FLASH vs. (Simulated) FLASH: Closing the Simulation Loop

Jeff Gibson, Robert Kunz, David Ofelt, Mark Horowitz, John Hennessy
Computer Systems Lab
Stanford University
Stanford, CA 94305
fvf@vic.stanford.edu

Mark Heinrich
Computer Systems Lab
School of Electrical & Computer Engineering
Cornell University
Ithaca, NY 14853
heinrich@csl.cornell.edu

ABSTRACT

Simulation is the primary method for evaluating computer systems during all phases of the design process. One significant problem with simulation is that it rarely models the system exactly, and quantifying the resulting simulator error can be difficult. More importantly, architects often assume without proof that although their simulator may make inaccurate absolute performance predictions, it will still accurately predict architectural trends.

This paper studies the source and magnitude of error in a range of architectural simulators by comparing the simulated execution time of several applications and microbenchmarks to their execution time on the actual hardware being modeled. The existence of a hardware gold standard allows us to find, quantify, and fix simulator inaccuracies. We then use the simulators to predict architectural trends and analyze the sensitivity of the results to the simulator configuration. We find that most of our simulators predict trends accurately, as long as they model all of the important performance effects for the application in question. Unfortunately, it is difficult to know what these effects are without having a a hardware reference, as they can be quite subtle. This calls into question the value, for architectural studies, of highly detailed simulators whose characteristics are not carefully validated against a real hardware design.

1. INTRODUCTION

The FLASH project at Stanford was a large research effort exploring methods of building large-scale shared-memory multiprocessors. Like most architecture research today, we relied heavily on simulation technology to evaluate architectural trade-offs in the machine, and during the course of the research developed a number of simulation methods, some of which have been widely deployed [19]. Unlike many research efforts, the FLASH project built a hardware version of the machine, and a 32-node multiprocessor is currently in operation at Stanford. A running machine gives us the (dis)advantage of being able to compare the simulations we used to design the machine with the actual hardware to determine the accuracy of our simulators. This paper describes the results of this comparison.

The simulators in this comparison span a range of detail from a simple, single-issue processor model without an operating system to an out-of-order processor model with a complete simulated operating system. In addition, to look at both processor modeling effects and memory system effects, we compare simulation results with our detailed memory system simulator against those from a much simpler one. Each simulator was built well in advance of the real hardware and has been used for approximately six years to explore the behavior of the machine and make trade-offs as the machine was designed.

Our results surprised us—overall the accuracy was not very good, even for the most detailed processor and memory system models. In fact, our more complex processor models were sometimes worse than simpler models in terms of performance accuracy. One easy explanation is that the performance errors that we report in Section 3 were the result of poor simulator design on our part. This seems unlikely. While clearly our simulation system was not bug free, both RSIM [14] and SimpleScalar [2] share some characteristics that caused performance errors in our systems. Industry stories about performance differences between a new processor and the company's internal performance model are also common. It is hard to believe that the entire architecture community cannot design a good simulator. A much more plausible answer is poor performance matching simply is the expected result. Modeling modern processor performance is difficult, requiring the writer of the simulator to understand low-level details of the hardware, and to be able to model these effects without greatly slowing down or complicating the simulator. Bugs or omissions are common, and without a methodology that points out something is wrong, it is hard for the simulation writer to know that errors still exist. As a result, simulators need to be validated against real hardware before they provide reliably accurate performance predictions. This result is somewhat unsettling, since some of the finer design tradeoffs we made probably affected performance by a smaller amount than the size of our simulator error. Indeed, many contributions to the architecture research literature report performance gains whose magnitude is less than the simulator modeling error we have ob-

served.

Yet, inaccurate absolute performance would not be as critical a problem if the trends predicted by simulators were accurate. Architects rely on being able to predict the relative magnitude of performance changes across a variety of alternative designs. For example, in a multiprocessor like FLASH, it is critical to understand how the system will respond as processor counts increase. Our results in predicting performance trends are more encouraging, though still not perfect. The most surprising result is that the simple in-order machine model used for most of the FLASH simulations predicts trends as well as the slower, more complex out-of-order model.

2. HARDWARE AND SIMULATORS

This section briefly describes the FLASH hardware system, followed by detailed descriptions of each of the simulators in this study. It concludes with a discussion of the methodology we use to compare the simulation results to the real hardware, including the parameters that we vary and the applications we use.

2.1 Hardware

The hardware used in this study is a 16-processor FLASH (FLexible Architecture for SHared memory) machine [9]. FLASH is a cache-coherent distributed shared-memory multiprocessor that runs its cache coherence protocol on a programmable node controller, MAGIC. The exact details of FLASH are not important to this study. Rather, what is important is that FLASH is an aggressive design in both the processor and memory system with abundant concurrency and many complex interfaces (like many other modern machines, non-trivial to simulate). The performance of the FLASH hardware is used as the standard that all of the simulators attempt to match. Table 1 lists important parameters of the actual FLASH hardware.

FLASH uses the MIPS R10000 [27] as its compute processor. The R10000 is a good example of a complex, superscalar microprocessor. It has an out-of-order core that can issue and retire up to four instructions per cycle, as well as lockup-free on-chip primary instruction and data caches and a lockup-free backside secondary cache. The R10000 manages its own secondary cache so all data replies and intervention requests must pass through the processor to get to the cache. For this paper, the FLASH hardware is configured with a processor clock speed of 150 MHz and the MAGIC node controller running at 75 MHz. FLASH runs a slightly modified version of the IRIX 6.4 operating system. Some boot code and device drivers specific to the SGI Origin 2000 are altered to boot IRIX on FLASH.

2.2 Simulators

The FLASH design effort produced a number of architectural simulators over the course of approximately six years [7]. Because the focus of the FLASH project was innovation in the memory system, most of our simulations during the design process used FlashLite, our detailed memory system simulator. What varied as the project matured was the level of detail used in modeling the processor and the operating system. The two main simulators we use in conjunction with FlashLite, Solo and SimOS, are described below. In addition, to examine the importance of simulation accuracy in the memory system, we describe a simpler, generic NUMA memory system simulator that we will compare to FlashLite.

