Appears in the proceedings of the
2002 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems
June 15, 2002

Full-System Timing-First Simulation

Carl J. Mauer, Mark D. Hill and David A. Wood
Computer Sciences Department
University of Wisconsin—Madison
{cmauer, markhill, david}@cs.wisc.edu
http://www.cs.wisc.edu/multifacet/

Abstract

Computer system designers often evaluate future design alternatives with detailed simulators that strive for *functional fidelity* (to execute relevant workloads) and *performance fidelity* (to rank design alternatives). Trends toward multi-threaded architectures, more complex micro-architectures, and richer workloads, make authoring detailed simulators increasingly difficult. To manage simulator complexity, this paper advocates decoupled simulator organizations that separate functional and performance concerns. Furthermore, we define an approach, called *timing-first simulation*, that uses an augmented timing simulator to execute instructions important to performance in conjunction with a functional simulator to insure correctness. This design simplifies software development, leverages existing simulators, and can model micro-architecture timing in detail.

We describe the timing-first organization and our experiences implementing TFsim, a full-system multiprocessor performance simulator. TFsim models a pipelined, out-of-order micro-architecture in detail, was developed in less than one person-year, and performs competitively with previously-published simulators. TFsim's timing simulator implements dynamically common instructions (99.99% of them), while avoiding the vast and exacting implementation efforts necessary to run unmodified commercial operating systems and workloads. Virtutech Simics, a full-system functional simulator, checks and corrects the timing simulator's execution, contributing 18-36% to the overall run-time. TFsim's mostly correct functional implementation introduces a worst-case performance error of 4.8% for our commercial workloads. Some additional simulator performance is gained by verifying functional correctness less often, at the cost of some additional performance error.

This work is supported in part by the National Science Foundation, with grants EIA-9971256, CDA-9623632, and CCR-0105721, two Wisconsin Romnes Fellowships (Hill and Wood), and donations from Compaq Computer Corporation, Intel Corporation, IBM, and Sun Microsystems.

1 Introduction

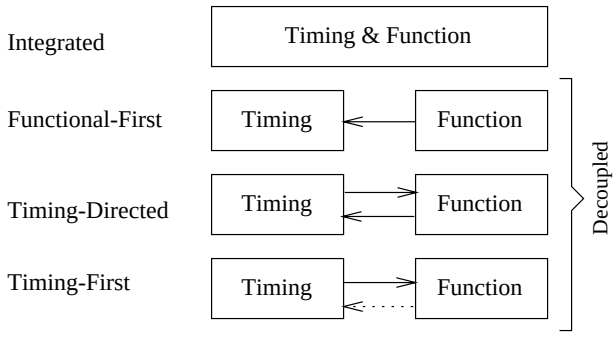
Execution-driven simulation is the preeminent method computer architects use to evaluate trade-offs in future computer system designs. Execution-driven simulators must meet two challenges. First, they must model computer system architectures with sufficient *functional fidelity* to execute workloads relevant to the system being designed. Second, they must model future computer system micro-architectures with sufficient *performance fidelity* to make meaningful performance projections of design alternatives. Performance fidelity depends on simulation precision (detail) and accuracy (error compared to real machines) [7].

Increasing challenges for execution-driven simulators. Meeting the twin challenges of functional and performance fidelity is becoming more difficult due to architectural, micro-architectural, and workload trends.

Future system architectures will likely exploit thread-level parallelism implemented with traditional multiprocessing, chip multiprocessing, and/or hardware multithreading. Thread-level parallelism permits multiple correct executions depending on how threads interleave. In real systems, the actual interleaving depends on the system's timing. For instance, assume a lock is free, but cached (and owned) by processor 1. Even if processor 2 executes a lock instruction a few cycles before processor 1, P1 will win the lock as its request will be seen at the cache first. To precisely model these *timing-dependent outcomes* in a simulator, the system's timing must determine the actual thread interleaving.

Future processor designs will likely grow in complexity, as architects design with increasing transistor budgets. Current designs already exploit instruction-level parallelism with pipelined, superscalar, speculative, and out-of-order execution. Processors speculatively execute (most) instructions that either commit (to become part of functional behavior) or are squashed (to affect performance only). As speculation can significantly affect system performance, timing simulators ordinarily model it in detail. Non-processor performance models, such as interconnects and memory, may also increase in complexity, as their (already large) contribution to system performance grows.

For many years user-mode-dominated, single-thread programs, such as the SPEC benchmarks [26], were considered sufficiently representative of end-users' workloads, particularly in simulation environments. Going forward, many designers are interested in the performance of applications such as databases and web servers. These commercial workloads spend up to one quarter of their total execution time in



Arrows indicate inter-simulator interactions per committed instruction.

Figure 1. Simulator Organizations

the operating system [3]. Simulators with high functional fidelity, called *full-system simulators*, can simulate OS code because they precisely model devices (e.g., ethernet, disks). *Static full-system simulators* playback a recorded trace of system operation, while *dynamic full-system simulators* are execution-driven and allow system behavior to be affected by timing (e.g., thread interleaving).

