Precise and Accurate Processor Simulation

Harold W. Cain, Kevin M. Lepak, Brandon A. Schwartz, and Mikko H. Lipasti

Computer Sciences Department
University of Wisconsin
1210 W. Dayton Street
Madison, WI 53706
{cain,baschwar}@cs.wisc.edu

Electrical and Computer Engineering
University of Wisconsin
1415 Engineering Drive
Madison, WI 53706
{lepak,mikko}@ece.wisc.edu

Abstract

Precise and accurate simulation of processors and computer systems is a painstaking, time-consuming, and error-prone task. Abstraction and simplification are powerful tools for reducing the cost and complexity of simulation, but are known to reduce precision. Similarly, limiting and simplifying the workloads that are used to drive simulation can simplify the task of the computer architect, while placing the accuracy of the simulation at risk. Historically, precision has been favored over accuracy, resulting in simulators that are able to analyze minute machine model variations while mispredicting—sometimes dramatically—the actual performance of the processor being simulated. In this paper, we argue that both precise and accurate simulators are needed, and provide evidence that counters conventional wisdom for three factors that affect precision and accuracy in simulation. First, we show that operating system effects are important not just for commercial workloads, but also for SPEC integer benchmarks. Next, we show that simulating incorrect speculative paths is largely unimportant for both commercial workloads and SPEC integer benchmarks. Finally, we argue that correct simulation of I/O behavior, even in uniprocessors, can affect simulator accuracy.

1.0 Introduction and Motivation

For many years, simulation at various levels of abstraction has played a key role in the design of computer systems. There are numerous compelling reasons for implementing simulators, most of them obvious. Design teams need simulators throughout all phases of the design cycle. Initially, during high-level design, simulation is used to narrow the design space and establish credible and feasible alternatives that are likely to meet competitive performance objectives. Later, during microarchitectural definition, a simulator helps guide engineering trade-offs by enabling quantitative comparison of various alternatives. During design implementation, simulators are employed for testing, functional validation, and late-cycle design trade-offs. Finally, simulators provide a useful reference for performance validation once real hardware becomes available.

Outside of the industrial design cycle, simulators are also heavily used in the computer architecture academic research community. Within this context, simulators are primarily used as a vehicle for demonstrating or comparing the utility of new architectural features, compilation techniques, or microarchitectural techniques, rather than for helping to guide an actual design project. As a result, academic simulators are rarely used for functional or performance validation, but strictly for proof of concept, design space exploration, or quantitative trade-off analysis.

Simulation can occur at various levels of abstraction. Some possible approaches and their benefits and drawbacks are summarized in Table 1. For example, flexible and powerful analytical models that exploit queueing theory can be used to examine system-level trade-offs, identify performance bottlenecks, and make coarse performance projections. Alternatively, equations that compute cycles per instruction (CPI) based on cache miss rates and fixed latencies can also be used to estimate performance, though this approach is not effective for systems that are able to mask latency with concurrent activity. These approaches are powerful and widely employed, but will not be considered further in this paper. Instead, we will focus on the last three alternatives: the use of either trace-driven, execution-driven, or full-system simulation to simulate processors and computer systems.

Trace-driven simulation utilizes execution traces collected from real systems. Collection schemes range from software-only schemes that instrument program binaries all the way to proprietary hardware devices that connect to processor debug ports. The former pollute the collected trace, since the software overhead slows program execution relative to I/O devices and other external events. The latter can run at full speed, but require expensive investment in proprietary hardware, knowledge of debug port interfaces, and are probably not feasible for future multi-gigahertz processors. Trace collection is also hampered by the fact that usually only non-speculative or committed instructions are recorded in the trace. Hence, the effects of speculatively executed instructions from incorrect branch paths are lost. Furthermore, once a trace has been col-

Modeling Technique	Inputs	Benefits	Drawbacks
Analytical models	Cache miss rates; I/O rates	Flexible, fast, convenient, provide intuition	Cannot model concurrency; lack of precision
CPI Equations	Core CPI, cache miss rates	Simple, intuitive, reasonably accurate	Cannot model concurrency; lack of precision
Trace-driven Simulation	Hardware traces; software traces	Detailed, precise	Trace collection challenges; lack of speculative effects; implementation complexity
Execution-driven Simulation	Programs, input sets	Detailed, precise, speculative paths	Implementation complexity; simulation time overhead; correctness requirement; lack of OS and system effects
Full-system, execution-driven simulation (PHARMsim)	Operating system, programs, input sets, disk images	Detailed, precise, accurate	Implementation complexity, simulation time overhead, correctness requirement

TABLE 1. Attributes of various performance modeling techniques.

lected, the cost associated with the disk space used to store the trace may be a limiting factor.

Prior work has argued that trace-driven simulation is no longer adequate for simulating modern, out-of-order processors (e.g. [1]). In fact, the vast majority of research papers published today employ execution-driven simulation and utilize relatively detailed and presumably precise simulation. A recent paper argued that precise simulation is very important and can dramatically affect the conclusions one might draw about the relative benefits of specific microarchitectural techniques [6]. Some of these conclusions were toned down in a subsequent publication [5].