Parameter	Value
Processor	MIPS R10000
Number of Processors	1-16
Processor Clock Speed	150 MHz
System Clock Speed	75 MHz
Instruction Cache	32 KB, 64 B line size
Primary Data Cache	32 KB, 32 B line size
Secondary Cache	2 MB, 128 B line size
Max. IPC	4
Max. Outstanding Misses	4
Network	50 ns hops, hypercube
Memory	140 ns to first double-word
Cache Coherence Protocol	dynamic pointer allocation

Table 1: FLASH hardware configuration.

Solo. Our simplest simulation framework is Solo, a standard processor simulator that allows an application to be run directly on simulated hardware. Using an OS modeling technique that is commonly used in both industry and the research community, Solo models parallel threads and shared memory, but it emulates system calls. Tango Lite [4] and MINT [24] are two other good examples of this type of simulator. Since Solo does not model the operating system or any I/O behavior, it consists mainly of a processor simulator, a memory system simulator, and a set of backdoor routines for handling system calls and page mapping. Although lacking in system details, simulators like Solo are easy to build and are commonly used for architectural evaluation.

Solo uses the Mipsy processor simulator, which models a single-issue, in-order MIPS processor. Pipeline effects and functional unit latencies are not simulated, so the Mipsy processor executes one instruction per cycle in the absence of memory stalls. Mipsy has blocking reads, but supports both prefetching and a write buffer. Our Solo simulations use prefetching and a four-entry write buffer in an attempt to model the FLASH hardware as closely as possible.

We use Solo with the standard memory system simulator for the FLASH machine, FlashLite [5]. FlashLite is a multi-threaded simulator of the memory bus, MAGIC node controller, network, memory, and I/O subsystems. One of FlashLite's threads is a cycle-accurate emulator of the embedded protocol processor that runs the cache coherence protocol. Other FlashLite threads model MAGIC's external interfaces and internal functional units with latencies extracted directly from the Verilog RTL design [6]. FlashLite also uses the same cache coherence protocol that runs on the FLASH hardware, though there are small modifications to the protocol in simulation to speed the initial boot phase. Once the application runs, however, the two protocols are identical in instruction and data cache behavior. Although we refer to FlashLite as the "memory system" simulator, it actually models everything in the FLASH system other than the main microprocessor and its caches.

SimOS. The SimOS simulation environment [19], like the Talisman [1] and SimICS simulators [10], models the system in enough detail to boot and run a full operating system. We are able to run a slightly modified version of IRIX 6.4 on our simulated machine. The kernel modifications are mostly to the boot code and I/O device drivers due to dif-

ferences between FLASH and the simulated hardware. Like Solo, SimOS contains a processor simulator and a memory system simulator. In addition, SimOS models the virtual memory system (including the processor TLB), I/O devices such as disks, ethernet, and a console. System calls and page mapping are managed by the simulated operating system (just like the hardware) rather than being the responsibility of the simulator.

SimOS can be run using three processor simulators that provide varying degrees of detail. The fastest processor simulator is Embra, a binary translation system that runs at roughly 10x slowdown from the host microprocessor. Unfortunately, Embra does not model either the processor or the memory system in enough detail to draw any useful conclusions. It is indispensable, however, since it allows us to boot the operating system and position our workloads in a reasonable amount of time via checkpointing. For generating results, we restore from the Embra checkpoint and use one of the more detailed simulators, Mipsy and MXS, to run the applications. The Mipsy processor model is the same simulator used with Solo. MXS, our most detailed processor simulator, models an out-of-order four-issue microprocessor. Like other generic out-of-order processor simulators [2, 14], MXS does not model any real processor in particular. Rather, it is a generic superscalar processor model that we have configured to be as close to an R10000 as possible. MXS models pipeline latencies and bandwidth, and in our experiments has the same type and number of functional units as the R10000, as well as the same branch prediction strategy. Prior to this work, MXS lacked resource constraints on functional units, but we added these constraints to compare MXS fairly with our other simulators and to make it similar to other detailed processor simulators used in the literature. Like the Solo simulations, both Mipsy and MXS SimOS simulations use the detailed FlashLite memory system simulator.

NUMA. To investigate the importance of an accurate memory system model, we can replace FlashLite in both Solo and SimOS with a generic non-uniform memory access (NUMA) model. While FlashLite models the FLASH hardware in great detail, the NUMA simulator models the memory system of a generic NUMA machine. It simulates network latencies, contention for main memory, and the latency through the directory controller (MAGIC, in the case of FLASH). However, it does not model occupancy of the directory controller beyond the normal latency path, nor does it model contention in the network or the routers. We use NUMA as an example of the type of memory system simulator that we might have used had we never designed and built real hardware or had we been interested only in processor effects and not in the details of a particular memory system.

2.3 Methodology

Although Mipsy is a simple, single-issue processor model that makes no attempt to model the instruction-level parallelism of the MIPS R10000, simulators like Mipsy are widely used due to their ease of development and their speed (Mipsy runs 4-5 times faster than MXS). A common technique to allow single-issue processor models like Mipsy to model a multiple-issue processor is to increase the speed of the processor with respect to the memory system. The single-issue processor cannot take advantage of ILP, so the speed increase is necessary to enable the processor to make requests to the memory system at the same rate as the real hardware. The correct speed for the processor model is related to the ILP that the real processor will be able to exploit and is empirically determined. The MIPS R10000 processors in the FLASH machine run at 150 MHz. Mipsy results for processor speeds of 150 MHz, 225 MHz, and 300 MHz are presented in Section 3 to show how well this common technique works in practice. We do not use 4 * 150 MHz = 600 MHz, even though the R10000 can issue and retire four instructions per cycle, because the processor never sustains this peak performance. We find that 300 MHz is more than sufficient to compensate for ILP. Because MXS is a multiple-issue simulator capable of exploiting ILP, its results are reported only for the hardware processor clock speed of 150 MHz.