In summary, future execution-driven simulators must attain greater functional and performance fidelity even as: thread-level parallelism creates timing-dependent outcomes, micro-architectures increase in complexity (potentially using more speculation), and workload trends require more functionality.

Structuring execution-driven simulators. As functional and performance models become more complex, execution-driven simulators should be structured to manage complexity. Figure 1 shows four possible simulator organizations, each with a different type of coupling between its functional and performance components.

An integrated simulator tightly-couples function and performance models, literally modeling the operation of all the system's components. The challenge for the integrated approach is to address the conflicting demands of precision, accuracy, flexibility, and performance in one simulator. Integrated simulators can be highly detailed, modeling speculative execution (producing values and side-effects) and timing-dependent outcomes. The flexibility of integrated simulators is hampered as new devices and new performance models (e.g., modeling a radically different micro-architecture) can potentially interact with each other. Complexity and frequent modifications can lead to functional bugs that are difficult to isolate and fix, as their effect may be detected millions of cycles after they occur.

There are several decoupled organizations that address functional and performance fidelity in separate code. The goal of decoupling is to reduce complexity, thereby gaining more flexibility and potentially other benefits (correctness checking or faster development). However, decoupled designs introduce redundancy that may reduce simulation performance. They also typically make one simulator subordinate to the other and make modeling interactions between function and performance more difficult. Next we discuss two existing organizations—functional-first and timing-directed—and then advocate a new organization called *timing-first simulation*.

A functional-first simulator uses a functional component to produce a (logical) stream of committed instructions that are fed to a timing component. Trace-driven simulators [28] are well-known examples of functional-first simulation. Static full-system simulators are functional-first simulators that use traces that include OS code and device events. Unfortunately, functional-first simulators have difficulty modeling speculative execution and cannot model timing-dependent outcomes between threads. If the functional component is augmented to support speculation, this organization becomes similar to a timing-directed simulator.

A timing-directed simulator lets a timing simulator direct a functional simulator to execute speculative paths and to select thread interleaving. This organization successfully models speculative execution and multithreading with two costs. First, the functional simulator must be augmented to execute each dynamic instruction in stages (e.g., fetch, ready, commit) and to support speculation down multiple alternative paths. Second, this design is more tightly-coupled than other designs to allow the timing simulator to choreograph partial functional execution (i.e., neither simulator is ordered before the other). If coupling becomes too great, this organization becomes similar to an integrated simulator.

Our contribution: Timing-First Simulation. Our first contribution is to define the timing-first decoupled simulation approach, in which the timing simulator executes each dynamic instruction ahead of the functional simulator. The timing simulator models micro-architectural features with enough detail to model speculative execution and predict the interleaving of inter-thread events. To do this, the timing simulator must also model architectural function mostly correctly. When the timing simulator commits instructions, it invokes the functional simulator to verify if the timing simulator has deviated from the functional simulator. On a deviation, the timing simulator's state is repaired to guarantee functional fidelity. If deviations are rare, timing-first can obtain good performance fidelity. Timing-first can be viewed as an almost correct integrated simulator followed by a correct functional simulator checker.

Timing-first has several advantages relative to an integrated simulator. First, its timing simulator need not model components that minutely contribute to the total execution time but are necessary for full-system simulation (e.g., PCI bus transfers and clock timer interrupts). Second, our experience is that the timing-first approach reduces development time, as it can leverage existing simulators, and provides immediate debugging feedback to the timing simulator. Third, timing-first simulators can detect and recover from deviations, producing approximate timing results in situations (like new workloads or timing model changes) where other simulators would crash. Relative to a functional-first organization, a timing-first simulator models speculation and timing-dependent outcomes more precisely. Relative to a timing-directed organization, a timing-first simulator is more decoupled and requires less features in its functional simulator (but require more features in its timing simulator).

Our second contribution is a description (Section 3) and analysis (Section 4) of TFsim, the first implementation of a full-system timing-first simulator. TFsim models a pipelined, out-of-

order micro-architecture in detail, as well as a cache-coherent multiprocessor memory system. As its timing simulator only implements a subset of all instructions, we measure how frequently it deviates from the functional simulator and calculate the performance error this could introduce in the timing results. The overhead of the timing-first approach is the redundant execution of instructions by the timing simulator. We measure the overhead in TFsim to be 18-36% of the total execution time, and we explore a technique to reduce this overhead by verifying bundles of instructions. We measure TFsim's absolute performance and find it to be comparable to previously published simulator performance results. With its development and debugging advantages, timing-first simulation is a promising approach for building full-system performance simulators.

2 Related Work

Table 1 compares TFsim to previous simulators that either have a decoupled organization, have an out-of-order processor model, or support dynamic full-system simulation. Table 1 does not include the many simulators that only model in-order processors [8] or implement only static full-system simulators (e.g., with system event traces).

Several previously published simulators are decoupled, but they are not dynamic full-system simulators. The idea of decoupling functionality from timing has its origins in trace-driven methodologies. The Multiscalar [6], MASE [18], and SimpleMP [22] simulators use decoupling as a development tool, running a simple, in-order functional component to validate a complex, out-of-order component. In these three simulators, both components functionally execute the instructions. Fastsim [24] and SimpleScalar's sim-outorder [2] are functional-first simulators that support modeling of speculative wrong path execution. ASIM [14] is a decoupled simulation framework. In one configuration, it uses a functional-first decoupling to implement a static full-system simulator. In another, it pioneers the timing-directed organization.

