprocessor system performance [16]. Both optimizations complicate deterministic simulation in PHARMsim because of LL-SC pairs (Section 3.4.4). Non-architected exclusive prefetches present in the execution generating the trace may have the side-effect of causing SCs to fail; if the exclusive prefetch is not present upon trace playback due to different speculative execution paths, an execution mismatch will result (since the SC will succeed in the recreated execution). Furthermore, exclusive prefetches not present in the execution generating the trace but occurring during trace playback may cause SCs to fail upon playback which succeeded during trace generation. A similar scenario exists for silent stores.

There are many potential solutions to the problems caused by the potential side-effects of memory references. The solutions which we have devised fall into three categories: to ensure such events never can be architecturally visible (by design), to ensure that a dynamic instance of the event will not affect execution, or use information from the trace to directly govern execution. A host of options are available to handle such problems. As a simple illustration of each type of method, consider the following:

- We can ensure an exclusive prefetch will never be architecturally visible by only prefetching ownership before the coherence point in the memory system (in PHARMsim, between the L1 and L2 caches). This may still provide performance benefit as we prefetch for L1 misses which hit the L2 in exclusive or modified state.
- We can ensure a dynamic exclusive prefetch will not affect execution by only issuing exclusive prefetches which are guaranteed to be protected by other synchronization, NACKing exclusive prefetches to reserved memory locations, or appropriately stalling succeeding LL-SC pairs when exclusive prefetches are in-flight to the target memory address.
- We can force a dynamic SC in the controlled execution to succeed or fail based solely on the trace (by simply reading the SC success or failure status from the trace and forcing the experiment to have the same behavior), ignoring the success or failure status produced within the memory system of the controlled execution (this is why we have records for every SC, i.e. "C" Table 2).

We tend to favor the first two methods (and avoid relying on the trace to govern execution) so the deterministic simulation environment does not mask functional errors within PHARMsim.

2. A similar problem occurs for the reference/change bits (R/C bits) which we neglect for the sake of brevity.

1. The WAW dependence can occur with R/C updates by the TLB fill mechanism and explicit kernel stores to clear these bits.

However, controlling execution from the trace is a valid solution, since the traces are either directly produced from, or validated by, the SimOS-PPC [15] functional simulator.[1] This guarantees the traces represent a legal execution under the PowerPC architecture (see Section 4.3).

3.5 Memory consistency considerations

PHARMsim deterministic execution traces, as described, provide a mechanism for describing coherence; the combining assumption for memory references in the trace is that program order rules within a processor apply to memory references. For example (all operations to the same memory location), if a load at (Time n, CPU x) must observe a store at (Time m, CPU y), we assume that a load at (Time n+1, CPU x) must also observe the store at (Time m, CPU y). This is a fundamental tenet of coherence (total order of writes to any memory location and program order) [8]. Therefore, the traces can be used to describe an execution from any common memory consistency model which requires coherence [1].

In the current implementation within PHARMsim, we only support trace creation from sequentially consistent systems. Similarly, we only support re-creating sequentially consistent executions when controlling execution from a trace. However, PHARMsim can exploit implementation freedoms available under other memory models (TSO and PowerPC weak ordering) when running in deterministic simulation mode. The constraints described previously only stipulate that the machine will follow a sequentially consistent execution. Extending the current infrastructure to support trace generation and controlled execution of weaker models is an interesting area of future research.

Furthermore, we note that deterministic simulation enforces a causal relationship between synchronizing events (i.e. interrupts and races). This approach allows maintaining the invariant that the architected state of each processor at each instruction boundary across deterministic simulations is identical. This is desirable for many reasons, including simulator verification. However, this causal relationship can also be overly conservative if we are only concerned with recreating the same "work" across deterministic simulations of an entire workload. As an example, consider the case of two processors racing to each increment a shared counter once; the two increments can be performed in either order by the respective processors, with the same result observed subsequently. Deterministic simulation artificially enforces whichever order was observed at trace creation upon playback.