Application	Problem Size
FFT	1M points
Radix-Sort	2M keys
Ocean	514x514 grid
LU	768x768 matrix, 16x16 blocks

Table 2: SPLASH-2 problem sizes.

Our experimental setup fixes all the other parameters so that they are identical between the simulators and the real machine. The same application binaries are used for all platforms. The same protocol is used in FlashLite and on the real hardware (with transparent changes to the system boot routines). The simulated kernel and the real kernel differ in many of their device drivers, but for this study there is no console or network I/O during the application runs, and we take the average of at least 5 hardware runs to avoid reporting any spurious system effects.

The SPLASH-2 applications [25] used in our study (FFT, Radix-Sort, LU, and Ocean) were all compiled for use with Solo, which has more restrictive requirements than either SimOS or the hardware about needing older, statically-linked ELF32 binaries. This is not a limitation of this study, and it allows us to use identical executables on Solo, SimOS, and the FLASH hardware. Each application includes hand-inserted prefetch instructions to hide read latency and improve parallel performance, and multiprocessor versions perform data placement to minimize communication and coherence traffic. The problem sizes we use are shown in Table 2. Note, however, that no matter how well or how poorly the applications perform on the hardware, the simulators should predict their performance!

3. RESULTS

In this section, we show how well our range of processor and memory system simulators predict actual hardware performance. We analyze where the discrepancies occur for each simulator and find that often the inaccuracies are intrinsic to the type or class of simulator and not simply our implementation. In addition, we show that significant errors still occur in carefully designed simulators that were used to build real hardware. Had we not compared our simulator results to the hardware performance, many of these errors would have gone unnoticed.

We then explore how well the simulators predict trends. There is a commonly held belief that even though simplified simulators will not give accurate estimates of absolute performance, they can still be used to predict speedup and

other architectural trends. We investigate the ability of our simulators to predict speedup and evaluate how sensitive these results are to the memory system model used.

3.1 FLASH vs. (Simulated) FLASH

We begin by studying how well our simulators predict actual hardware performance. We describe sources of error in our simulations and how we tuned the simulations to better model the hardware.

3.1.1 Initial Performance Comparison

For our initial performance comparisons, we ran several applications from the SPLASH-2 suite on both the FLASH hardware and our simulators to see how well the simulators predict FLASH hardware performance. We examined execution time for the parallel section of each application, normalized to the execution times on the actual hardware. The initial four-processor results were not encouraging.

Most, but not all, of the simulator configurations were faster than the real hardware, but none even relatively agree and they do not track each other. For instance, SimOS-Mipsy at 150 MHz gives a good prediction of the performance of Ocean (low by only 7%), but it under-predicts the execution time of Radix-Sort by 39% and over-predicts the execution time of LU by 53%. Many of the other simulators fared even worse. Solo-Mipsy at 300 MHz, for example, under-predicts the hardware execution time by 61%!

Given the confusion in our initial multiprocessor results, we simplify the picture and examine the initial results for uniprocessor runs of the same applications. These results are shown in Figure 1. The X-axis is broken down into simulator configurations for each application. The Y-axis gives the execution times relative to the real FLASH hardware. A value of 1.0 means that the simulator reported the same time as the hardware. Values below 1.0 signify that the simulator was executing faster than hardware and values above 1.0 mean that the simulator was running slower than hardware. Unfortunately, the simulators and the hardware are still very far apart. Because these applications rarely use operating system services, we expected the Solo-Mipsy simulations to produce nearly the same results as the SimOS-Mipsy simulations, but this is not the case.

3.1.2 Sources of Mismatch

With the hardware results in hand, we were able to find several sources of simulator error. The following sections categorize the sources of error into performance bugs, omission of large effects, and lack of sufficient detail. Within each category we describe the specific simulator problems that affect the quality of the simulation results.

Bugs. Performance bugs can be subtle but disastrous to the accuracy of simulation results. An easy trap to fall into is to believe that just because a program runs in the simulator without crashing or that the basic statistics seem reasonable, the simulator is giving you a meaningful prediction of performance. On the contrary, subtle performance bugs can live in a production simulator for years.

For example, the MXS simulator had a bug where an instruction would move through the pipeline too quickly if all of its resources were available when it issued. The simulator ran fine and generated results that were at first glance believable, since the circumstances that triggered the bug were not the most common case. This particular bug was found

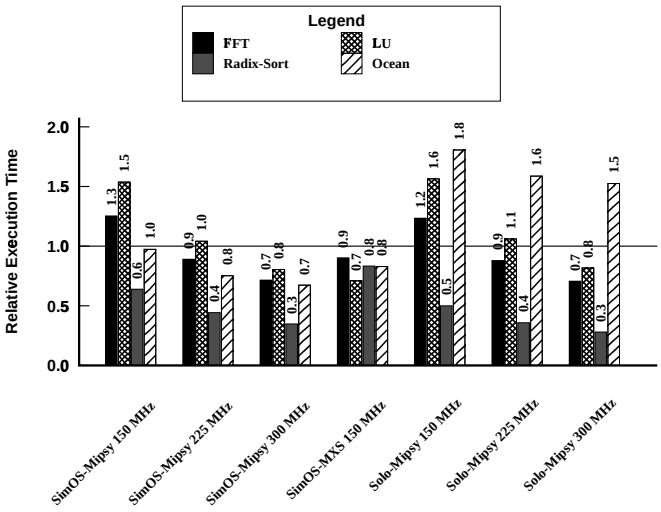

Figure 1: Initial uniprocessor SPLASH-2 results before simulator tuning.

by the Rivet group, which was visualizing the pipeline [22] and noticed the modeling error.

If MXS had been designed to model a specific processor and then had been validated against it, this type of bug would have been easy to find. However, MXS, like other publicly-available simulators (e.g. RSIM [14] and SimpleScalar [2]), was not designed to model a specific processor, so no such standard exists.

