Performance Validation of Network-Intensive Workloads on a Full-System Simulator

Ali G. Saidi Nathan L. Binkert Lisa R. Hsu
Steven K. Reinhardt
{saidi,binkertn,hsul,stever}@umich.edu

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122

Abstract

Performance accuracy is a critical but often neglected aspect of architectural performance simulators. One approach to evaluating performance accuracy is to attempt to reproduce observed performance results from a real machine. In this paper, we attempt to model the performance of a Compaq Alpha XP1000 workstation using the M5 full-system simulator. There are two novel aspects to this work. First, we simulate complex TCP/IP networking workloads and use network bandwidth as our primary performance metric. Unlike conventional CPU-intensive applications, these workloads spend most of their time in the operating system kernel and include significant interactions with platform hardware such as the interrupt controller and network interface device. Second, we attempt to achieve performance accuracy without extremely precise modeling of the reference hardware. Instead, we use simple generic component models and tune them to achieve appropriate bandwidths and latencies.

Overall, we were able to achieve reasonable accuracy even with our relatively imprecise model, matching the bandwidth of the real system within 15% in most cases. We also used profiling to break CPU time down into categories, and found that the simulation results correlated well with the real machine.

1 Introduction

The computer architecture community makes wide use of simulation to evaluate new ideas. To provide meaningful results, execution-driven architectural simulators must both be functionally correct and model performance accurately. Functional correctness, though often challenging, is typically straightforward to test. In many cases, a lack of functional correctness has catastrophic consequences that cannot be ignored. Performance accuracy, on the other hand, is much harder to verify and much easier to neglect. As a result, it generally gets short shrift from the research community. This situation is ironic: given that the primary output of these simulators is performance data rather than simulated program output, performance accuracy is at least as important as functional correctness, if not more so.

One of the key obstacles to validating performance accuracy is that accuracy measurement requires a reference implementation with which the simulator's performance results can be compared. Because the primary purpose of simulation is to model designs that have not been (and may never be) implemented, this reference usually does not exist. Some of the few simulator validation studies in the literature came from situations where the simulator was used in the design of a system that was afterwards implemented in hardware, at which point the designers could go back and retroactively measure the accuracy of their simulators [1, 2]. While valuable for the insights provided, these studies do not provide validation before design decisions are made, when it is needed most.

Another approach to performance validation is to configure a parameterizable simulator to model an existing system and evaluate its accuracy in doing so [3, 4]. Although this process does not fully validate the simulator's accuracy in modeling all the various configurations of interest, it identifies common-mode errors and provides a much higher degree of confidence than having no validation whatsoever. Unfortunately, modern computer systems are extremely

complex, and modeling the myriad subtleties of a particular hardware implementation is a painstaking effort. Capturing these subtleties may be required to correlate the simulator's absolute performance with that of the actual reference hardware. However, to the extent that these details are orthogonal to the architectural features under study, more approximate models would suffice to provide accurate relative performance measurements. Developing performance models that incorporate numerous potentially irrelevant details may not be the most productive use of a researcher's time.

The difficulty of performance validation increases as researchers attempt to investigate more complex applications. For example, Desikan *et al.* [4] were able to model the performance of CPU-bound microbenchmarks on an Alpha 21264 with an accuracy of 2%, but complexities in the memory system (including the impact of virtual-to-physical page mappings and refresh timing) caused their modeling error to average 18% on the SPEC CPU2000 macrobenchmarks.

This paper describes our experience in validating the performance accuracy of the M5 full-system simulator for TCP/IP-based network-intensive workloads. Our efforts differ from previous work in this area in two key aspects. First, we use complex workloads that are both OS- and I/O-intensive. In contrast, earlier simulation validation studies used application-only simulation [3, 4] or used workloads that did not stress the OS or perform significant I/O [1, 2]. Second, while we adjust our simulator configuration to model our reference machine (a Compaq Alpha XP1000), we do not strive to model that system precisely. One of our goals is to determine how accurately we can model the overall behavior of a complex system using appropriately tuned but relatively generic component models. Avoiding irrelevant modeling details both saves time and increases our confidence that our set of model parameters captures the most important behavioral characteristics.