Due to the causal relationships and execution restrictions imposed, any performance optimization which changes or exploits relaxed architectural semantics should be thoughtfully considered before using deterministic simulation (e.g. delayed consistency [9]). However, as illustrated in the following section, deterministic simulation may be used for many architectural studies.

4 Evaluation

In this section, we define a metric used to gauge the degradation of fidelity in our deterministic environment (*determinism-delay*) and also explore the suitability of this method for various architectural evaluations. The simulation parameters used for all simulators in the evaluation are given in Table 4.

Table 4: Simulated machine parameters. Functional unit latencies are shown in parenthesis.

Attribute	Value
Fetch/Xlate/ Decode/Issue/Commit	4/4/4/4/4
Pipeline Depth	6 stages
BTB/Branch Predictor/RAS	8K sets, 4-way/8K combining (bimodal and GShare)/64 entry
RUU/LSQ	128 entry/64 entry
Integer	ALUS: 4 simple (1), 1 mul/div (3/12); Memory: 2 LD/ST
Floating Point	ALUS: 4 add/sub (4/4), 2 mul/div/fmac (4/4/4)
L1-Caches	I$: 32KB, 2-way, 64B lines (1); D$: 32KB, 2-way, 64B lines (1)
L2-Cache	Unified: 2MB, 4-way, 64B lines (8)
Memory/ Cache-to-Cache	Minimum latency: 500 cycles, 50 cycles occupancy/txn, crossbar
Address Network	Minimum latency: 30 cycles, 20 cycles occupancy/txn, bus
TLB	Hardware page table walker, 1-level, 2K sets, 2-way, 4K pages

4.1 The determinism-delay metric

Architectural studies normally consist of relative performance comparisons; we have a base machine with a given performance and we want to determine the impact of a novel/modified microarchitectural feature. We call the base execution the *control* and the subsequent execution an *experiment*. For example, we choose a particular cache size (the *control*) and ask the question: "Does doubling the cache size improve performance? If so, by how much?" (the *experiment*).

To enable direct comparison of the control and experiment and avoid non-determinism effects, we propose re-creating an execution by appropriately delaying interrupt signaling or selected memory operations. Intuitively, if the amount of delay injected is small relative to the measurement interval of our workload, we can directly compare the control and experiment to determine relative performance. However, to make the comparison meaningful, we must know how much the experiment's execution was affected by the artificial delay introduced.

We can bound the error by counting the number of cycles in which any operation within a processor is stalled and dividing the number of stall cycles by the total number of cycles executed by all processors in the experiment. For example, in a 16 processor simulation, if a total of 10M stall cycles (across all processors) were introduced for a 100M cycle run to complete the workload, we say the experiment had (10M/(100M*16 processors)) * 100% = 0.63% *determinism-delay*. This method is conservative; if only a single operation within the processor is delayed, and execution of this operation does not contribute to the critical path through the workload, this will artificially inflate the determinism-delay.

We can use the formulas from Figure 6 to determine the rela-

1. The details of validating traces from PHARMsim with SimOS-PPC are non-trivial and beyond the scope of this work. However, the validation assures the architected state observed by every committed instruction in both simulations is identical without passing any execution semantic information between simulators. Therefore, a trace is only validated if its execution can be recreated with a semantically unmodified SimOS-PPC.

(1)

if ($IPC_{Experiment}$ > $IPC_{Control}$)
=> *Better*
else if ([$IPC_{Experiment}$ / (1 - Determinism-Delay)] > $IPC_{Control}$)
=> *Inconclusive*
else
=> *Worse*

(2)

[$IPC_{Experiment}$ / $IPC_{Control}$,
($IPC_{Experiment}$/(1 - Determinism-Delay)) / IPC_{Con-

FIGURE 6. Determining relative performance in the presence of determinism-delay. The figure indicates (1) how to determine if an experiment is better or worse than the control and (2) how to bound relative performance..

tive performance of the experiment. If the experiment provides greater instruction throughput even with inserted delay, it is better; if the experiment provides lower instruction throughput, but the decrease in performance is less than the injected delay, the result is inconclusive; otherwise the decrease in performance is greater than the injected delay and the experiment performs worse. Since determinism-delay measures the fidelity sacrificed to maintain deterministic simulation (a worst-case bound on absolute simulation error), when graphing results, we use error bars to indicate determinism-delay. In similar fashion, we can bound the relative performance benefit as within the interval shown in (2). Note that this formulation is equivalent to (1) if we subtract unity from both sides of the bound; strictly positive indicates performance improvement, alternating signs indicates an inconclusive result, and strictly negative indicates performance degradation.