One other performance bug in the MXS simulator was that the MIPS `CACHE` instruction was not implemented correctly. The instruction had invalidated a dirty line, but the successful completion of the memory operation was not properly communicated to the processor. As a result, a processor would not graduate this or any other instruction for approximately one million cycles. Eventually a timing interrupt caused the processor to retry the operation, which immediately succeeded. Since the processor recovered and one million cycles was small relative to the total simulated run time of the application, this problem went unnoticed for months.

Omissions. Another class of problems in simulators is the deliberate omission of detail. The Solo simulator, for instance, does not model any operating system effects or the TLB. The lack of an operating system is well-known to any user of Solo, and it has long been assumed that as long as the applications under study did not use a substantial number of operating system services (and the SPLASH-2 applications certainly fit into this category), the operating system effects could be safely ignored. But with the advent of SimOS, we found that the omission of the TLB and the corresponding lack of modeling of TLB misses in Solo were more than a second-order performance effect. Although other researchers had also noted the importance of the TLB on performance [17, 23], we found the need to model the TLB for the already highly-tuned SPLASH-2 applications surprising.

The TLB of the R10000 is small (64 entries) and the penalty for a TLB miss is at least 65 processor cycles. The original SPLASH-2 studies [25] were done on a simulator without a TLB, and early SimOS studies showed that even some of the carefully-tuned SPLASH-2 applications experi-

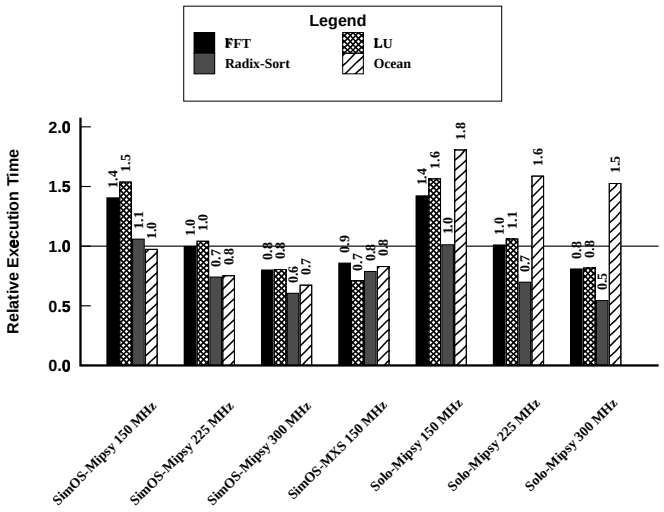

Figure 2: Uniprocessor SPLASH-2 results after blocking fixes.

ence a substantial number of TLB misses [18]. Since there are so few entries in the TLB, it is easy for a program to run in a regime where even though its working-set resides in the primary data cache, it incurs a substantial number of TLB misses. FFT, for instance, exhibits this behavior. The SPLASH-2 studies [25] recommended blocking FFT for the primary data cache, but with a million-point FFT, this leads to a TLB miss on every store during the transpose phase for both one and four-processor runs. Changing the blocking parameters to block for the TLB improves the performance of FFT by 14% on a uniprocessor and by 16% on four processors. Similarly, Radix-Sort has traditionally been run with a large radix to reduce overhead. This causes a pathological number of TLB misses. Reducing the radix from 256 to 32 on a 2 million key hardware run gives a 31% performance improvement on a uniprocessor and 34% on four processors.

Like many researchers, the authors have previously underestimated the effects of TLB misses. TLB performance is as important as cache performance and can easily be an application's primary bottleneck. When a simulator does not model the TLB [14, 25], the results should be viewed with skepticism, as a major performance effect is missing. As we will see later in this section, neglecting the TLB is not the only common simplification of the virtual memory system that can cause unacceptable simulator inaccuracies.

We changed the input parameters for FFT and Radix-Sort to give the optimal performance on actual hardware. All subsequent results reflect these TLB blocking fixes. Figure 2 shows the performance of our simulators running these tuned applications. Note that the simulated times for Radix-Sort are now much closer to the hardware times. While this makes sense for the Solo runs, it means that SimOS was also not modeling the cost of a TLB miss correctly. This is also described in more detail later in this section.

The uniprocessor Ocean results in Figure 2 are surprising because Solo predicts much slower times than either hardware or SimOS-Mipsy. At first glance, this seems impossible since the lack of operating system effects should only speed up the program. The cause of this behavior is that Solo predicts a secondary cache miss rate that is approximately three times higher than that reported by SimOS-Mipsy, due to an increased number of cache conflicts. Cache conflicts are caused by poor layout of physical memory, which is controlled by the operating system. Because Solo does not model an operating system, it performs physical memory allocation itself. Like many architectural simulators [2, 4, 14], Solo neglects the page-coloring algorithms used in modern operating systems. In fact, Solo is more realistic than many simulators that dispense with virtual memory altogether and operate with raw, physical addresses (or equivalently, a mode where physical addresses equal virtual addresses).

The reason that operating system writers worry about page coloring is that improper coloring can decimate the performance of an application. It is important that in simulations that do not include an operating system, architects give careful thought to page coloring. For instance, the uniprocessor cache conflicts predicted by Solo for Ocean do not occur on four processors, so Solo predicts huge superlinear speedup for Ocean that does not occur on the hardware.

We found that the lack of instruction-latency modeling is another deliberate simulator omission that causes performance mispredictions. Mipsy will execute one instruction every clock cycle, unless it experiences a cache miss. In reality, some instructions take far longer to execute. Radix-Sort, for instance, executes many high-latency operations such as integer multiplications and divisions. Similarly, Ocean executes many high-latency floating point operations. The effect is that Mipsy will tend to under-predict the execution times of Radix-Sort and Ocean. We will examine this behavior in more detail in the next section, once the simulators have been tuned to correctly model memory effects.

Lack of Detail. Even when effects are modeled by the simulator, they are not always modeled correctly. Neither Mipsy nor MXS was designed to be an accurate model of the MIPS R10000. The focus of the FLASH project has been on the memory system, so the purpose of our processor models was to generate believable streams of references that we could use to evaluate memory effects. In the following paragraphs, we describe how we were forced to tune our simulation parameters so that they matched the hardware for a set of simple microbenchmarks.