Conversely, several previously published simulators are dynamic full-system simulators, but are not decoupled. g88 [4], gsim [19], and Talisman [5] are in-order functional simulators for the 88000 processor, which can simulate a modified version of UNIX. SimOS is a dynamic full-system simulator that supports out-of-order processor models for the MIPS [23] and Alpha instruction sets [3]. PharmSim is a dynamic full-system simulator based on SimOS and SimpleMP with an out-of-order processor model for the PowerPC instruction set [7]. Virtutech Simics [20] is a commercial simulator that supports system-level simulation of five target architectures: Alpha, IA-32 (x86), PowerPC, SPARC, and x86-64. Simics can boot unmodified operating systems, and it can be extended for cache timing simulations, but it only models simple (scalar, in-order) instruction execution.

TFsim, introduced in this paper, uses Simics as its functional component and is the first dynamic full-system timing-first simulator.

Name	Dynamic Full System	Out-of-Order	Multi-processor	Decoupled
Multiscalar Simulator [6]		Yes		Yes
FastSim [24]		Yes		Yes
SimpleScalar 3.0 [2]		Yes		Yes
MASE [18]		Yes		Yes
RSIM [15]		Yes	Yes	
SimpleMP [22]		Yes	Yes	Yes
ASIM [14]		Yes	Yes	Yes
g88 [4]	Yes†			
gsim [19]	Yes†		Yes	
Talisman [5]	Yes†		Yes	
Simics [20]	Yes		Yes	
PharmSim [7]	Yes†	Yes	Yes	
SimOS [23]	Yes†	Yes	Yes	
TFsim [This paper]	Yes	Yes	Yes	Yes

† indicates OS modifications are required.

Table 1. Summary of related simulators

3 Timing-First Simulation

Timing-first simulation is a decoupled organization in which a timing simulator runs ahead of a functional simulator. The timing simulator executes instructions using a mostly correct functional implementation of the instruction set. This execution is compared to, and corrected by, the functional simulator.

3.1 Timing-First Mechanisms

In timing-first simulation, the timing simulator controls the advance of each processor in the functional simulator. Therefore, it can create timing-dependent outcomes by advancing one processor before another, but it does not modify the actual architected state of the functional simulator. This preserves the independent, correct execution of the functional simulator. This reflects a decoupling principle—the timing simulator does not modify the functional simulator's registers, memory, or devices. Therefore, the timing simulator sequences the instructions in the system, but the functional simulator is the ultimate authority on their register, memory, and I/O effects.

The timing simulator must verify its functional execution of every instruction. This verification occurs as instructions commit (when they become non-speculative). The timing simulator retires an instruction, steps the functional simulator, and checks that the two simulators' architected states match. As most instructions only modify one register, one optimization that we implement is to compare only the destination register (in addition to some control registers, such as the program counter). Instructions that pass this retirement check are called *compliant*, otherwise they are called *deviant*. When the timing simulator detects a deviation, it reloads its architected state, resets parts of its micro-architectural model, and restarts execution. To assure that subsequent instructions are executed correctly, it eliminates all other speculative instructions (i.e., squashes its pipeline). This recovery does not affect memory

values, as no speculative data leaves the processor (similar to real systems). However, the recovery changes the simulated system's timing, as the pipeline squash is a simulation artifact. The simulated system's timing is delayed *n*-cycles (where *n* is the length of the target micro-architecture's pipeline), and not all micro-architectural state is patched up on mispredictions (e.g., there may be some pollution of the cache). If recoveries are as infrequent as we observe in our implementation (only 3 per 100,000 instructions committed), their timing impact should be small (as shown in Section 4.1).

The timing simulator in this organization does not model devices. As such, it cannot model device accesses or interrupts. However, the timing simulator is able to detect when these events occur and correctly model their timing. Device accesses cannot be speculatively executed in real machines, as they have side-effects that can be non-recoverable (e.g., sending a file to the printer). When the timing simulator detects that an instruction is accessing a device (as its physical address is in the I/O range), it delays producing a value until retirement. At retirement, it copies the value from the functional into the timing simulator, imitating the non-speculative execution of a real system. Interrupts are similarly detected and handled at retirement time.

The timing simulator models the micro-architecture and memory system, in addition to implementing the functional execution of most instructions. It does this by modeling out-of-order execution precisely (e.g., renaming registers, passing values using a physical register file), and executing code that implements the instruction set architecture independently of the functional simulator. As the timing simulator can execute instructions, speculation is implemented by literally making predictions, executing based on the predictions, and validating them before retirement. The following section describes our implementation of a timing-first simulator in more detail.

3.2 Our Implementation: TFsim

TFsim is an implementation of a timing-first simulator that combines a timing simulator with Virtutech Simics functional simulator. TFsim's timing simulator is a portable C++ program that compiles under IA-32 (x86) or SPARC into a dynamic library. It implements most user and privileged instructions in the SPARC V9 instruction set, including parts of the multimedia (VIS) extensions. It performs address translation, takes common traps (e.g., TLB misses, save and restore traps), can execute both in both 32-bit and 64-bit addressing modes, and models user and privileged control registers. How TFsim models its timing and functional components is detailed in Table 2.