We use two Alpha 21264-based Compaq XP1000 systems as our reference platforms, with CPUs running at 500 and 667 MHz. We evaluate both the absolute and relative performance accuracy of our simulator modeling these systems, using network bandwidth as our primary metric. After correcting several inaccuracies in our model, we have narrowed the discrepancy between it and the actual hardware to less than 15% in most cases. We also evaluate simulations on their ability to model the relative time spent in different portions of the software (application, protocol stack, device driver, etc.). We see that the simulated and actual CPU utilization numbers are strongly correlated.

In addition to the validation, we perform a sensitivity analysis on our final simulator to determine which parameters have the largest impact on simulated system behavior. Identifying components and parameters that do not contribute significantly to overall accuracy is important, as we can save development time by not writing or validating detailed models for these components, and save simulation time by not executing those detailed models. We find that most of the parameters we modified had little effect on the bandwidth of the simulation.

The remainder of this paper begins with a discussion of our M5 simulator in Section 2. We then discuss the benchmarks we used in Section 3 and how we gathered the data in Section 4. The results we ultimately achieved are presented in Section 5, and we discuss the sensitivity of the results in Section 6. Section 7 presents related work, and we conclude in Section 8.

2 The M5 Simulator

The primary goal of the M5 simulator is to enable research in end-system architectures for high-bandwidth TCP/IP networking. TCP/IP network protocol and device interface code resides in the operating system kernel, so TCP-intensive workloads spend much of their time there. As a result, full-system simulation is a necessity. This research requires additional capabilities beyond previously existing full-system simulators [5, 6, 7], such as a detailed and accurate model of the I/O subsystem, including the network interface controller (NIC), and the ability to simulate multiple networked systems in a controlled and timing-accurate fashion. The difficulty of adapting an existing full-system simulator to meet our needs seemed larger than that of developing our own system, so we embarked on the development of M5.

A key goal of M5 is modularity. All simulated components are encapsulated as C++ objects with common interfaces for instantiation, configuration, and checkpointing. The following subsections describe the categories of component models most relevant to this study: processors, memory and I/O systems, and the network interface.

2.1 Processor Models

M5 includes two processor models used in this study: a simple functional model used primarily for fast-forwarding and cache warmup, and a detailed out-of-order model used for collecting performance mea-

surements. (The latter was originally derived from Simplescalar's `sim-outorder` [8], but has been almost completely rewritten.) In addition to standard out-of-order execution modeling, the detailed processor model includes the timing impact of memory barriers, write barriers, and uncached memory accesses.

These processor models functionally emulate an Alpha 21164 (EV5). The simulator executes actual Alpha PALcode to handle booting, interrupts, and traps. The 21164 was originally chosen because it allowed us to use SimOS/Alpha [5] as a reference platform during initial development of our full-system support. Although we implement 21164 control registers and execute 21164 PALcode, both the OS and application code see the processor as an Alpha 21264, with all associated 21264 ISA extensions [9].

To boot Linux, M5 functionally models a Compaq Tsunami platform [10], of which the XP1000 is an example. Our implementation includes the chipset and corresponding devices such as a real-time clock and serial ports. These devices are emulated with enough fidelity to boot an unmodified Linux 2.4 or 2.6 series kernel.

2.2 Memory and I/O System

The busses and chips connecting the CPU, memory, and I/O devices of the system are a key factor in the performance of a system. We build the memory and I/O system out of memory and device models interconnected by busses and bus bridges.

The bus model is a simple split-transaction broadcast channel of configurable width and frequency. When used as a point-to-point interconnect, as in some places in our Tsunami model, performance is optimistic, as it provides full bandwidth at half duplex (i.e., in each direction, though not concurrently), while the real link provides half the total bandwidth but in full duplex.

The bus bridge model joins two busses and has configurable delay and buffering capacity. Bridges are used to connect busses with different bandwidths and to model the latency of chip crossings. These bridges also have the capability to acknowledge writes back to the requester before passing them on, which is needed to accurately model Compaq Tsunami timing (see Section 6.1).

Using these simple components, we can assemble an entire memory hierarchy from a CPU through multiple levels of cache to DRAM or I/O devices as desired. I/O DMA transactions are kept coherent by having the cache hierarchy snoop them. DMA reads get modified data from the cache if present, and DMA writes invalidate any cached copies. Since we are modeling a uniprocessor system, a more complex coherence scheme is not required.