Unlike Solo, the SimOS-Mipsy and SimOS-MXS simulators model the TLB of the R10000, but neither predict the performance degradations due to TLB misses well, as is evidenced in Figure 1. The modeling of the TLB itself is not the issue—the problem is the processor models do not properly model the time associated with a TLB miss. This is typical of many simulators that do, in fact, model a TLB [2]. On the R10000, TLB misses are handled by an exception handler that consists of 14 instructions. However, TLB misses take 65 cycles to execute, even if everything hits in the cache. This strange effect occurs for three reasons: the processor takes a large number of cycles to take and return from an exception, nearly all the instructions are dependent, and there are numerous pipeline-flushing co-processor instructions. The Mipsy processor model takes 25 cycles for these 14 instructions. MXS, which models instruction latency but not the pipeline flushes associated with co-processor instructions, predicts 35 cycles. With hardware results and a microbenchmark that times TLB misses, we were able to tune our simulators to give the correct value of 65 cycles for a

Protocol Case	HW	Tuned FL	Untuned FL
Local, clean	587	615 (1.05)	510 (0.87)
Local, dirty remote	2201	2202 (1.00)	2152 (0.98)
Remote, clean	1484	1457 (0.98)	1311 (0.88)
Remote, dirty home	2359	2378 (1.01)	2215 (0.94)
Remote, dirty remote	2617	2658 (1.02)	2957 (1.13)

Table 3: Dependent load tests on the FLASH hardware and tuned and untuned versions of the FlashLite simulator (times in ns, parenthesized times are relative to the hardware).

TLB miss. Though it is not surprising that processor models that were not designed to model the R10000 do not predict that processor's cost of a TLB miss accurately, this deficiency can cause researchers to overlook the primary bottleneck of a simulated application, even though they believe that they are taking TLB effects into account.

To uncover more sources of error in the processor and memory system, we used the `snbench` [16] microbenchmark tool to measure the memory latency of several protocol cases. One test that `snbench` employs is a string of dependent loads (p = *p) that all miss in the secondary cache. This technique was first introduced in `lmbench` [13]. The performance of these dependent loads is more complicated than we originally thought. On a MIPS R10000, the secondary cache tags and data are off-chip, and the occupancy of the external cache interface is a serious performance concern. The R10000 has the peculiarity that while data is being returned from the memory system and the processor is forwarding this data to the external cache, the external cache interface is occupied for the entire duration of the cacheline transfer. Even subsequent tag checks have to wait for the cacheline transfer to complete (this problem was fixed in the R12000 [26]). Our processor models mispredicted the latency of back-to-back loads because they did not model the occupancy of the secondary cache interface. We added this effect to Mipsy, so that the local read latencies reported by `snbench` on Mipsy would match the ones reported on the hardware. Once local read latencies matched, we easily tuned FlashLite parameters until read latencies for all five protocol read cases shown in Table 3 also matched. Even before tuning, FlashLite latencies already closely matched the hardware. For an architectural simulator, FlashLite is remarkably well-informed as to memory system latency and occupancy effects since it was developed as part of the design of the modeled hardware. Without such a detailed memory system simulator, we suspect that our initial results would have been much worse than those reported in Figure 1.

Our simulator tuning consisted of adjusting the TLB miss time, changing FlashLite bus timing to accommodate Mipsy's new and more accurate secondary cache interface, adjusting the latency through the network router, and tuning the latencies from the network to the node controller and vice-versa. We also used `snbench`'s restart time test, based on Hristea's microbenchmark suite [8], to set Mipsy parameters that determine delays between the processor core and the processor pins.

3.1.3 Results with Tuned Simulator

Figure 3 shows the uniprocessor results with the simulators tuned as we describe above. We expect that since these

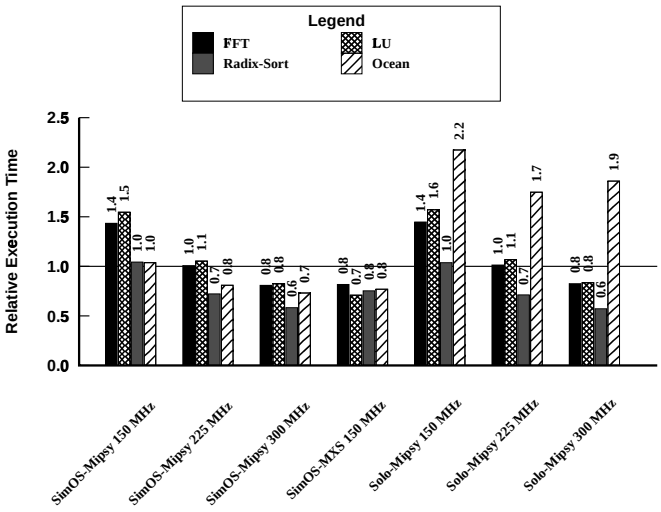

Figure 3: Final uniprocessor SPLASH-2 comparison.

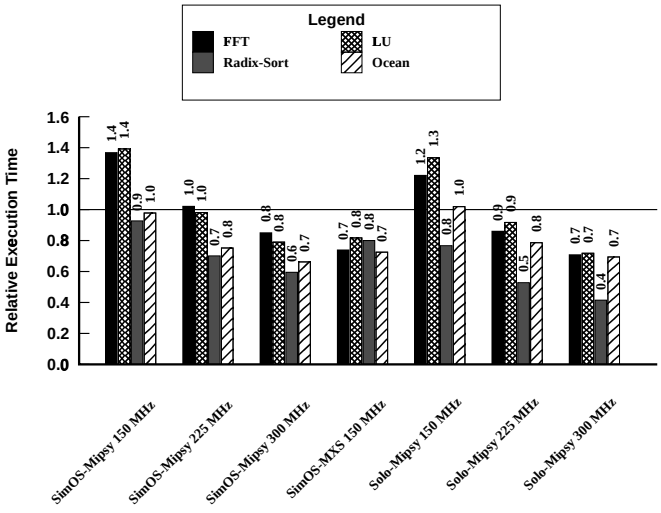

Figure 4: Final 4-processor SPLASH-2 comparison.

applications have a considerable amount of ILP we can get a good estimate of performance from a Mipsy processor running at 225 MHz (corresponding to an IPC of roughly 1.5).