TFsim's timing simulator does not implement the complete SPARC V9 instruction set, nor does it implement full-system functionality. The UltraSparc User's Manual [25] defines 183 instructions, whereas the timing simulator only uses 103 of them in all of our workloads. The timing simulator does not implement any of the devices necessary for dynamic full-system simulation including: SCSI controllers and disks, PCI and SBUS interfaces, interrupt and DMA controllers, and temperature sensors (all of which are implemented by Simics).

Timing Simulator	The timing simulator's target is a shared-memory multiprocessor system with MIPS R10000-like out-of-order processors [31] executing the SPARC V9 instruction set. Each processor models a 4-wide machine, with a 64-entry instruction window and an eleven-stage pipeline. The pipeline stages (and latencies) are fetch (3), decode (3), schedule (1), execute (1 or more, depending on operation), and writeback (3). TFsim models a 64-KB YAGS direct branch predictor [13], a cascaded indirect branch predictor [11], and a return-address stack predictor [16]. L1 caches model 64 KB, four-way set-associative split instruction and data caches (with single cycle access times). The L2 cache models a unified 4 MB 4-way set-associative exclusive cache (with a twelve cycle access time). Main memory has an eighty-cycle access time. The memory system models a MOSI broadcast snooping protocol and a interconnection network composed of a hierarchy of switches.
Functional Simulator	We use Virtutech Simics [20] to model a Sun E6000-like multiprocessor with one through sixteen processors using the SPARC V9 instruction set. Our target machine runs unmodified Solaris 8 and commercial workloads and has 2 GB of physical memory, multiple SCSI disks, and ethernet.

Table 2. Implementation details of timing and functional simulators

The timing simulator uses Simics's application programming interfaces (APIs) to advance a processor in Simics a given number of instructions and to access the architected state of a given processor and its memory management unit (MMU). It uses these APIs to perform the retirement check by stepping Simics and reading register values (for the check and recovery, if necessary). The interfaces are established when the timing simulator library is dynamically loaded.

TFsim's timing simulator uses some structures in Simics, including the memory image and MMU state. Since all values in memory are non-speculative, the timing simulator uses values (for loads) from Simics's memory image. It models speculative memory values by storing and forwarding them from an object modeling the load/store queue. Finally, TFsim's timing simulator uses Simics's MMU to perform address translation. This models the timing of a real system that uses a non-speculative MMU.

TFsim's timing simulator uses other Simics interfaces to perform some optimizations. For performance reasons, we cache decoded instructions on a per-program basis. When a different program is scheduled, the timing simulator is notified by Simics through a standard callback interface, so it knows to fetch instructions from a different address space. A similar callback is used to detect when interrupts occur, so they can be handled without causing recoveries.

The timing of device events, such as disk accesses and clock timer interrupts, is determined by Simics. Simics maintains two queues, one for events in absolute times, and one for events relative to the number of instructions retired. These two queues are directly related by the simulated processor frequency. As the timing of the device layer is related to the num-

ber of instructions that are retired in Simics, we believe that high instruction per cycle (IPC) workloads will see device events faster than low IPC workloads. To compensate for this, Simics's processor frequency could be divided by the IPC of a given workload to normalize the frequency of device events.

Implementing a privileged-mode out-of-order simulation of the SPARC V9 instruction set requires dealing with register windows, global register sets, aliased floating-point registers, traps, and control registers. Register windows allow faster function calls and returns by allowing the program to see a portion of a larger register file. To rename integer registers, we maintain a speculative version of the current window pointer (CWP) at fetch and rename a flattened view of the global and integer register file. The floating-point register file has aliased floating-point registers, in which single precision registers are overlapped with double precision registers. To model this aliasing, we rename on the single-precision granularity. Every instruction can be tagged with an exception code that is detected at commit to redirect control to trap handlers. Instructions that modify control registers rename all control registers as a set, permitting more than one to be in-flight at once.

TFsim supports the ability to speculate past any number of branches using predictors, and it implements a simple load-value predictor. These two predictors show that the timing-first organization can support different types of micro-architectural speculation.

3.3 Software Development Time

TFsim's development is the effort of one graduate student working half-time over the course of a year and a half. The implementation uses several pre-existing components, including Virtutech Simics [20], a multiprocessor memory timing simulator [21], and a user-level, out-of-order timing model for the Alpha instruction set [32]. The majority of new development involved converting the timing model to SPARC V9, extending it to execute privileged instructions, and integrating the components. While software development time reflects a host of factors, including experience, planning, motivation, desired features, and end quality, this is a remarkably short time to develop a dynamic full-system, out-of-order timing simulator. Leveraging existing simulators and the software development and debugging advantages in decoupled simulation greatly speeded the implementation of this system.