One can reasonably conclude that the majority of recently published computer architecture research papers place a great deal of emphasis and effort on precision of simulation, and researchers invest large amounts of time implementing and exercising detailed simulation models.

In this paper, we show that conventional approaches to exercising processor simulators do so poorly vis-a-vis accuracy that, practically speaking, precision is unimportant. We argue that the correct approach is to build a simulator that is both precise and accurate. We accomplish this by building a simulator--PHARMsim--that does not *cheat* with respect to any aspect of simulation. Without the investment in such a simulator, we assert that it is impossible to determine whether or not the right abstractions and simplifications have been applied to either the simulator or the workload that is driving it.

We provide evidence that counters conventional wisdom for three factors that affect precision and accuracy in simulation. First, we show that operating system effects are important not just for commercial workloads (as shown by [8] and numerous others), but also for SPEC integer benchmarks [13]. Surprisingly, omitting the operating system can introduce error that exceeds 100% for benchmarks that, according to conventional wisdom, hardly exercise the operating system. Next, we show that simulating incorrectly predicted speculative paths is largely unimportant for both commercial workloads and SPEC integer benchmarks. In most cases, even with an aggressively speculative processor model that issues twice as many instructions as it retires, the bottom line effect on performance is usually less than 1%, and only 2.4% in the worst case. Finally, we argue that correct simulation of I/O behavior, even in uniprocessors, can affect simulator accuracy. We find that a direct-memory-access (DMA) engine that correctly models the timing of cache line invalidates to lines that are written by I/O devices can affect miss rates by up to 2% and performance by up to 1%, even in a uniprocessor with plenty of available memory bandwidth.

We have found that the main drawback of a simulator that does not cheat is the expense and overhead of correctly implementing such a simulator. For example, since our DMA engine implementation relies on the coherence protocol to operate correctly, the coherence protocol must be correctly implemented. Similarly, since the processors actually read values from the caches, rather than cheating by reading from an artificially-maintained flat memory image, DMA and multiprocessor coherence must be correctly maintained in the caches. Furthermore, within the processor core, the register renaming, branch redirection,

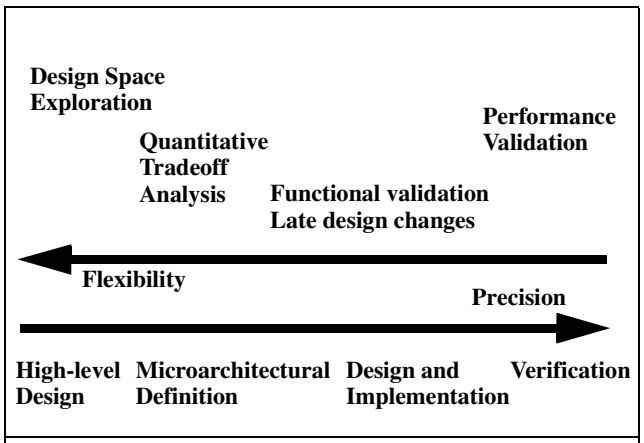

FIGURE 1. Use of Simulation During Design.

store queue forwarding, etc., must all operate correctly for the simulator to follow the correct path. Of course, this drawback is also an advantage: forcing a correctness requirement also forces us, as researchers, to be more thorough and realistic about the techniques that we propose, since we cannot "cheat" when we implement them in our simulator. Counter to our initial expectation, the simulation-time overhead of our simulator is surprisingly low, compared to competitive trace- or execution-driven simulators.

Section 2 provides further discussion on precision and accuracy and how they relate to processor simulation; Section 3 presents details of our PHARMsim simulation environment; Section 4 provides evidence for our three claims concerning operating system effects, speculative wrong-path execution, and DMA implementation; and Section 5 discusses conclusions and implications of our findings.

2.0 Flexibility, Precision, and Accuracy

As discussed already, design teams need simulators throughout all phases of the design cycle. As shown in Figure 1, the *precision* of a simulator tends to increase as the project proceeds from high-level design to later design stages. Here, we define *precision* as a measure of the fidelity of the simulated machine to the actual machine, as the machine is first envisioned and finally realized by its designers. The simulator's precision increases as a natural side effect of the needs of the designers; as the design itself is refined and more precisely defined, making quantitative design trade-offs requires a more precise simulator. Hence, additional features are added to the simulator to model these details. On some development projects, a separate performance simulator effectively disappears, and is replaced by simulation of register-transfer-level models expressed in a hardware-definition language (HDL).

As a consequence of increasing precision, flexibility in turn decreases. By flexibility we mean the ability of the simulator to continue to explore a broad design space. As more and more features are modeled precisely, it becomes increasingly difficult to support design space exploration that strays too far from the chosen direction. This trend also mirrors what is occurring in development; the further the project is from its initial concept, and the closer it is to final realization, the more difficult it becomes to make the major changes required by a broad change in the high-level design.

Besides flexibility and precision, there are several additional important attributes that characterize a simulator or simulation approach. These include simulation speed, functionality, usability, and *accuracy* of simulation. The *accuracy* of a simulator gauges its ability to closely model the real-world behavior of the processor or system being simulated, and manifests itself through simulated performance results that closely match the performance of the real system.