4.1.1 Intrinsic and artifactual determinism-delay

The determinism control implemented in PHARMsim has overhead in tracking visibility and binding of memory values (Section 3.1). This overhead manifests itself as delay injected even when the control and experiment configurations are identical. This delay is an artifact of the mechanics used to control determinism, thus we call it *artifactual determinism-delay*. In contrast, when delay is injected due to different intrinsic workload executions observed between the control and experiment (because of machine model differences), we call such delay *intrinsic determinism-delay*.

We determine the artifactual determinism-delay by running an experiment which exactly matches the control. We call this simulation the Artifactual Simulation. To improve the utility of the formulas proposed previously, we may then imagine using $IPC_{Artifactual}$ in place of $IPC_{Control}$ for relative performance comparisons. Strictly speaking, determinism-delay is the only valid metric for measuring the absolute amount that we have compromised the fidelity of simulation to maintain deterministic execution, and thus it can be considered the precision of the deterministic approach. However, we will show empirically throughout the next sections that the precision of the deterministic approach is much better than the conservative determinism-delay metric indicates. We also note that additional tuning of our deterministic simulation control mechanism can decrease the artifactual determinism-delay to near zero; therefore the true limit to precision of the methodology itself is intrinsic determinism-delay. We report both metrics throughout our results.

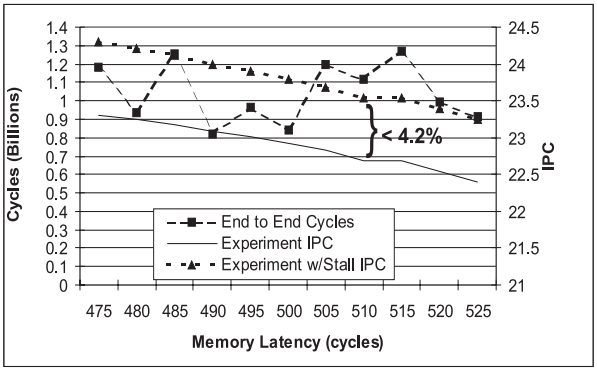

FIGURE 7. Comparison of non-deterministic vs. deterministic simulation. The performance of SPECjbb2000 (16 processor) is shown with varying main-memory latency. The "End to End" result indicates the difference in simulated cycles observed, illustrating significant potential for measurement error. The Experiment results show the change in IPC simulating deterministically, including/excluding IPC contribution due to determinism-stall.

4.2 Restoring intuition of simulation results via deterministic simulation

In Figure 1, we varied main-memory latency and showed the measured difference in cycles to complete our *SPECjbb-16p* workload. These results indicate that significant non-determinism exists in multithreaded commercial workloads, even with long simulation intervals (approximately 30 billion instructions per run), as described in [3]. With PHARMsim, each data point took more than 300 simulation hours to collect and the result is definitely counter-intuitive; from these runs, we would conclude that a memory latency of 525 cycles is better than 475 cycles.