3.4 Memory Consistency Model Caveat

Timing-first simulators operate with memory values verified by the functional simulator's memory consistency model. Memory consistency models constrain the order in which memory operations appear to execute in a multiprocessor system [9]. Sequential consistency (SC) requires that memory accesses (appear to) execute in a total order that respects the program order on each processor [17]. SPARC V9, the target architecture for TFsim, supports total store order (TSO) [29]. TSO allows all SC executions, as well as additional executions in which a processor's store is ordered after some of the same processor's loads (but not after other stores). Thus, a TSO implementation can employ a first-come-first-serve write buffer to hold stores (that cache miss), even as subsequent loads access the cache.

Name	Description
OS Boot	Boot of unmodified 64-bit Solaris 8, using a reconfiguration option to detect the disks and devices present. The first 1.2 billion instructions are simulated after skipping the first 50 million instructions.
OLTP	On-line transaction processing based on the TPC-C v 3.0 benchmark [27] using IBM DB2 version 7.2 EEE.
Dynamic Web	An open-source dynamic web message posting system used by `www.slashdot.com`.
Static Web	The open-source web server, Apache, serving static web pages.
Barnes-Hut	A scientific workload from the SPLASH-2 [30] benchmark suite with 128K bodies.

Table 3. Summary of workloads

Timing-first simulators can model the timing performance of many alternative memory consistency models. For example, when modeling TSO, stores would complete in the timing simulator when they enter the (non-speculative) write buffer. Nevertheless, the functional effects of stores will only become visible in program order. This is because most dynamic full-system functional simulators maintain a single copy of memory and interleave processor memory accesses, thereby implementing sequential consistency. Therefore, timing-first simulators can model the performance of TSO for only SC memory interleavings. We are investigating ways to relax this restriction.

4 Evaluation

We evaluate TFsim using a suite of commercial workload benchmarks to characterize its performance accuracy, overhead, and simulator performance. As some instructions are not implemented by the timing simulator, we measure how often TFsim's execution deviates and discuss the worst-case performance error that could be introduced. We perform two sensitivity analyses: one that characterizes the deviations and performance error introduced in a less complete implementation of the instruction set, and another that increases performance by verifying bundles of instructions. Finally, we present an absolute performance comparison between TFsim and RSIM [15], a previously published out-of-order simulator.

Table 3 presents the benchmarks we used to evaluate TFsim's performance accuracy and overhead. Alameldeen et al. [1] contains a detailed description of these workloads, including their setup and configuration options. We simulate a total of 200 million instructions in all runs except the OS boot (which is run for 1.2 billion instructions). For example, each processor in a four processor system would execute approximately 50 million instructions.

4.1 Timing-First Performance Accuracy

Performance measurements done using a timing-first simulator may be less accurate than those done using a completely correct integrated simulator. The reduction in performance accuracy is related to the frequency of deviant instructions and

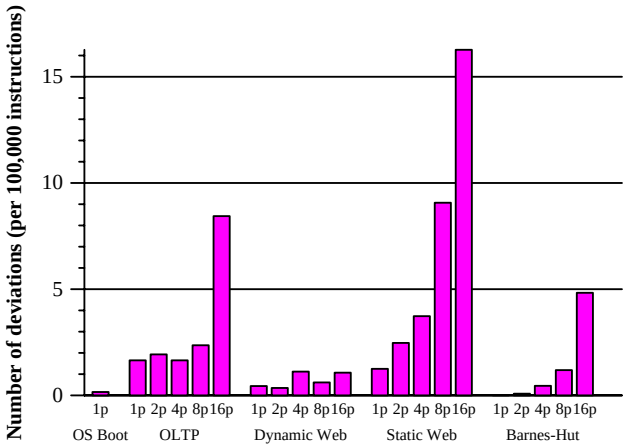


Figure 2. Deviations per 100,000 instructions

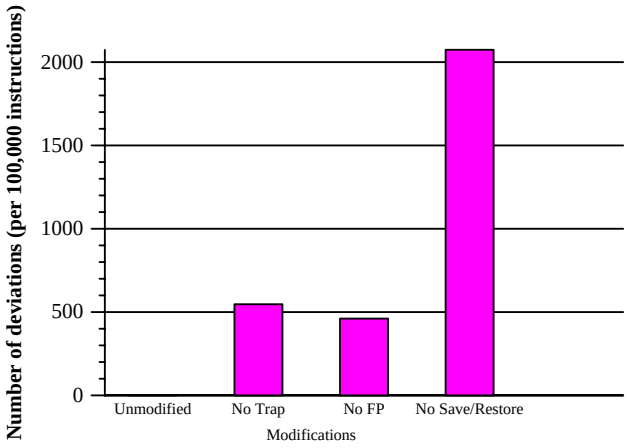


Figure 3. Number of deviations after modifications

the deviant instruction's correct timing. Figure 2 shows the number of deviations we observed per hundred thousand instructions committed by TFsim. On average, there are only three deviations per 100,000 instructions, representing 0.003% of all dynamic instructions. Multiprocessor results tend to have progressively more deviant instructions that are primarily loads and atomic swaps.