Accuracy is determined by two factors: again, by how closely the model matches the actual design (i.e. precision), but also by how the model is driven: how realistic is the "input" to the model? In general, analytical models and even fixed latency CPI equation models, as presented in Table 1, can be reasonably accurate, but are not very precise. Hence they are commonly used in industry, particularly for performance projections and competitive analysis, as well as early-stage feasibility and proof-of-concept analyses.

In contrast, academic researchers are prone to spend a great deal of time and energy building detailed simulation models that provide lots of precision, so that minute microarchitectural trade-offs can be studied thoroughly and exhaustively. Of course, the level of detail in an academic simulator must match the purpose of the study. For example, high-level limit studies are appropriately conduct with abstract and flexible models. On the other hand, detailed trade-off analyses must be made with fairly precise models. Some academic work exists that attempts to quantify simulator accuracy [2, 6].

Unlike precision, which can be quantified and rectified relatively early in the design cycle, accuracy is much more difficult to measure. Precision can be quantified by exercising both the performance simulator and progressive register-transfer-level realizations of the design with identical test cases, and the cycle-accurate results can be compared and reconciled to correct either the simulator or the design. This is in fact a natural side effect of the performance validation that should occur during a properly managed design cycle.

However, accuracy cannot be so easily determined, since accuracy depends not only on the simulator's precision,

but also on how closely the inputs to the simulation match the real-world environment in which the system being designed will ultimately operate. It is usually considered extremely difficult to recreate these circumstances in such a way that they can be used to drive a detailed performance simulator.

The initial work on full-system simulation from the Stanford SimOS project [10] established that this is indeed possible. However, the complexities of doing so have effectively deterred the majority of the research community from adopting full-system simulation into their repertoire. We emphatically agree with other proponents of full-system simulation and argue that all architecture researchers should seriously consider adoption of full-system simulation, despite the up-front cost of doing so. The evidence in Section 4 strongly supports the assertion that simulators that ignore system effects, no matter how precise, are likely to be so inaccurate as to be useless, even for CPU intensive benchmarks like SPECINT 2000.

In practice, the accuracy of a performance simulator is usually not evaluated until it's "too late," that is to say after hardware is available and stable enough to boot an operating system and run real workloads. At this point, due to several generations of software changes and numerous potentially compensating errors, it becomes very difficult to precisely quantify the accuracy of a performance model. Furthermore, from a practical standpoint, doing so is only useful from an academic and quality assurance viewpoint, and is not driven by immediate design needs. Hence, at least in our experience, such an evaluation is either performed poorly or not at all.

3.0 PharmSim Overview

We have constructed a PowerPC-based simulation infrastructure using the SimOS-PPC and SimpleMP simulators. SimOS is a complete machine simulation environment consisting of simulators for the major components of a computer system (cpus, memory hierarchy, disks, console, ethernet) [10]. We use a version of SimOS which simulates PowerPC-based computer systems running the AIX 4.3 operating system [7]. SimpleMP is a detailed execution-driven multiprocessor simulator that simulates out-of-order processor cores, including branch prediction, speculative execution and a cache coherent memory system[9] using a Sun Gigaplane-XB-like coherence protocol [4]. Integrating the SimpleMP simulator into SimOS required significant changes to SimpleMP in order to accurately support the PowerPC architecture. In this section, we discuss these modifications.

SimpleMP was missing much of the functionality necessary to support system level code, in both the processor core and memory system. We augment SimpleMP with

Processor Parameters	
decode/issue/commit width	8/8/8
RUU/LSQ size	128/64
Functional Units	8 Int ALUs, 3 Int Mult/Div, 3 FP ALUs, 4 FP Mult/Div 3 LD/ST Ports
Branch Predictor	Combined bimodal (8k entry)/ gshare (8k entry) with 8k choice predictor, 8k 4-way SA BTB, 64 entry RAS
Memory System	
L1 I Cache (latency)	64K 2 way set associative (1 cycle)
L1 DCache (latency)	256K 4 way set associative (1 cycle)
L2 Unified Cache	4MB 4 way set associative (10 cycles)
blocksize (all caches)	64 bytes
DRAM latency	70 cycles

TABLE 2. Simulation Parameters.

support for all of the instructions (system-mode and user-mode) in the PowerPC instruction-set architecture. For some of the relatively complex PowerPC instructions (e.g. load/store string instructions) we use an instruction-cracking scheme similar to that used in the POWER4 processor which translates a PowerPC instruction into several simpler RISC-like operations [14]. We also augment the processor core with support for precise interrupt handling and PowerPC context-synchronizing instructions (e.g., isync, rfi).

The SimpleMP memory system required major changes in order to support unaligned memory references, PowerPC address translation, and the set of PowerPC cache management instructions. To handle unaligned memory references (which are allowed in the PowerPC architecture) the processor core splits each unaligned memory reference that crosses a cache block boundary into two smaller aligned references which are then each issued to the SimpleMP memory system.