2.3 Ethernet Interface

We chose to model a National Semiconductor DP83820 network interface controller (NIC), as it is the only commercial gigabit Ethernet interface for which we could find publicly available documentation. This card supports full-duplex transmission at 1Gb/s and comprises two separate units, the MAC and the PHY.

The Bus Interface Unit/Media Access Controller (MAC) connects to the PCI Bus on one side and to the physical layer interface on the other. This unit holds configuration and status registers which can be accessed by the device driver through programmed I/O, as well as a DMA engine to transfer packets to/from main memory. It participates in the memory model as a first class device, arbitrating for the PCI bus and transferring data like any other device in the memory hierarchy.

The NIC's physical interface (PHY) provides a connection between the MAC FIFOs and the physical link, modulating and demodulating signals on the wire. Since this component contains no buffering and represents minimal delay, we do not model it explicitly.

The Ethernet link we model is a lossless full-duplex link of configurable bandwidth and delay. The time on the link is calculated by dividing the packet size by the link bandwidth. If the result is less than the wire delay, only a single packet is allowed on link at a time. If the delay is large, then it is possible for the link to have multiple packets in flight. If insufficient buffer space is available at the receiver when a packet exits the link, the packet is dropped.

The DP83820 was used in many actual gigabit Ethernet cards including the NetGear GA622T, D-Link DGE500T, and SMC 9462TX. We used a NetGear GA622T for our tests. The DP83820 has a bug which requires it to DMA to a 8-byte aligned address. A NIC can normally DMA to an arbitrarily aligned address so that the payload within the Ethernet packet can be aligned at the expense of unaligning the Ethernet header. We have fixed this bug in our simulated model. However, to be true to the actual card, we removed that fix for this paper. As a result, we take many unaligned access traps on both the real and simulated Alphas.

3 Benchmarks

We used several workloads to compare the performance of our simulated systems and the two Alpha XP1000s. Two memory microbenchmarks, the `mem_lat_rd` utility from LMBench [11] and a custom Linux kernel module, provided detailed memory timing information. To measure networking performance, we used the netperf microbenchmark [12] and a modified version of SPEC WEB99 [13].

3.1 Memory Microbenchmarks

We used two different microbenchmarks to calibrate memory and I/O device delays in our simulator. The first is a small custom Linux kernel module that calculates the latency to a given uncachable memory location. It accomplishes this by using the Alpha `rpcc` instruction to time the execution of loads and stores to the address in question. With this kernel module on a real machine we were able to discern the read and write latency to chipset registers and I/O device registers. We used these measured delays to calibrate the bus-bridge timing in our simulator appropriately.

The `mem_lat_rd` benchmark from LMBench helped us calibrate and verify the DRAM and cache timing parameters used in the system. This tool allows the user to select a stride and a region size of memory for testing. The benchmark then builds a linked list in the memory region with each memory location holding a pointer to a location a stride away. The pointer dependence forces each load to complete before the next load can issue. By adjusting the stride it is possible to cause a variety behaviors in the cache hierarchy and thus determine timing information for accesses to different levels of cache and for events like TLB misses.

3.2 Netperf

Netperf is a collection of network microbenchmarks developed by Hewlett-Packard, including benchmarks for evaluating the bandwidth and latency characteristics of various network protocols. Of the various benchmarks, we selected TCP stream, a transmit benchmark, and TCP maerts, a receive benchmark. In both of these benchmarks, the client informs the server of the benchmark and the server acts either as a sink (for the transmit benchmark) or as a source (for the receive benchmark), consuming or producing data as fast as it can. These benchmarks are simple, just filling a buffer and calling `send()` or `receive()`. Thus they spend most of their time in the kernel TCP/IP stack and interrupt handler, and very little time in user mode.

3.3 SPEC WEB99

SPEC WEB99 is a popular benchmark for evaluating webserver performance. It simulates multiple users accessing a mix of both static and dynamic content over HTTP 1.1 connections. The benchmark includes CGI scripts to do dynamic ad rotation and other services a production webserver would normally handle. For our simulations, we used the Apache webserver [14] version 2.0.52. We also used the Apache mod_specweb99 module, which is available on the SPEC and Apache websites. This module replaces the generic reference CGI scripts with a more optimized C implementation.