For FFT and LU, we see the SimOS-Mipsy simulator at 225 MHz is almost exact. Since both Radix-Sort and Ocean contain many high-latency instructions, however, we expect that the Mipsy speed that best models the memory request rate will under-predict the execution time. For these applications, SimOS-Mipsy at 225 MHz does predict fast execution times. Even though the 150 MHz SimOS-Mipsy runs are close to the hardware results for Radix-Sort and Ocean, it is just coincidence. The lower clock rate is compensating for the unmodeled instruction latencies. To verify that incorrect instruction latencies are causing the 225 MHz SimOS-Mipsy simulation to underestimate the execution time of Radix-Sort, we ran a simple experiment. Radix-Sort contains many integer multiplications and divisions, so we counted the number of times those two instructions were executed by Mipsy. We find that when we add 5

cycles per multiplication and 19 cycles per division (which are the latencies for those operations on the R10000) to the execution time, the 225 MHz SimOS-Mipsy relative execution time improves from 0.71 to 1.02!

Solo performs well for FFT, Radix-Sort, and LU—the results are nearly identical to SimOS-Mipsy. These applications have been tuned to avoid TLB misses, so there are no relevant operating system effects. Ocean is badly mispredicted by Solo, however, because Solo's page coloring causes conflict misses that do not occur under IRIX.

MXS displays disappointing results, being 20%–30% faster than the real hardware. This indicates that MXS is exploiting far more ILP than the R10000. For FFT and LU, we see that MXS predicts similar times to Mipsy running at twice its speed. While MXS is configured to match the number and latencies of the functional units in the R10000, the fact that MXS is a generic processor model means that it does not handle corner cases the way a real processor would. For example, Ofelt showed that the effects of address interlocks in the R10000 pipeline can in some cases cause a 20%–30% decrease in performance [11]. Unfortunately, a real superscalar processor is an exceedingly complex piece of hardware and these types of unmodeled corner cases abound.

Unlike Mipsy, MXS models instruction latencies, and it is not the fastest (worst) processor model for Radix-Sort and Ocean. However, this mitigating factor is not a vindication of the ILP modeling of MXS as much as it is an artifact of the fact that Mipsy does not model the delays associated with the high-latency instructions present in Radix-Sort and Ocean.

The results for four processors, shown in Figure 4, indicate that the relevant performance effects on the four processor machine are the same as those on a uniprocessor. The only major difference is that the physical memory allocation in Ocean, which was done so poorly by Solo on a uniprocessor, is not a problem for the four-processor Solo runs.

3.2 Speedup Studies

Now that we are familiar with many of the sources of error in our simulators, we evaluate how well the simulators are suited to measuring effects other than absolute performance. We start by examining how well the simulators predict speedup. We find that to first-order, the processor model is usually unimportant for predicting speedup, provided that that the simulator models the operating system and that it has been tuned so that memory request rate matches the hardware. This is in contrast to recent research stressing the importance of detailed processor models [3, 15].

3.2.1 FFT

FFT exhibits near-linear speedup on 16 processors on the FLASH hardware. Figure 5 shows representative speedup curves. All our simulators under-predict speedup. SimOS-MXS and SimOS-Mipsy at both 150 MHz and 225 MHz (not shown) give speedup predictions that are very close to each other, and reasonably close to the hardware. Although the quality of the processor model is important for predicting absolute performance, the inaccuracies tend to offset when the relative metric (in this case, speedup) is a ratio of execution times.

The important result is that the 300 MHz SimOS-Mipsy simulator gives a very misleading prediction of the speedup on 16 processors. This faster processor model makes memory requests more frequently than the MIPS R10000, and it causes contention that is not present in the hardware.

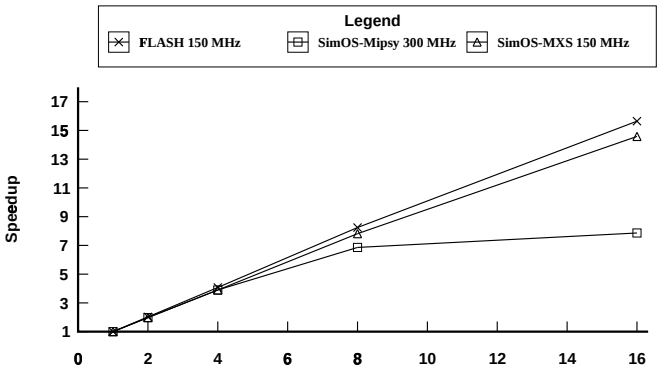

Figure 5: Speedup trend study for FFT.

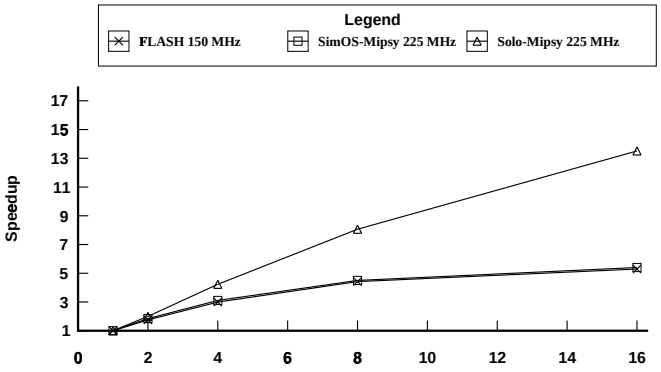

Figure 6: Speedup trend study for Radix.