In order to accurately model PowerPC virtual memory hardware, we were forced to implement a PowerPC memory management unit (MMU) from scratch, including a translation lookaside buffer (TLB), TLB refill mechanism, and reference and change bit setting hardware. On a TLB miss, we simulate a hardware TLB miss handler which walks the page table by issuing memory references to the

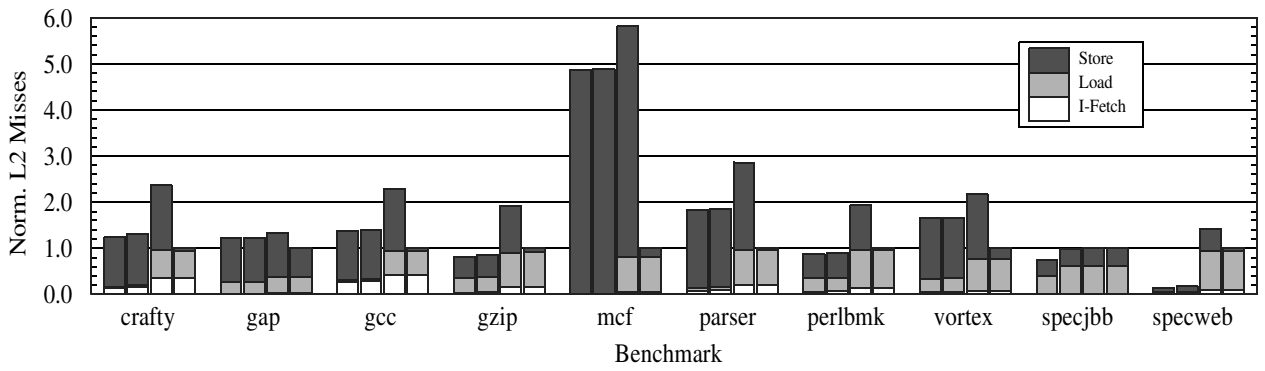

FIGURE 2. Cache effect of simulating the whole system.

The stacked bars show misses to an 8MB, 4-way set-associative cache with 64B lines due to instruction fetches, loads, and stores, normalized to the rightmost case where the whole system is simulated. The leftmost case shows the effects of program references only. The second case adds shared library code, the third case adds operating system code, and the rightmost case adds cache control instructions issued by the operating system. The worst-case error, in MCF, is 5.8x.

simulated memory hierarchy. In the event of a memory management exception (e.g., page fault, protection exception), the MMU signals the processor which traps to the appropriate OS exception handler. The MMU also maintains and updates a page's reference and dirty bits by issuing single-byte stores to the simulated memory hierarchy when a page whose reference or change bit is not set is first referenced or written.

The PowerPC architecture includes many cache management instructions (e.g. data cache block invalidate, data cache block zero, etc.) which are used in both system and user-level code. Implementing each of these instructions required significant changes to the SimpleMP coherence protocol.

We also augment the SimpleMP memory system to support coherent I/O. Both SimpleMP and SimpleScalar [3] perform I/O "magically" by proxying system calls and instantaneously updating a cache's contents to reflect the new memory contents. Obviously, this mechanism does not accurately model how I/O is performed in real systems. To accurately model coherent I/O, we added support to SimOS and SimpleMP for I/O controllers to initiate DMA transfers into memory and invalidate the corresponding blocks in each processor's caches.

For all of the data presented in this paper, we use the machine configuration summarized in Table 2.

4.0 Sources of Inaccuracy

In this section, we study three possible sources of inaccuracy in processor simulation, and quantify their effects on a set of SPECINT2000 and commercial server workloads. The three case studies are the effect of operating system paths, the effect of direct memory access (DMA) transfers caused by disk input/output (I/O), and the effect of speculatively executed incorrect branch paths.

We are able to study these effects in detail only because we have implemented a simulator that does not cheat. The vast majority of prior simulation work either assumes that these effects are insignificant, and fails to consider them, or, assuming the opposite, do implement them but do not quantify the necessity of this implementation overhead.

We present these three case studies to examine if and how inaccuracy is introduced into simulation, and to quantify how relatively important each of these effects is. Current practice in the architecture research community focuses lots of effort on wrong-path execution, and arguably trades off investment in the other two factors to capture wrong-path behavior.

4.1 Operating System Effects

The first effect we study has been examined at length in prior work, particularly for commercial workloads that spend a nontrivial fraction of execution time in the operating system (e.g. [8]). However, conventional wisdom holds that the SPECINT2000 benchmarks spend very little time in the operating system, and can be safely modeled with user-mode instructions only. Our experience with PHARMsim, however, has shown that this is a fallacy. Our detailed simulations have shown that ignoring the effects of operating system instructions can lead to errors of 100% or more when executing SPECINT2000 benchmarks. A detailed analysis of these findings is beyond the scope of this paper, and is left to future work. However, we do present evidence here that helps explain this unanticipated source of error.