We also wrote our own client request generator that is more appropriate for a simulation environment while preserving the workload characteristics of the benchmark. The standard SPEC WEB99 score is the maximum number of simultaneous clients a server can support at a minimum bandwidth per connection. Each client requests a reasonably small amount of data at a maximum rate of 400kbps and expects that request to be serviced within a certain time. The server's score is normally determined by configuring a very large testbed of client machines and iteratively tuning the number of connections and various server parameters. Given the enormous slowdown incurred by simulation, this iterative approach is clearly impractical for our purposes. Since we are interested in the performance characteristics of the benchmark and not the actual score achieved, we chose to modify the client to enable a single client to scale up its performance to saturate the web server, thus avoiding the iterative tuning step. Instead of modifying the stock SPEC WEB99 client, we created our own lightweight client based on the Surge [15] traffic generator, but using the statistical distribution of requests from the stock SPEC WEB99 client.

4 Methodology

In this section we describe the metrics we used to compare our simulated results to the real hardware and the methodology we chose to gather data.

4.1 Metrics

We used two metrics to compare our simulated and real systems: bandwidth and CPU utilization. Our primary comparison focused on the bandwidth achieved by the server, since that is of greatest concern for network-oriented benchmarks. Our sec-

Category	Description
Alignment	Time spent processing reads or writes to unaligned addresses.
Buffer	Time spent dealing with buffer management issues.
Copy	Time spent copying packets around in the kernel and to/from user space.
Driver	Time spent executing code from the network driver.
Idle	Time spent with the CPU being idle.
Interrupt	Time spent handling I/O and timer interrupts (not including device driver code).
Other	Time spent in the kernel that doesn't fall into any of the other categories.
Stack	Time spent processing packets in the TCP/IP protocol stack.
User	Time spent executing a user level process.

Table 1: Description of CPU Utilization Categories.

ondary metric was CPU utilization. We broke CPU time down into several categories (Idle, Other, User, Copy, Buffer, Stack, Driver, Interrupt, and Alignment) and compared the relative amount of time spent in each category across the simulated and actual machines. See Table 1 for a description of the categories.

4.2 Simulated System

Full-system cycle-level simulation is orders of magnitude slower than real hardware, making it impractical to run benchmarks to completion. To cope with this limitation, we turned to fast-forwarding, cache warmup, and sampling to reduce the simulation time while maintaining the characteristics of the benchmark. These techniques, however, can not be applied blindly as the TCP protocol is self-tuning. Sampling too soon after switching from a function to a detailed model can produce incorrect results [16].

To apply fast-forwarding we functionally executed the code with a perfect memory system. This allowed us to quickly boot the system and reach an interesting point of execution. After we reached a point of interest we switched to a detailed CPU with a memory hierarchy. To make sure that we gave the TCP stack ample time to tune itself to the new system configuration it was operating on we waited for 1.4 seconds of simulated time to elapse before gathering data. This delay also gives the caches and TLB time to warm up. At 1.4 seconds we then sampled every 100ms until the simulation reached 2.5 seconds and terminated.

The system that we are interested in varies depending on the benchmark; the system under test is the client for netperf and the server for SPEC WEB99. To ensure that the system under test can perform to its fullest the other machine is simulated with a perfect memory system, making it artificially fast.

We sampled the program counter every 100 cycles to measure the amount of time the CPU spent executing different categories of code. With the use of the kernel symbol table we found the symbol closest to that address and kept track of how many times each symbol was seen during the execution. In a post processing step these function names were aggregated into the broad categories presented above.

4.3 Real System

Our testbed consisted of one Compaq Alpha XP1000 running with either a 500MHz EV6 or 667MHz EV67 processor and a National Semiconductor NS82830 based Gigabit Ethernet card. To stress the Alpha we used a dual-processor Opteron with a Tigon III Ethernet card. For the real system we had the luxury of running benchmarks to completion at the price of the data gathering slightly perturbing the results. To reduce this interference as much as possible we ran for hundreds of seconds sampling the transmit and receive byte counts on the Opteron at 30 second intervals and using OProfile[17] sampling every 100,000 cycles to obtain the Alpha's CPU utilization breakdowns.

5 Comparison to M5

In this section, we compare our measurements of two real Alpha XP1000 systems to the simulator's results for similarly configured systems. The first section describes our results for the memory microbenchmarks we used to calibrate M5's modeled memory latencies. The second section discusses the network benchmarks that are of primary interest.