The performance error introduced by these deviant instructions is the difference between their correct timing and the timing provided by TFsim. The correct timing for these instructions could be as little as zero cycles or as much as a multi-hop memory miss, which we will estimate as two main memory latencies (160 cycles). The timing impact of deviant instructions in TFsim is a pipeline squash (11-cycles in our model). We calculate the worst-case performance error for these workloads to be 2.4% (149-cycle difference for 0.016% of all instructions). With a sustained IPC of 2, the upper-bound on the IPC error would be 4.8%. In practice, we believe the actual performance error will be much less than this, because the assumption that every deviant instruction is a multi-hop memory miss is not likely to be true. Assuming deviant instructions have average memory latencies (for instance, twelve cycles), they introduce less than 1% IPC error. One recent simulator validation study found a tuned simulator to have an average IPC error of 2% on microbenchmarks, and IPC errors ranging from -38% to 40% on macrobenchmarks, compared to a real machine [10].

The performance error introduced by deviant instructions is both stable and pessimistic. For a given workload, regardless of how micro-architectural parameters are varied, the numbers of deviant instructions at commit will be similar. Thus, the relative errors between two simulations will be much smaller than the absolute errors. Second, the performance error is generally biased to be pessimistic, since most instructions execute in less time than a pipeline flush. Thus when developing a timing simulator to evaluate a new microarchitectural idea, preliminary timing results will underestimate final performance. This has the desirable property that good preliminary performance results get better as the simulator is refined.

4.2 Accuracy Sensitivity Analysis

Timing-first simulators can produce results before the timing simulator's implementation is complete. In this section, we present a sensitivity analysis of the performance accuracy of these early simulators. We compare our full implementation with one that has some instructions disabled (representing an incomplete implementation), using the representative uniprocessor OLTP benchmark. The decrease in performance accuracy depends on the timing of the original instructions. We disabled three classes of instructions, each with different timing behaviors: excepting instructions (that always squash), floating-point instructions (that rarely squash), and save and restore instructions[1] (that sometimes squash). Deviant instructions cause the pipeline to be squashed in the timing simulator (introducing an 11-cycle delay).

Figure 3 shows how these modifications increase the number of deviant instructions observed in simulation. Figure 4 shows how these modifications affect the total (simulated) cycle time. The observed bar is the percent increase in cycle time found by simulation, and the expected bar is the product of the squash penalty and the number of new deviant instructions, as a percent of the original total simulation cycle time.

Instructions that cause exceptions (that normally squash) have the same timing if they are implemented or deviant. For instructions which do not squash, like floating-point instructions, the performance error introduced is identical to the squash penalty. For instructions like saves and restores that squash part of the time, the change in timing is less than the expected change. This analysis shows that the performance error introduced by deviant instructions ranges from zero to the pipeline squash penalty, depending on the instruction's correct timing behavior.

1. SPARC has integer register windows. On function calls, 'save' allocates a new window context. On function returns, 'restore' pops the window context. These instructions raise exceptions on window over- and under-flow conditions.

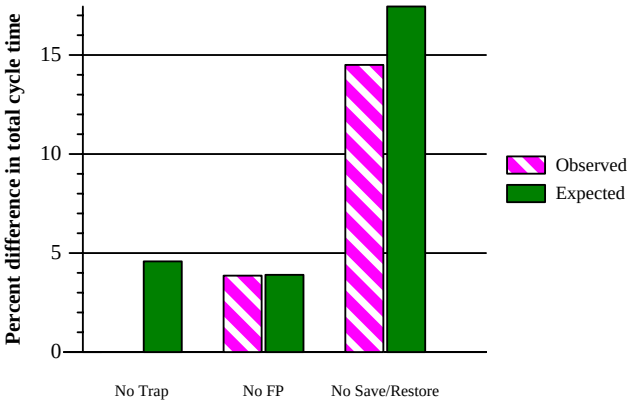


Figure 4. Increase in total cycle time under modifications

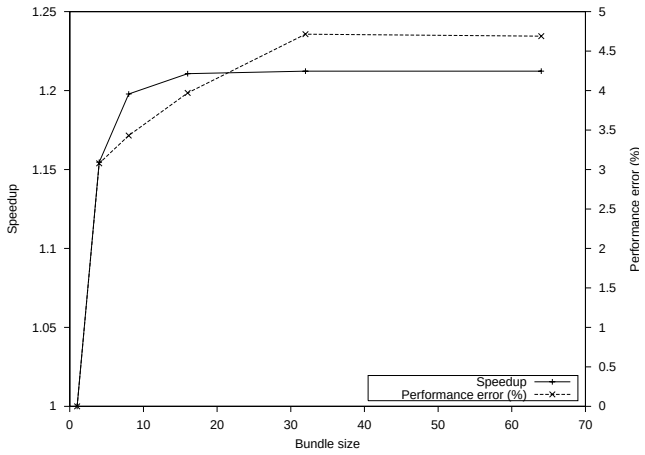


Figure 5. Speedup and performance error from verifying bundles of instructions

4.3 Timing-First Overhead

Timing-first simulators redundantly execute instructions in both the timing and functional simulators. We calculate overhead, compared to an equivalent correct integrated simulator, by dividing the functional simulator's execution time by the total simulation time. As in-order functional simulators are much faster than out-of-order timing simulators, our intuition is that timing-first simulation should be a low overhead technique. We instrumented the timing simulator to measure the wall clock total simulation time and the time spent advancing Simics. The overhead for uniprocessor benchmarks is between 12-21%, with an average of 18%. For multiprocessor benchmarks, the average overhead increases from 18% for two processors up to 36% for sixteen processors.