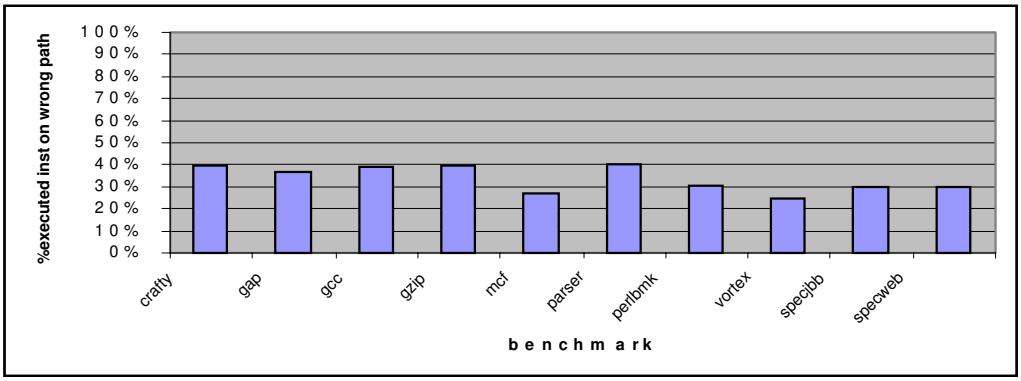

FIGURE 3. % Executed Instructions on Wrong Path.

Figure 2 summarizes off-chip memory traffic for the set of benchmarks we studied, which includes eight of the SPECINT2000 benchmarks and two multiuser commercial workloads, SPECWEB99 and SPECJBB2000 [13]. The stacked bars show misses from an 8MB, 4-way set-associative cache with 64B lines. Miss rates are normalized to the correct case, where all references generated by both the user-mode program, shared libraries, operating system, and special cache control instructions are properly accounted for. The four stacked bars for each benchmark show, from left to right, the effects of the user program only, additional effects from shared library code, additional effects from all of the operating system, and, in the rightmost case, additional effects from special PowerPC cache control instructions issued by the operating system.

These four cases roughly correspond to simulation approaches used in the past: the leftmost case to trace-collection schemes like Atom [12] that instrument and trace user programs only (later versions of Atom handled shared libraries and even parts of the operating system); the second case to a tool like Simplescalar that requires statically-linked objects and then performs system-call translation [3]; the third case to a tracing technique that captures all loads, stores, and instruction fetches by recording off-chip bus signals, but fails to capture explicit cache control references that avoid such signals; and finally, a full-system simulator like PHARMsim that captures and correctly models all references that affect the cache.

In all cases, we see that more references are captured, and additional misses generated, as we proceed from modeling only the user program's references to modeling all of the operating system. In the worst case (*perlbmk*), the number of misses more than doubles, and is significant even in the best case (*specjbb*).

However, interestingly enough, adding the effects of special cache control instructions reverses this direction, and dramatically reduces the number of misses that the cache model encounters. This behavior is caused by the AIX operating system's aggressive use of the PowerPC dcbz (data cache block zero) instruction. This instruction writes an entire cache line with zeroes. Aggressive hardware implements this instruction by avoiding an off-chip memory reference even when it misses the cache, since the whole line will be overwritten by the instruction. This instruction is similar to the wh64 (write hint 64) instruction in the Alpha instruction set [11].

We have analyzed the use of dcbz in the AIX operating system, and have found that the page fault handler issues a series of dcbz instructions that span the entire 4K virtual memory page whenever a program page faults on a new, previously unmapped page. This is a legitimate optimization, since a newly mapped page cannot contain any valid data. Hence, the operating system can safely zero out the page before returning to the user. As a side effect, the program can avoid cache misses to newly referenced pages, since the dcbz instructions directly install those lines into the cache.

The effect of the dcbz instructions is particularly pronounced for the *mcf* benchmark, which allocates and initializes tens of megabytes of heap space to store its internal data structure. Naive simulation, whether of the user program only, or even including operating system effects, can dramatically overstate—to the tune of 5.8x—the number of cache misses encountered by this program. Note that virtually all of the store misses, which dominate the memory traffic of this benchmark, disappear when the dcbz instructions are correctly modeled. There is a similar, though less pronounced, trend for all of the other workloads except *specjbb*. We attribute specjbb's behavior to the fact that we are capturing a snapshot of steady-state execution for this benchmark, rather than end-to-end program execution. As a result, memory has already been allocated and initialized, and the AIX page fault handler does not issue *dcbz* instructions.

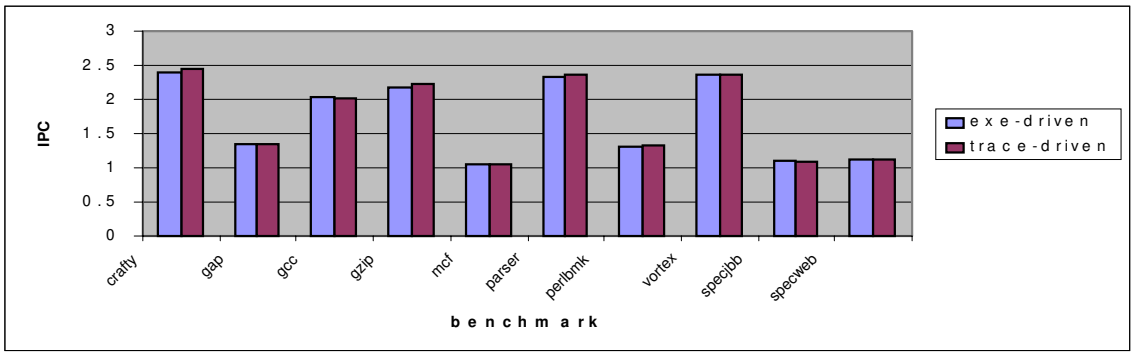

FIGURE 4. Effect of wrong-path instructions on IPC.