5.1 Memory Latency

Figures 1 through 3 show the results of our simulated runs compared to the real hardware.

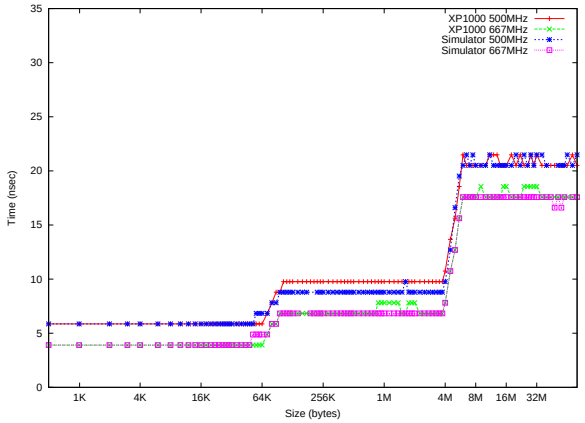

Figure 1: mem_lat_read: Memory Latency with Stride 8

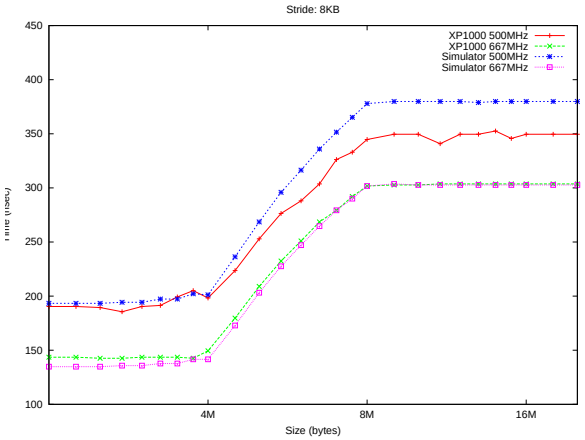

Figure 3: mem_lat_read: Memory Latency with Stride 8k

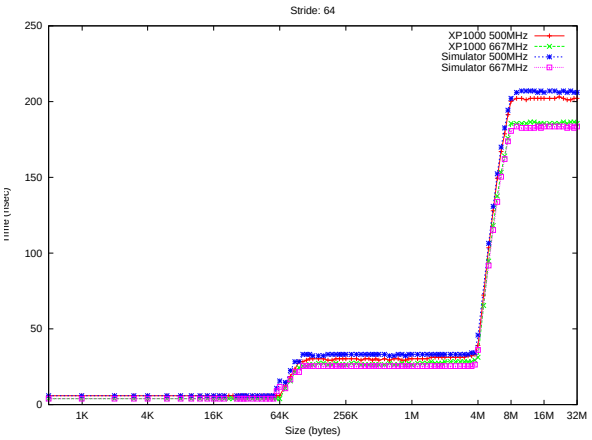

Figure 2: mem_lat_read: Memory Latency with Stride 64

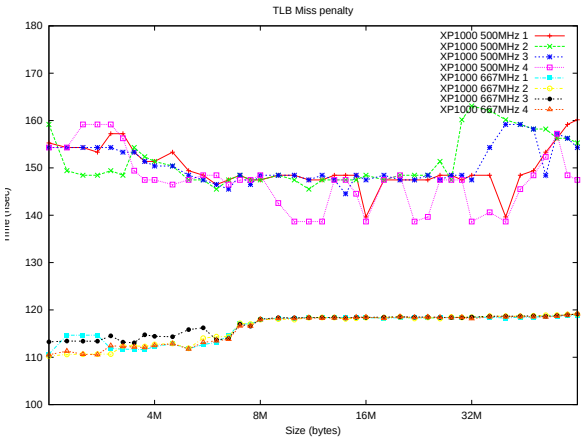

Figure 4: mem_lat_read: Memory Latency Stability with Stride 8k

In Figure 1 we configured `mem_lat_rd` to read every word sequentially. This setup incurs an L1 cache miss every 8 accesses. For sizes over 4MB, the L1 misses also miss in the L2. We tuned our memory system latencies so that our data matches the real data rather closely. The little bump around 64kB is due to page mapping issues. The Alpha has a virtually indexed physically tagged L1, while M5 currently supports only physically indexed caches. Thus pages within a 64KB array could conflict in our L1 where they cannot on the real machine. The other discrepancies observed for the 500MHz system are within the timer resolution latency.