Timing-first simulators can verify more than a single instruction at a time. The main advantage to verifying bundles of instructions is that it increases Simics's performance. We observed an order of magnitude increase in speed between stepping Simics one instruction compared with hundreds of instructions. Simics implements threaded-code simulation, and the overhead of restarting the simulator thread is fixed irrespective of the step size.

For the uniprocessor OLTP benchmark, Figure 5 plots the speedup and performance error for verifying bundles of instructions, relative to a simulation that verifies every instruction. By bundling instructions, simulation speeds up by 22%, and the overhead due to Simics is reduced from 19% to 6%. The reduction in overhead is responsible for speeding up the simulation 15% (1/(1-0.13). We hypothesize the additional improvement is due to better cache performance from the reduced context-switching. The figure shows that reducing verification to every eight instructions gives a speedup of almost 1.2 at the cost of 3.4% in performance accuracy. This technique could be used during initial state-space exploration (reducing turn-around time) and then turned off for final experiment runs.

The benefit of verifying bundles of instructions depends on the relative speed of the functional and timing simulators. If the timing simulator is faster, for example, on simple benchmarks that commit a greater portion of their instructions, the speedup is greater. We observed that for some SPLASH-2 kernels [30], using a simple uniprocessor cache hierarchy, the overhead from Simics could be as high as 42%. For these kernels, bundling is much more effective (enabling simulation to run up to 114% faster).

When multiple instructions are verified as a bundle, the functional simulator does not provide any intermediate changes to the timing simulator. Instead, the timing simulator verifies that the architected state of all registers written in the bundle match between simulators. If an instruction writes the incorrect value in a register, it may be masked by a later (correct) instruction in the same bundle. Some optimizations, like detecting interrupts, are impossible to implement with bundled verification. Device accesses, like accesses to the MMU, cause the bundle to be verified immediately, as these instructions are functionally implemented by Simics. The timing simulator must also track memory values for stores that have committed but have not been verified. We now examine the absolute performance of TFsim compared to a previously published simulator.

4.4 Performance Comparison

Timing-first simulation trades some performance for the software development advantages of decoupled simulation. In this section, we compare TFsim's performance to previously published results. A compelling simulator performance evaluation would compare TFsim with another dynamic full-system simulator that has the same target system, host system, architecture, micro-architecture, and workload. Unfortunately, the prime candidate for comparison, SimOS, models the MIPS architecture running the IRIX operating system, with published slowdown comparisons to a MIPS host. TFsim models a different target architecture (SPARC V9) running a different operating system (Solaris), and it is hosted cross-platform on a Pentium III system running Linux. This cross-architecture and cross-platform hosting makes measuring slowdown versus native execution measurements difficult. As no ideal comparison is possible, we present the following limited comparison.

We compare the absolute performance of TFsim and RSIM in terms of kilo-instructions executed per seconds (KIPS), using details from Durbhakula et al. [12]. The performance of RSIM presented in Hughes et al. [15] is similar (an average of 26

Application	RSIM Base	RSIM Scaled	MP TFsim	Uniprocessor TFsim
Radix	15.7	75.4	31.0	102
FFT	23.5	112.8	39.3	132
LU	27.6	132.5	38.9	119
Average	22.3	106.9	36.4	117.6

Table 4. Simulated KIPS comparison

KIPS on a 16.5 SPECint95 machine). Both simulators model the out-of-order execution of the SPARC V9 instruction set. We configure TFsim's micro-architectural parameters to match RSIM's as closely as possible. However, the micro-architectural models are still quite different. TFsim models address translation and an instruction cache, both of which are not currently done in RSIM, and it uses a different branch prediction scheme. We measured the wall clock time of TFsim to simulate the same three kernels from the SPLASH-2 [30] benchmark suite to completion on an unloaded 933 MHz Pentium III system with RedHat Linux 7.2.

Table 4 presents the results of the performance comparison. The RSIM base performance results are measured on a 250 MHz UltraSparc-II (SPARC V9) system [12]. To normalize for host machine speed, we present RSIM results scaled by a factor of 4.8 (the relative SPECInt95 ratings of the systems are 9.75 and 46.8) [26]. MP TFsim presents performance results using a detailed multiprocessor cache model (that models cache coherence and network traffic). Uniprocessor TFsim does not model coherence, and verifies bundles of 100 instructions (introducing an average performance error of 4% compared to unbundled execution). This cross-platform comparison of simulators, using kernels as benchmarks, is very limited. However, we believe it demonstrates that TFsim's absolute performance is comparable to previous simulators, making timing-first an attractive alternative organization given its other benefits.

5 Conclusions

This paper defines and evaluates timing-first simulation, a decoupled organization for building dynamic full-system performance simulators. The organization decouples the high functional fidelity requirements of dynamic full-system simulators from the performance fidelity requirements of timing simulators. The goal of this approach is to enable faster development of flexible, stable, less complex simulators, and to enable the exploration of radical alternative architectures. In this organization, the timing simulator implements the functional execution of instructions important to performance, backed by a completely correct functional simulator.