We suspect that other operating systems have similar optimizations in their page fault handlers, though we currently have no way of verifying this fact. However, the fact that a detailed, precise, simulator described in a recent study [6] reflected a roughly 20% error on many real benchmarks leads us to believe that such optimizations are widespread and dramatically impact the accuracy of simulation that ignores operating system references.

4.2 Direct Memory Access (DMA)

High-performance systems use some form of autonomous, coherent, I/O agent to perform I/O requests in the system using DMA. This mechanism allows the processor to continue performing other tasks (for example, run another ready process) while the I/O request serviced. Most current simulators either handle I/O through system call proxy or perform the I/O into a flat memory image. Performing flat-memory I/O is functionally correct in simulators which do not include values in cache models, because all values are obtained from the flat memory. However, using this technique the state of cachelines accessed by the I/O agent can be incorrect in one of two ways: 1) In the case of a DMA write (i.e. write to memory, read from I/O device), the line should be marked invalid in the processor's cache; 2) In the case of a DMA read (i.e. read from memory, write to I/O device), the line should become shared.[1]

In execution driven simulators which track memory values throughout the hierarchy (i.e. SimpleMP) I/O causes an additional problem. Since multiple coherence transactions pertaining to regions of memory subject to I/O can be in flight (i.e. a cache block may be in a pending state in multiple caches at the time of an I/O request), the system must be quiesced (i.e. all processor and coherence activity must stop) to effectively flatten the memory image. The I/O can then be performed with any updated values copied (magically, bypassing the performance model) into processors' caches to maintain a consistent view of memory.

1. Provided DMA read requests are handled as regular read requests.

In PHARMsim, in order to determine the effect of cache state errors due to disk I/O activity, as well as system quiescing, we have added an I/O agent (DMA engine) which actually performs the necessary coherence transactions and data transfers between the coherence network/cache model and the disk. This approach avoids the aforementioned inaccuracies and additionally contributes realistic contention on the address and data networks due to disk I/O. Without building this model, we cannot know whether neglecting these I/O effects maintains our stated goals of both simulator accuracy and precision.

We found that for the benchmarks in Figure 2, the I/O effects are small. The only benchmark showing a non-trivial change in cache hit rate is a version of *mcf* in which we have artificially constrained the physical memory size to 64MB to force paging (the L1-I, L1-D, and L2 cache hit rates are reduced by 1.5%.) If we increase the available physical memory to eliminate paging, because AIX implements the *dcbz* optimization mentioned previously (Section 4.1) for newly allocated pages, all I/O coherence events in *mcf* are eliminated. We believe the relative insensitivity to I/O effects occurs due to the effectiveness of disk caches (the benchmarks shown have a paltry number of I/O coherence events compared with other coherence events) and also the nature of the benchmarks--which are not meant to stress I/O performance. We also point out that we expect the execution time difference for single-programmed workloads, provided I/O latency is modelled, to be small given the large disparity between I/O and coherence latency.

One might expect the multi-programmed commercial workloads (*specjbb* and *specweb*) to have required disk activity due to database logging or increased working-set sizes common to commercial applications. However, *specjbb* has no database component and *specweb* has less than 1% of coherence transactions due to I/O in our snapshots, leading to negligible I/O effects.

In order to stress the I/O subsystem, we created our own

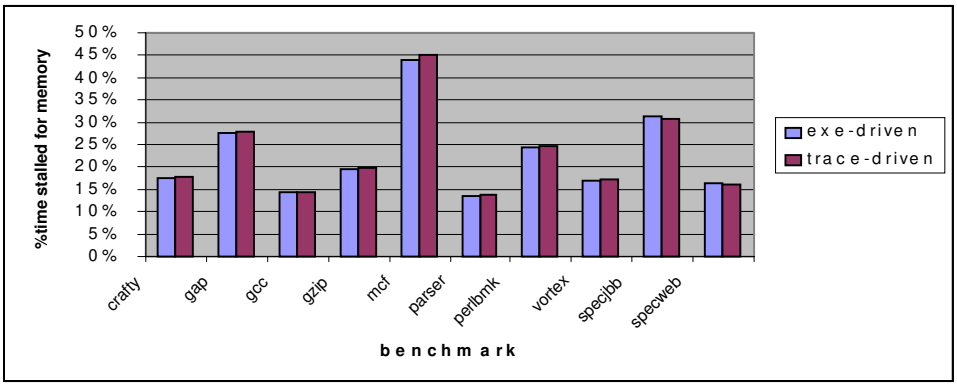

FIGURE 5. Effect of wrong-path instructions on memory stall time.

multi-programmed workload with a combination of file I/O (reading multiple uncached files using the unix 'cat' command) and computation (using gzip of a simulator source code file) and measured the execution time to complete the workload end-to-end. We measured up to a 2.5% reduction in cache hit rate, and a 1.1% increase in execution time when I/O traffic was modelled. We also determined the effect of quiescing the system for I/O related events was less than 0.9% on overall execution time.