Figure 2 shows results for a 64-byte stride, and is mainly concerned with L2 hit time. For array sizes larger than the L1 cache, all accesses are L1 misses. Again we have successfully tuned M5 to model the real hardware latencies. The simulated main memory access times seen at the right end of the graph are within 4ns of the measured times.

Figure 3 focuses on TLB miss latency by showing results for a stride of 8kB. Below 4MB, each access is a TLB miss but an L2 hit. At the right end of the graph, accesses miss in both the TLB and the L2 cache. We were able to model the TLB miss latency precisely for both situations on the 667MHz machine. However, the 500MHz machine posed more of a challenge. Using the same TLB miss overhead (in CPU cycles) as in the 667MHz machine, we were able to correctly model the cost of a TLB miss that hits in

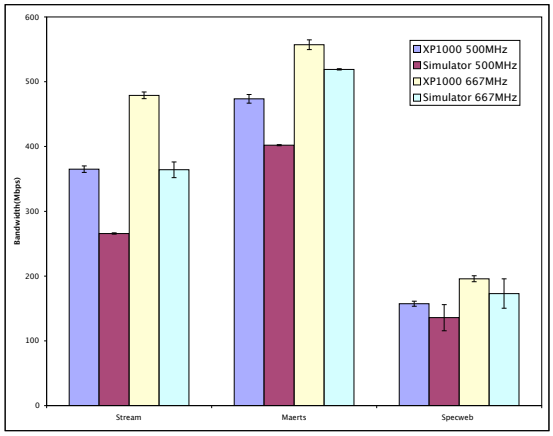

Figure 5: Bandwidth(Mbps) for benchmarks

the L2 cache. However, the latency for a combined TLB/L2 miss on the 500MHz system takes approximately 30ns longer on the simulator than on the real machine, in spite of having an accurate latency for main memory accesses on this system (see Figure 2). We are unable to explain this phenomenon. We also noticed that the TLB miss penalty is much more stable in the 667MHz system than in the 500MHz system. Figure 4 illustrates this effect by plotting just the TLB miss penalty (subtracting out the memory access latency) for multiple runs of the microbenchmark. The reason for this difference in stability also eludes us.

5.2 Network Benchmarks

Figure 5 shows the final bandwidth results we produced for the networking workloads. Error bars indicate one standard deviation of error across the measurement samples we took. Although the simulator consistently underestimates the network bandwidth, the results are reasonably close and stable.

For the stream benchmark we see an absolute error of 37% and 32% for the 500MHz and 667MHz machines respectively. We achieve errors of 18% and 7% for maerts and 16% and 13% for SPEC WEB99. Though the simulated systems always underestimate the absolute bandwidth, the relative change in bandwidth due to varying the CPU frequency is modeled accurately.

We believe that the consistently lower performance of the simulated systems is related to the fact that they transmit half as many packets per transmit interrupt as the real hardware does. The overhead of twice as many interrupt-handler invocations per transmitted packet could account for the reduced bandwidth. Our hypothesis is supported by the fact that the stream benchmark exhibits the largest error; being a transmit-oriented benchmark, it would suffer the most from this effect. We do not currently understand the cause of this interrupt-rate differential.

In Figure 6 we compare the fractional CPU utilization broken down into the above mentioned categories. As mentioned above, the data was gathered from OProfile on the real hardware and using high-frequency sampling (every 100 cycles) on the simulator. Note that the simulator spends almost twice as much time in interrupt code as the real system. This result is consistent with the simulated system taking more transmit interrupts, but is possibly due to OProfile not being able to interrupt the CPU for a monitoring event if the CPU is at a higher interrupt priority level. Unlike OProfile, our simulator's sampling is not affected by the CPU's interrupt priority level.

The other two discrepancies seen in the utilization graph are a difference in time spent in TCP/IP stack processing and the lack of idle time we see in the simulated stream runs. We hypothesize that this is due to our CPU model underestimating some aspects of the real CPU's performance. This effect is not surprising, as we spent most of our tuning effort on the memory and I/O system.

6 Sensitivity of Results

During the process of running experiments we found and remedied a number of modeling errors in our simulator. In this section, we describe these errors and present an analysis of the impact they each have on the final results. Other than the TLB miss penalty, the impact of most of the changes presented in this section are largely within the sampling noise; the addition or removal of a feature generally yielded an average change of only a few percentage points. Nevertheless, some of these parameters were important in improving correlation of the utilization breakdown seen above.