TFsim, our implementation of a timing-first simulator, precisely models speculative execution of out-of-order processors in a shared-memory multiprocessor, running commercial workloads on an unmodified operating system. TFsim is implemented using Virtutech Simics [20], an existing dynamic full-system functional simulator. To our knowledge, TFsim is the first dynamic full-system timing-first simulator. TFsim's timing simulator can functionally execute 99.997% of all dynamic instructions seen in our commercial workloads, and little performance error is introduced by the small number of deviations. The overhead of functional execution is 18% for uniprocessors and can be reduced further at the cost of some reduced performance accuracy. These results show that decoupled timing-first simulation is a promising alternative for rapid development of dynamic full-system timing simulators.

Acknowledgments

We thank Craig Zilles for allowing us to use his micro-architectural timing model, Milo Martin for his support integrating the multiprocessor memory timing model, and Ravi Rajwar for describing SimpleMP's operation. We also thank the following people for their comments on this work and/or paper: Alaa Alameldeen, Harold Cain, Joel Emer, Peter Magnusson, Dan Sorin, and Min Xu. Finally, this work would not be possible without the support of the entire Wisconsin Multifacet group.

References

[1] A. R. Alameldeen, C. J. Mauer, M. Xu, P. J. Harper, M. M. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Evaluating Non-deterministic Multi-threaded Commercial Workloads. In *Proceedings of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 30–38, Feb. 2002.

[2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, Feb. 2002.

[3] L. A. Barroso, K. Gharachorloo, A. Nowatzyk, and B. Verghese. Impact of Chip-Level Integration on Performance of OLTP Workloads. In *Proceedings of the Sixth IEEE Symposium on High-Performance Computer Architecture*, Jan. 2000.

[4] R. C. Bedichek. Some Efficient Architecture Simulation Techniques. *Winter 1990 USENIX Conference*, pages 53–63, Jan. 1990.

[5] R. C. Bedichek. Talisman: Fast and accurate multicomputer simulation. In *Proceedings of the 1995 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 14–24, May 1995.

[6] S. E. Breach. *Design and Evaluation of a Multiscalar Processor*. PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, Feb. 1999.

[7] H. W. Cain, K. M. Lepak, B. A. Schwartz, and M. H. Lipasti. Precise and Accurate Processor Simulation. In *Proceedings of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 13–22, Feb. 2002.

[8] R. F. Cmelik and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. In *Proceedings of the 1994 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, May 1994.

[9] D. E. Culler and J. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., 1999.

[10] R. Desikan, D. Burger, and S. W. Keckler. Measuring Experimental Error in Microprocessor Simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266–277, July 2001.

[11] K. Driesen and U. Holzle. Accurate Indirect Branch Prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 167–178, June 1998.

[12] M. Durbhakula, V. S. Pai, and S. V. Adve. Improving the Accuracy vs. Speed Tradeoff for Simulating Shared-Memory Multiprocessors with ILP Processors. Technical Report TR9802, Rice University, 1999.

[13] A. N. Eden and T. Mudge. The YAGS branch prediction scheme. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 69–77, June 1998.

[14] 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, Feb. 2002.

[15] C. J. Hughes, V. S. P. Pai, P. Ranganathan, and S. V. Adve. Rsim: Simulating Shared-Memory Multiprocessors with ILP Processors. *IEEE Computer*, 35(2):40–49, Feb. 2002.

[16] S. Jourdan, T.-H. Hsing, J. Stark, and Y. N. Patt. The Effects of Mispredicted-Path Execution on Branch Prediction Structures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 58–67, Oct. 1996.

[17] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.

[18] E. Larson, S. Chatterjee, and T. Austin. MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling. *International Symposium on Performance Analysis of Systems and Software*, Nov. 2001.

[19] P. S. Magnusson. A Design For Efficient Simulation of a Multiprocessor. *First International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 69–78, Jan. 1993.

[20] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[21] M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Bandwidth Adaptive Snooping. In *Proceedings of the Eighth IEEE Symposium on High-Performance Computer Architecture*, Jan. 2002.

[22] R. Rajwar. Personal Communication, Oct. 2001.

[23] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(4):34–43, 1995.

[24] E. Schnarr and J. R. Larus. Fast Out-Of-Order Processor Simulation Using Memoization. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 283–294, Oct. 1998.

[25] Sun Microsystems. *UltraSPARC User's Manual*. Sun Microsystems, Inc., July 1997.

[26] Systems Performance Evaluation Cooperative. SPEC Benchmarks. http://www.spec.org.

[27] Transaction Processing Performance Council. TPC Benchmark C, Draft Specification, Revision 4.0.q, Aug. 1999.

[28] R. A. Uhlig and T. N. Mudge. Trace-Driven Memory Simulation: A Survey. *ACM Computing Surveys*, 29(2):128–170, 1997.

[29] D. L. Weaver and T. Germond, editors. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.

[30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, June 1995.

[31] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, Apr. 1996.

[32] C. B. Zilles, J. S. Emer, and G. S. Sohi. The Use of Multithreading for Exception Handling. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 219–229, Nov. 1999.