Even for this I/O intensive workload (4.8% of coherence traffic due to I/O), the overall effect of I/O on simulation accuracy and precision is small, but measurable, largely due to low contention in the coherence network (over 97% of all coherence transactions occur with fewer than two transactions already outstanding, with 16 possible in our network). However, in larger-scale systems with more processors, we expect greater contention in the coherence and data networks due to three things: 1) Additional demand traffic from other processors; 2) Increased I/O requirements for supplying them; and 3) Difficulty in scaling coherent interconnect in large systems. Evaluating this space is beyond the scope of this work.

4.3 Effect of Wrong-Path Execution

In this section we quantify the impact of ignoring wrong-path instructions on a simulator's results. To perform this comparison, we execute each benchmark on two versions of PHARMSim: the standard execution-driven version, and a modified "trace-based" version which uses a perfect branch predictor to throttle the PHARMSim fetch stage when the machine would ordinarily be fetching an incorrect branch path. Using these two configurations we can evaluate the impact of wrong-path instructions on final performance results.

Figure 3 shows the percentage of instructions reaching the execution stage of the pipeline that are on a mispredicted branch path. For this machine configuration, between 25% and 40% of all instructions executed are on the wrong path. Because this percentage is so high, one would intuitively expect the side-effects of wrong path instructions to have a significant effect on total execution time. However, as can be seen in Figure 4 this is not the case.

Figure 4 shows the difference in IPC as measured by the trace-driven version of PHARMSim and the execution-driven version of PHARMSim for each of the benchmarks. We see that the difference in IPC reported by the two simulators is only 0.97% on average. In the worst case, the benchmark crafty, the difference is only 2.4%. The marginal differences in IPC are the caused by the interaction of wrong-path instructions with non-speculative instructions through the cache hierarchy and branch-predictor.

Wrong-path memory operations may have a positive or negative effect on overall performance. If a wrong path memory operation touches a cache block that the correct execution path will touch in the future, the wrong path instruction may have a prefetching effect, reducing the number of memory related stall cycles for the program. However, if a wrong-path instruction touches a cache block that will not be used in the immediate future, the speculative memory reference may pollute the cache or may compete with subsequent correct-path memory operations for memory system bandwidth. The PHARMSim TLB does not service a TLB miss until the instruction which caused the miss is non-speculative, which in effect filters some of the wrong-path memory references from the cache hierarchy. Figure 5 shows the effects of wrong-path instructions on memory system stall time. For all of the benchmarks except SPECjbb and SPECweb, the inclusion of wrong-path instructions has a positive effect on memory stall time. In all cases, however, the effect is very small. On average, there is only a 0.3% difference between the total number of memory stall cycles when executing with and without wrong-path instructions.

The execution of wrong-path instructions may also affect

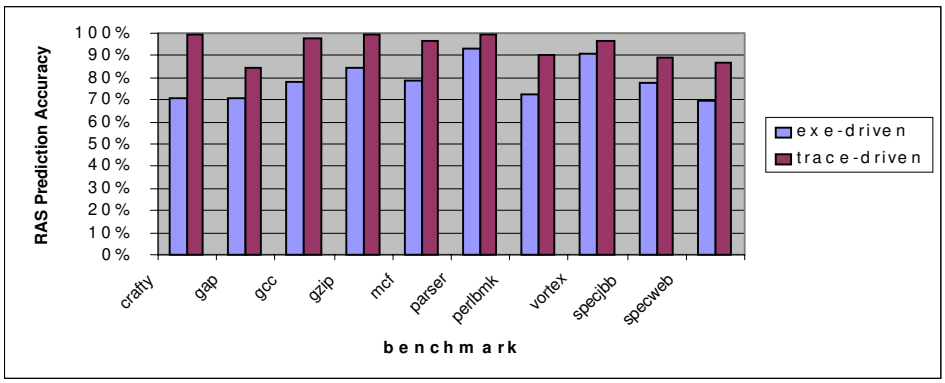

FIGURE 6. Effect of wrong-path instructions on return-address stack accuracy.

branch predictor performance. The branch predictor used for the results in this section is a combining predictor whose pattern history tables are not updated speculatively, and whose branch history register is checkpointed before each branch and restored in the event of a branch misprediction. Consequently, the main branch predictor is not polluted by wrong-path instructions and suffers no performance degradation. However, the 64-entry return address stack is updated speculatively and is not recovered in the event of a branch misprediction. As shown in Figure 6 the pollution of the return address stack by wrong path instructions significantly affects its performance. On average, the accuracy of the return address stack decreases by 15%. In the worst case (crafty) accuracy is reduced by 29%. Despite this RAS performance loss, overall performance results are not barely affected, as shown in Figure 4.