In Figure 7, we overlay the results shown previously with the IPC results from our deterministic simulation approach (using a main-memory latency of 500 cycles as the control). The figure shows both $IPC_{Experiment}$ and IPC considering determinism-stall. We can see the intuitive behavior of monotonically decreasing performance for increasing memory latency for both IPC curves. Note that no statistical method is used to generate this result--each data point is collected from a single simulation run. Furthermore, deterministic simulation allows reliable measurement of even minute machine changes; each data point corresponds to roughly a 1% difference in main-memory latency, which translates to approximately 0.4% IPC per data point. Rigorously validating this observation requires statistical methods since the determinism-stall is approximately 4%. To obtain this level of resolution statistically (95% confidence interval, coefficient of variation=18%, relative error=0.4%) we would need 8100 runs to prove each reduction of 5 cycles was advantageous. This translates into 8100*11*300=27M simulation hours (3000 simulation years). Obviously such validation is not tractable. Therefore, we let the intuitive result justify that the precision of the deterministic simulation approach is greater than the conservative determinism-stall metric indicates.

4.3 Exploration of different control simulators

In the previous section we showed that deterministic simulation can reliably measure minute changes in machine performance for relative performance characterization. How well does the deterministic simulation approach work for large machine changes? What sacrifice in fidelity do we observe with the deter-

Table 5: Benchmark description and characteristics. IPC (across all processors) for the PRAM, Inorder, $OoO_{Control}$ and $OoO_{Artifactual}$ simulations as well as Artifactual Determinism-Delay are shown.

Workload	Description	IPC PRAM	IPC InOrder	$IPC_{OoO, Control}$	$IPC_{OoO, Artifactual}$	Artifactual Determinism-Delay
barnes-4p	SPLASH-2 N-body simulation (8K) [22]	4.0	3.58	5.77	5.77	0.0%
ocean-4p	SPLASH-2 Ocean simulation (258x258)	4.0	1.45	3.34	3.29	1.5%
radiosity-4p	SPLASH-2 Light Interaction application (-room -ae 5000.0 -en 0.050 -bf 0.10)	4.0	3.6	5.95	5.94	0.2%
raytrace-4p	SPLASH-2 Raytracing application (car)	4.0	2.80	5.46	5.45	0.2%
SPECjbb-4p	Commercial Server-Side Java [19]	4.0	1.70	1.94	1.87	3.6%
tpc-h-4p	Commercial Decision Support [21]	4.0	1.34	1.80	1.54	14.4%
tpc-w-shopping-4p	Commercial 3-Tier Web-Based OLTP application (shopping mix) [7]	5.0	2.10	7.65	7.52	1.7%
barnes-16p	SPLASH-2 N-body simulation (8K)	16.0	13.6	36.2	35.3	2.4%
ocean-16p	SPLASH-2 Ocean simulation (514x514)	16.0	7.05	27.9	27.4	1.4%
radiosity-16p	SPLASH-2 Light Interaction application (-room -ae 5000.0 -en 0.050 -bf 0.10)	16.0	13.7	24.9	24.0	3.7%
raytrace-16p	SPLASH-2 Raytracing application (car)	16.0	11.8	21.0	19.9	5.4%
SPECjbb-16p	Commercial Server-Side Java [19]	16.0	13.5	22.3	21.5	3.4%
tpc-h-16p	Commercial Decision Support [21]	16.0	8.03	29.3	27.0	7.8%

ministic approach? We explore these questions by making traces from three different control simulators: An in-order model which runs at 1 IPC for all instructions including memory (PRAM); An in-order model with perfect core IPC of 1 and the memory latencies shown in Table 4 (InOrder); and the PHARMsim out-of-order model as described in Table 4 (OoO). Note that these machines correspond to a pure functional multiprocessor simulator, a functional simulator simply augmented with memory system timing, and a fully-integrated out-of-order simulator, respectively. The benchmarks are described and IPCs for control and artifactual simulations are given in Table 5.

We can estimate the performance difference between these machines and our faithful out-of-order model by loading the trace collected from each simulator (as the control) and duplicating the execution in PHARMsim. The determinism-delay metric provides a bound on how much fidelity has been sacrificed between the control environment (i.e. the one creating the trace) and the experiment environment. Put another way, determinism-delay measures how "difficult" it is for PHARMsim (our target simulator) to re-create the same execution. Results of this study are shown in Figure 8.