3.2.2 Radix-Sort

Radix-Sort exhibits a poor speedup of only 5.3 on 16 processors, but we still expect our simulators to predict the performance of the hardware they are modeling. The speedup curves are shown in Figure 6. All SimOS runs, both with Mipsy and with MXS (only SimOS-Mipsy at 225 MHz is shown), accurately predict the poor speedup. Good speedup is incorrectly predicted by Solo because, somewhat surprisingly, Solo does a better job of physical memory allocation than IRIX. Cache conflicts that are present on the hardware and in SimOS are absent in Solo. While it is tempting to assume that this is a problem with the application and not the simulator, we view this as a failure of Solo to correctly predict a performance problem.

3.2.3 Ocean and LU

Speedup studies for both Ocean and LU reveal that all simulators give good speedup predictions, once the results are corrected for the poor performance prediction of Solo for uniprocessor Ocean (shown in Figure 3).

3.3 Effects of the Memory System Model

FlashLite is atypical of many commonly used memory system simulators. Its authors were quite well-informed of every aspect of the hardware design because the simulator's authors were, in fact, the hardware designers. FlashLite even uses delays that were extracted directly from the Ver-

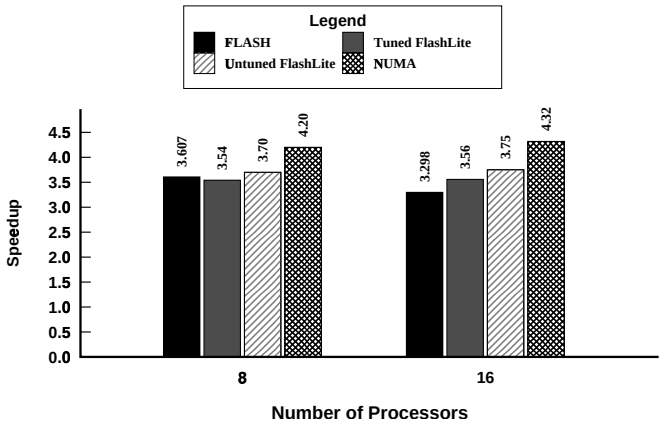

Figure 7: Speedup for unplaced Radix-Sort predicted by SimOS-Mipsy at 225 MHz.

ilog model of the hardware. Most simulators are far less detailed, so we examine how sensitive our results are to the accuracy of the memory system model.

In this section, we use the simulators with the NUMA memory system model in place of FlashLite. The latency parameters in NUMA were set to match hardware latencies, known well in advance of building the hardware. Though it models latencies faithfully, NUMA does not model any extra occupancy of the directory controllers beyond the normal latency path. It also does not model contention in the network or in the routers. NUMA gives similar results to FlashLite for applications that do not experience contention, but we would expect to see large differences for applications that do. To highlight this effect, we study a version of Radix-Sort with data placement disabled. Placing all of the data on a single node will create a hotspot, and we test the simulator's ability to predict the performance impact of the hotspot.

Figure 7 shows the 8 and 16 processor speedup predicted by SimOS-Mipsy at 225 MHz. All SimOS simulators give a good speedup prediction, and we choose this one as being representative. Note that on both 8 and 16 processors, the hardware speedup is poor due to the memory hotspot. The tuned FlashLite simulator does a good job (within 7%) of predicting this performance problem. In addition, the untuned version of FlashLite also predicts speedup well.

The NUMA memory system model will underestimate MAGIC occupancy, so we expect that to find a larger error in an application that is sensitive to occupancy. This is indeed the case. The NUMA model is able to correctly predict that unplaced Radix-Sort will get terrible speedup, but the actual value it predicts is off by 31% for 16 processors.

3.4 Summary

The results show that the major sources of error for our simulators are related to the character of the simulators, and not to our particular implementations. For our trend studies, we find that any simulator that does a reasonable job of modeling the important performance effects of an application will do a reasonable job of predicting trends. The details of the processor model or memory system model, while important for predicting absolute performance, are not critical for trend prediction, provided that the memory request rate is similar to the processor being modeled.

This leads to the surprising result that a simple processor model, Mipsy, run at higher speeds relative to the memory system to model ILP, gives results that are at least as accurate as the more detailed multiple-issue simulator, MXS. Even though MXS was carefully tuned to match the MIPS R10000 instruction latencies and functional unit bandwidth, it does not model the intricacies of the R10000. In our studies, this lack of detail negates the advantages of modeling a multiple-issue, out-of-order processor. MXS takes far longer to run that Mipsy, and since it lacks any clear advantage in the accuracy of the results, we view this as major deficiency in this type of simulator. Our results validate the common practice of using fast, in-order processor to model an out-of-order processor. Researchers still must be careful, however, as running the in-order model too fast can produce poor results, as shown in Figure 5.

Our results seem to paint a rosy picture for simulators in that as long as the important effects are modeled, the details are not crucial. Unfortunately, it is difficult to tell what the important performance effects are until the results are compared with a hardware reference platform. If the simulator does not faithfully model the virtual memory system of the target hardware and operating system, for instance, the results can be seriously distorted by TLB misses and/or page-coloring effects. There is no way to tell if the model is "good enough" unless you compare it with hardware. Even when a simulator does give results that are qualitatively correct, there is still substantial error. For instance, even simulators that we view as predicting speedups well can make predictions that are off by 30% or more—a simulator error that is often larger than the performance gains from architectural enhancements reported in the research literature.

4. CONCLUSIONS

Building FLASH, a machine that has been extensively simulated over the past six years, gave us a unique opportunity to evaluate some of the simulation tools that form a core part of current architecture research. Our results, while humbling, are not that surprising. We find that getting accurate performance data from a simulator is difficult without having a real machine to compare the simulations against. Unfortunately, accurate performance estimation does not seem to be getting easier. Improvement in simulation technology seems to be neutralized by the increasing complexity of the machines we are interested in simulating. While the specific modeling problems we encountered may be less likely to effect future systems, it seems probable that new issues will take their place simply because without a reference platform it is hard to know that these errors exist.