Overall, ignoring the effects of wrong-path execution has almost no impact on performance. Consequently, we believe trace-driven simulation without wrong-path instructions is a valid method for estimating uniprocessor performance. Execution-driven uniprocessor simulators may also ignore wrong path instructions to improve simulation efficiency. Although these results validate the use of trace-driven simulation to evaluate single-threaded workloads, the performance of multi-threaded workloads which include communication among threads should not be evaluated using trace-driven simulators for reasons which are beyond the scope of this paper.

5.0 Conclusions

This paper studies three factors that can affect the accuracy of uniprocessor simulation: operating system effects, direct memory access by I/O devices, and wrong-path speculative execution. Using PHARMsim, a detailed full-system simulator that does not cheat, we are able to show that operating system references should be fully modeled, even for benchmarks like SPECINT2000 that have historically been considered safe for user-mode-only simulation. In the case of the AIX operating system, this is due to optimizations in the page fault handler that employ explicit cache control instructions to avoid unnecessary cache misses. Further, we find that correct modeling of DMA traffic can have a nontrivial effect on performance, and should be accounted for in workloads that perform a significant amount of I/O. Finally, we show that wrong-path speculative execution has a nearly indiscernible effect on overall performance. Though individual microarchitectural structures like the return address stack can be negatively affected by these paths, the overall contribution of these effects is so minimal that ignoring speculative paths is safe for the workloads we study.

Our study is far from complete, as there are numerous other effects we are currently studying and plan to report on in the future. These include the effects of speculative and non-speculative TLB refills, more detailed analysis of branch predictor updates, evaluation of DMA transfers directly into the cache hierarchy, etc. However, we do make the following conclusions and suggestions based on the evidence presented herein:

- All detailed processor simulations, even if only running SPECINT-like benchmarks, should fully account for operating system references. The extreme errors that are introduced if these effects are not modeled make any such simulations so inaccurate as to be meaningless. This conclusion should have a significant impact on the research community and the peer review process.
- Trace-based simulation, which is still widely used in industry, should not be dismissed in favor of execution-driven simulation. Our evidence suggests that traces that include operating system references but omit wrong-path speculative references are far more useful than execution-driven simulation of user-mode programs.

- Though demonstrably more accurate than execution-driven simulation, trace-based simulations may still have shortcomings that make execution-driven simulation attractive. For example, most hardware tracing schemes are incapable of capturing register or memory values. Hence, study of techniques that exploit value locality is not possible with such traces.
- Workloads that perform a nontrivial amount of I/O should be simulated in a way that properly accounts for the additional memory traffic induced by DMA transfers. Without such an accounting, simulation results may not be acceptably accurate. On the other hand, workloads like SPECINT2000, with minimal I/O, can safely be simulated without accurate modeling of DMA effects.

Finally, we want to point out that lack of proper simulation infrastructure should not serve as a valid excuse for avoiding both precise and accurate processor simulation, and the research community as a whole needs to accept this fact. Our research group has made a significant investment in simulation infrastructure that also builds heavily on work done by others. The fact that we have been able to develop this infrastructure serves as an existence proof that it is possible, even with the limited means available within academia.

6.0 Acknowledgments

Many individuals have contributed to the work described in this paper. Among them are current and former members of the PHARM research group at the University of Wisconsin, Ravi Rajwar who wrote the SimpleMP simulator, Pat Bohrer and others at IBM Research who ported SimOS to the PowerPC architecture, as well as the original authors of the SimOS toolset at Stanford. We are heavily indebted to all of these individuals. This work was also supported by donations from IBM and Intel and NSF grants CCR-0073440, CCR-0083126, and EIA-0103670.

References

[1] Bryan Black, Andrew S Huang, Mikko H. Lipasti, and John P. Shen. Can trace-driven simulators accurately predict superscalar performance? In *Proceedings of the 1996 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '96)*, October 1996.

[2] Bryan Black and John Paul Shen. Calibration of microprocessor performance models. *Computer*, 31(5):59–65, May 1998.

[3] D.C. Burger and T.M. Austin. The simplescalar tool set, version 2.0. Technical report, University of Wisconsin Computre Sciences, 1997.

[4] Allan Charlesworth, A. Phelps, R. Williams, and G. Gilbert. Gigaplane-XB: Extending the ultra enterprise family. In *Proceedings of the International Symposium on High Performance Interconnects V*, August 1997.

[5] R. Desikan, D. Burger, S.W. Keckler, L. Cruz, F. Latorre, A. Gonzalez, and M. Valero. Errata on measuring experimental error in microprocessor simulation. Computer Architecture News, March 2002.

[6] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA-01)*, June 2001.

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

[8] Ann Marie Grizzaffi Maynard, Colette M. Donnelly, and Bret R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. *ACM SIG-PLAN Notices*, 29(11):145–156, November 1994.

[9] Ravi Rajwar and Jim Goodman. Simplemp multiprocessor simulator. Personal communication., 2000.

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

[11] Richard L. Sites. *Alpha Architecture Reference Manual*. Digital Press, Maynard, MA, 1992.

[12] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

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

[14] Joel M. Tendler, S. Dodson, S. Fields, and B. Sinharoy. IBM eserver POWER4 system microarchitecture. IBM Whitepaper, October 2001.