On average, 19.6%, 16.2%, and 3.5% determinism-delay is observed for each control simulator, respectively, indicating that executions in our OoO model most closely mirror themselves, then the InOrder model, and finally the PRAM model. This result shows that an out-of-order model with cache hierarchy more is more closely approximated by an in-order model with cache hierarchy than PRAM, as one would expect. Note that even though the average delay for the InOrder model is large, it is modest in many cases (all 4p benchmarks except *tpc-h-4p*[1]). This result is encouraging from an engineering perspective. Because determinism eliminates races, a simulator designed solely for deterministic simulation can simplify coherence protocol race handling and rely on conservative deterministic simulation control to resolve them and still maintain correct execution. Our results indicate that one may be able to rely on a functional with memory-system timing simulator to generate traces and build an out-of-order model with this substantial simplifying assumption. This may be

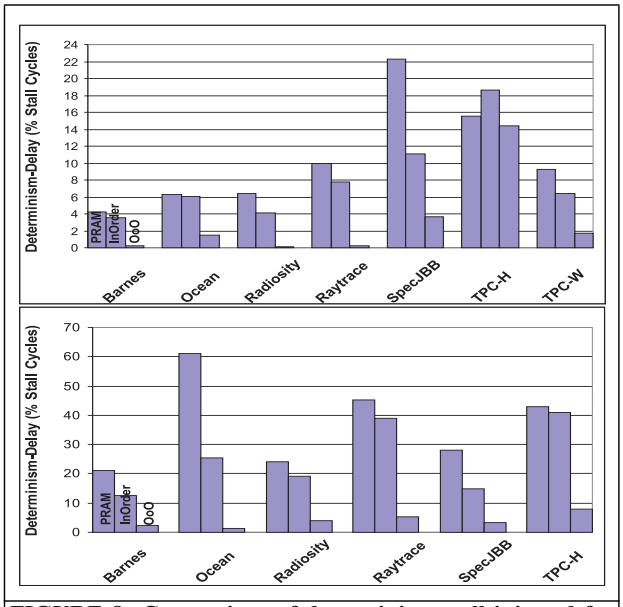

FIGURE 8. Comparison of determinism stall injected for different control simulators. We indicate the IPC and stall for PHARMsim running traces created from three different simulators (PRAM, InOrder, and itself). IPC is normalized to the control simulator (see Table 5). 4p workloads are shown above, 16p below..

1. In this case, delay from the OoO model (i.e. artifactual delay) is also large, implying that most of the delay can be removed by improving determinism control within PHARMsim, and is not in fact due to intrinsic machine differences.

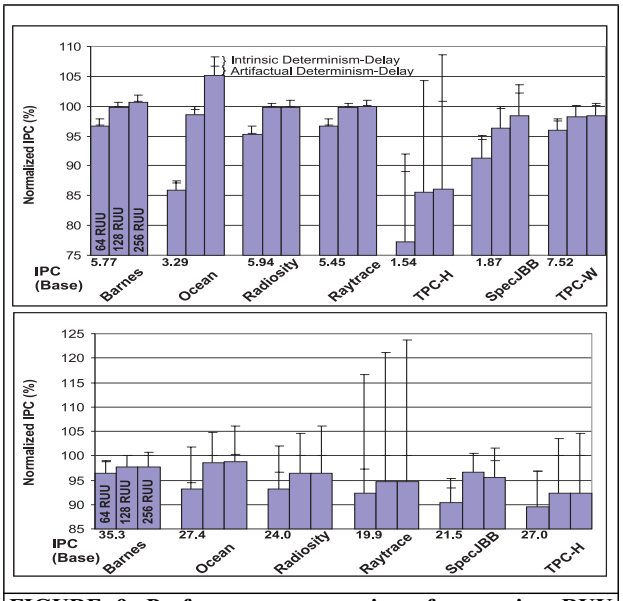

FIGURE 9. Performance comparison for varying RUU sizes. We indicate the relative IPC for each machine model relative to the control of 128-entry RUU. Baseline IPC, normalized IPC, artifactual determinism-delay, and intrinsic determinism-delay are shown for each benchmark. 4p workloads are shown above, 16p below.

beneficial, since races and concurrency within a coherence protocol are widely accepted to be the most difficult cases to handle correctly. Furthermore, the fidelity sacrificed with such an environment can always be bounded by the determinism-delay metric. Although we do not explore it in this work, we expect that hardware traces (commonly used in industry) might also be used to drive such a simplified, deterministic, simulator, simplifying the design of high-fidelity performance models.