One corollary to the difficulty with producing accurate performance numbers is that a more complex simulator does not ensure better simulation data. We compared a number of processor simulation models and found that MXS, our out-of-order simulation model, was no more accurate than a simple processor model using a simple speedup factor. While MXS was configured to have the same global resource constraints as an R10000, there are a number of "implementation" constraints that are not modeled that inevitably reduce the performance of the processor. The difficulty in modeling these issues meant that the simulator that gives the best results was, in fact, SimOS-Mipsy at 225 MHz, in spite of its simple processor model. For FFT and LU, SimOS-Mipsy at 225 MHz predicts absolute per-

formance within 5% for both one and four-processor runs. However, knowing how much to speed up the simple model depended on having real hardware to compare against or knowing something about the expected ILP in the application.

Our results showing the effectiveness of a simple processor model disagree with recent work by Durbhakula et al. [3] who found that simple simulators like the Solo simulator used in our study badly mispredicted performance. They showed that simple simulators usually predict larger execution times than the detailed out-of-order models; a result that we often saw as well. Our conclusion differs from theirs however, because we found that our detailed out-of-order simulator was substantially faster than the hardware it modeled, and the performance of the real machine (the true gold standard) was predicted at least as well and often better by the simpler simulators.

Given the intrinsic difficulty in producing an accurate simulator, it is comforting to know that many results only weakly depend on the absolute accuracy of the simulator. Even inaccurate performance simulators are useful for gaining intuition about a system and they help find early stage problems. Functionality gaps, deadlock issues, race conditions, and protocol problems are all things that may be discovered in simulators that do not model performance correctly. Even performance-sensitive parameters like parallel processor speedup can be estimated well without needing to tune the simulator, as long as the "critical" performance aspects are modeled correctly. Unfortunately having confidence that these critical issues are modeled correctly is difficult without some validation method.

Simulation is a vital tool for architecture research. A well-designed simulator can provide important insight into the proposed machine. Yet our results show that architectural trend predictions—critical to the evaluation of new architectural ideas—can be erroneous if the underlying simulation lacks an important performance effect. This conclusion demonstrates the importance of building real hardware and should cause some concern not only for computer architects, but for the broader research community for whom simulation is the only method of performance evaluation. In addition to a multitude of other benefits, building hardware allows one to compare simulations to reality, and provides the essential feedback needed to continue to improve simulation technology.

ACKNOWLEDGMENTS

We would like to thank Robert Bosch for his help in configuring the MXS simulator. Robert and Kinshuk Govil also provided valuable assistance in debugging simulator problems. This research was supported by Department of Energy ASCI contract LLL-B341491 and DARPA contract DABT63-94-C-0054. Mark Heinrich is supported in part by NSF CAREER Award CCR-9984314.

REFERENCES

[1] R. Bedichek. Talisman: Fast and Accurate Multicomputer Simulation. Performance Evaluation Review, vol. 23, no. 1, pp. 14-24, May 1995.

[2] D. C. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. *Computer Architecture News*, **25**(3), pages 13–25, June 1997.

[3] M. Durbhakula, V. Pai, and S. Adve. Improving the Speed vs. Accuracy Tradeoff for Simulating Shared-Memory Multiprocessors with ILP Processors. Rice University ECE Technical Report 9802, June 1998.

[4] S. Goldschmidt. Simulation of Multiprocessors: Accuradcy and Performance. Ph.D. Dissertation, Stanford University, June 1993.

[5] M. Heinrich. The Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols. Ph.D. Dissertation, Stanford University, October 1998.

[6] M. Heinrich, J. Kuskin, D. Ofelt, et al. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274–285, October 1994.

[7] M. Heinrich, D. Ofelt, M. Horowitz, and J. Hennessy. Hardware/Software Codesign of the Stanford FLASH Multiprocessor. In *Proceedings of the IEEE Special Issue on Hardware/Software Co-design*, **85**(3), March 1997.

[8] C. Hristea, D. Lenoski, and J. Keen. Measuring Memory Hierarchy Performance of Cache-Coherent Multiprocessors Using Micro Benchmarks. In *Proceedings of Supercomputing 1997*, November 1997.

[9] J. Kuskin, D. Ofelt, M. Heinrich, et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[10] P.S. Magnusson, F. Dahlgren, H. Grahn, et al. SimICS/sun4m: A VirtualWorstation. In *Proceedings of the Usenix Annual Technical Conference*, June 1998.

[11] D. Ofelt. Efficient Performance Prediction for Modern Microprocessors. Ph.D. Dissertation, Stanford University, August 1999.

[12] M. Martonosi, D. Ofelt, and M. Heinrich. Integrating Performance Monitoring and Communication in Parallel Computers. In *1996 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems* , May, 1996.

[13] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. *USENIX technical conference*, pages 279–284, January 1996.

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

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

[16] U. Prestor. Snbench homepage, on-line at `http://www.cs.utah.edu/~uros/snbench`.

[17] S. K. Reinhardt, M. D. Hill, J. R. Larus, et al. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *ACM SIGMETRICS Conference on Measurement & Modeling of Computer*

Systems, May 1993.

[18] M. Rosenblum. Personal Communication.

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

[20] Standard Performance Evaluation Corporation. The SPEC95 Benchmark Suite. Details on-line at `http://www.specbench.org/`.

[21] Stanford Parallel Applications for Shared Memory. SPLASH–2 web page, on-line at `http://www-flash.stanford.edu/apps/SPLASH`.

[22] C. Stolte, R. Bosch, P. Hanrahan, and M. Rosenblum. Visualizing Application Behavior on Superscalar Processors. In *Proceedings of IEEE Information Visualization, 1999*, pages 10-17, 1999.

[23] M. Talluri and M. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 171–182, October 1994.

[24] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the Second International Workshop on Modeling. Analysis, and Simulation of Computer and Telecommunication Systems*, pages 201–207, January 1994.

[25] S. C. Woo, M. Ohara, E. Torrie, et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[26] Kenneth Yeager. Personal Communication.

[27] Kenneth Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, **16**(2):28–40, April 1996.