However, if a validated out-of-order simulator exists, such traces are most suitable, as the least determinism-delay (3.5%) needs to be injected in this case. This delay is exactly the artifactual determinism-delay (as explained in Section 4.1.1 and shown in Table 5). This low determinism-delay indicates that recreating an execution by delaying operations introduces little error. However, we point out that the artifactual determinism-delay can be large (*tpc-h-4p* is 18%, *raytrace-16p* is 22%), indicating additional tuning of our control mechanisms is worthwhile. We are continuing to tune our infrastructure to improve this result. Note that reducing artifactual stall to near zero is an engineering problem within the simulator and is not a fundamental shortcoming of our approach.

4.4 Suitability for architectural evaluation

To illustrate the suitability of deterministic simulation for relative performance comparison, we perform a simple study with our OoO model--varying the RUU size from 64 entries to 256 entries and measuring the delivered performance (128 entries is the control). Intuitively, we expect monotonic increase in performance for increasing RUU sizes; this result was shown to be statistically valid for a similar simulation environment in [3]. The results of this experiment are shown in Figure 9.

Focusing first on the normalized IPC results, we see the expected trend, i.e. increasing RUU size is an effective means of increasing performance. As described in Section 4.1, to strictly determine relative performance we must rely on the formulas in Figure 6. Graphically, these formulas correspond to comparing the reported IPC against the determinism-delay measurements. As a specific example, consider *ocean-4p*: the graph indicates 64-entry RUU IPC as 87% of the baseline including determinism-delay. 128-entry RUU IPC (without determinism-delay) is 98% of the baseline. Therefore, we can conclude that a 64-entry RUU is at least 11% worse than a 128-entry RUU for this benchmark. The upper bound is the converse (IPC without determinism-delay for 64-entry RUU, with determinism-delay for 128-entry RUU); 86% and 100% (respectively), yielding 14%. Consider further *tpc-h-4p*: here we can make no strict comparison because determinism-stall for a 64-entry RUU exceeds reported IPC for a 128-entry RUU. The rest of the results can be interpreted similarly.

However, as we pointed out in Section 4.2, the precision of deterministic simulation may be higher than the conservative determinism-stall metric indicates. In that section we showed that even with 4% determinism-delay, we could reliably measure changes in performance of 0.4%. These results show a similar conclusion; in all cases the IPC, artifactual stall, and intrinsic stall show the expected trend. The only exception in *specjbb-16p*, which shows a performance decrease for a 256-entry RUU vs. a 128-entry RUU. We have examined the data further, and we observe additional cache misses in the 256-entry RUU case, implying that the slowdown reported may indeed be plausible--possibly caused by additional cache misses due to wrong-path memory references. Overall, these results indicate substantial precision (beyond the conservative determinism-delay bound) for deterministic simulation.

5 Summary and conclusion

The goals of this paper were to:

- Describe a new simulation methodology which increases our confidence in the outcomes of multiprocessor simulation experiments by removing variances intrinsic to non-deterministic workloads.
- Provide a precise mechanism which can identify those experiments for which this methodology is suitable, and those for which it is not.

Using the determinism-delay metric, we are able to gauge the amount that a workload is perturbed by forcing its execution to be deterministic. We have shown that for many experiments this value is quite low. For such experiments, it is therefore valid to draw conclusions based on the outcome of a single run. If determinism-delay is high, one can simply fall back on performing multiple runs and gaining experimental confidence through statistics. Furthermore, deterministic simulation must be carefully considered when used to explore any optimization which is designed to exploit changed/relaxed architectural semantics.

Although there are caveats associated with deterministic simulation, we believe that we have achieved our goals. There remain several open opportunities to improve our deterministic simulation environment. We are actively reducing the artifactual delays induced when following a trace, and believe we will achieve substantial reduction by migrating the load/store stall point from the processor issue/commit stage toward the race resolution point (the address network). There are also opportunities for reducing intrinsic stall through the identification and removal of spin-loop iterations which can artificially inflate the instruction count of the control execution but essentially perform no useful work. This inflated instruction count in the control execution can cause intrinsic stall because of the overhead associated with executing the precise number of spin iterations present in the control.

Acknowledgements

This work was supported in part by the National Science Foundation with grants CCR-0073440, CCR-0083126, EIA-0103670, and CCR-0133437, and generous equipment donations and Fellowship support from IBM and Intel. We also thank the anonymous reviewers for their many helpful comments.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[2] Alaa R. Alameldeen, Carl J. Mauer, Min Xu, Pacia J. Harper, Milo M.K. Martin, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Evaluating non-deterministic multi-threaded commercial workloads. *Workshop On Computer Architecture Evaluation using Commercial Workloads*, February 2002.

[3] Alaa R. Alameldeen and David A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the 9th Annual International Symposium on High Performance Computer Architecture*, 2003.

[4] David F. Bacon and Seth Copen Goldstein. Hardware-assisted replay of multiprocessor programs. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 26(12):194–206, 1991.

[5] J. Borkenhagen and S. Storino. 5th Generation 64-bit PowerPC-Compatible Commercial Processor Design. IBM Whitepaper available from http://www.rs6000.ibm.com, 1999.

[6] Harold W. Cain, Kevin M. Lepak, Brandon A. Schwartz, and Mikko H. Lipasti. Precise and accurate processor simulation. Proceedings of Computer Architecture Evaluation using Commercial Workloads (CAECW-02), February 2002.

[7] Harold W. Cain, Ravi Rajwar, Morris Marden, and Mikko H. Lipasti. An architectural characterization of Java TPC-W. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 229–240, Monterrey, Mexico, January 2001.

[8] David E. Culler and Jaswinder P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1999.

[9] M. Dubois, L. Barroso, J. C. Wang, and Y. S. Chen. Delayed consistency and its effects on the miss rate of parallel programs. In Proceedings of Supercomputing ’91. ACM Press, 1991.

[10] Michel Dubois, Jonas Skeppstedt, Livio Ricciulli, Krishnan Ramamurthy, and Per Stenström. The detection and elimination of useless misses in multiprocessors. In *20th Annual International Symposium on Computer Architecture*, May 1993.

[11] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *IEEE Computer*, 35(2):68–76, February 2002.

[12] D. T. Marr et. al. Hyper-Threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), 2002.

[13] P.S. Magnusson et. al. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[14] Intel Corporation. *IA-64 Application Developer’s Architecture Guide*, 1999.

[15] Tom Keller, Ann M. Maynard, Rick Simpson, and Pat Bohrer. Simos-ppc full system simulator. http://www.cs.utexas.edu/users/cart/simOS.

[16] Kevin M. Lepak and Mikko H. Lipasti. On the value locality of store instructions. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 182–191, Vancouver, B.C., Canada, June 2000.

[17] Carl J. Mauer, Mark D. Hill, and David A. Wood. Full-system timing-first simulation. In *Proceedings of the 2002 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 2003.

[18] Mendel Rosenblum. Simos full system simulator. http://simos.stanford.edu.

[19] Systems Performance Evaluation Cooperative. SPEC benchmarks. http://www.spec.org.

[20] Joel M. Tendler, J. Steve Dodson, J. S. Fields, Hung Le, and Balaram Sinharoy. Power4 system microarchitecture. http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.htm% l, November 2001.

[21] Transaction Processing Performance Council. TPC benchmarks. http://www.tpc.org.

[22] Steven C. Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder P. Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[23] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th International Symposium on Computer Architecture*, San Diego, CA, USA, 2003.