6.1 Acknowledging Writes

While analyzing a previous version of the CPU utilization numbers mentioned in Section 5, we noticed that the time the simulated system spent in a function updating the chipset interrupt mask register was much higher than it was on the real machine. An investigation into this issue led us to conclude that, on the real machine, uncached writes to this register were getting acknowledged by the memory controller

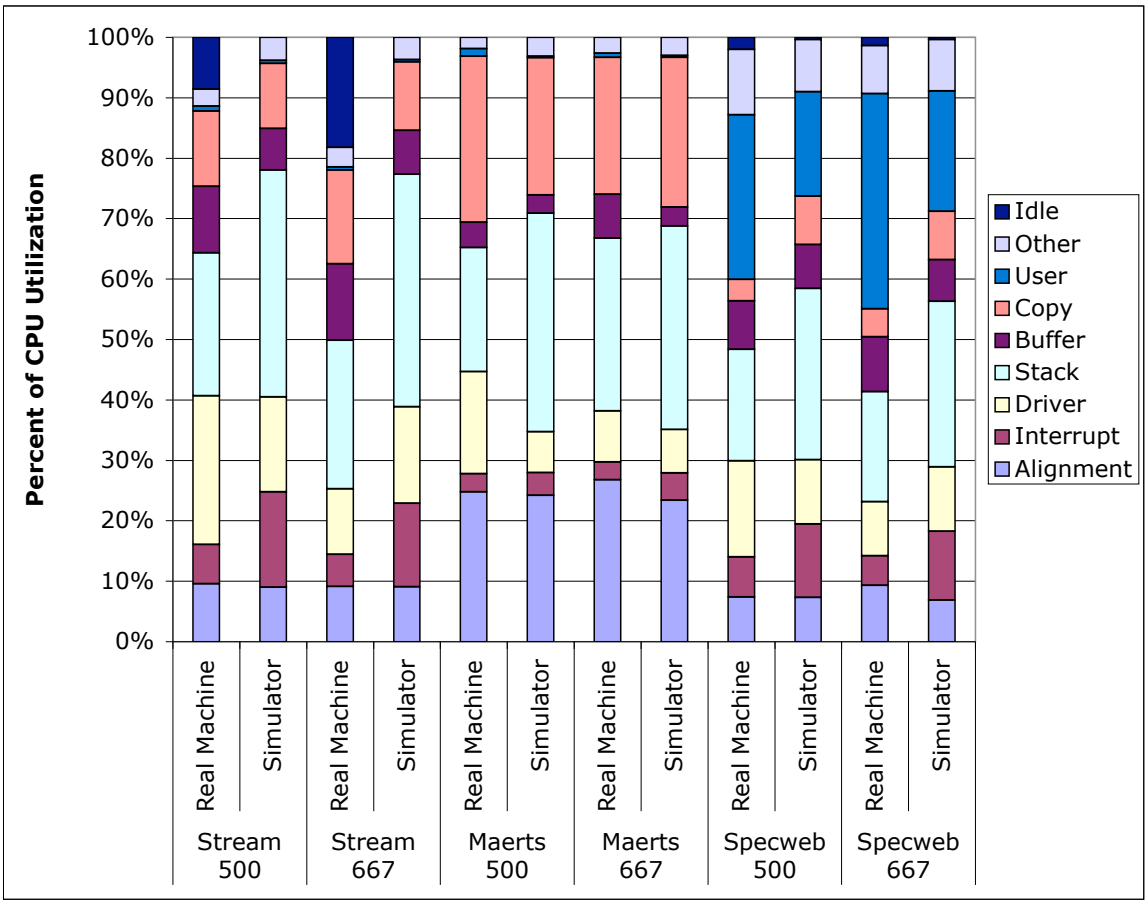

Figure 6: Processor Utilization over Benchmarks

before the write had actually propagated to the device register. As a result, the register's write latency was substantially smaller than the read latency we measured with the kernel microbenchmarks described in Section 3.

The effect of the corrected write acknowledgment timing on network bandwidth depended on the workload. The number of uncached register writes is correlated with the number of network packets transmitted. The maerts (receive) benchmark only transmits TCP ack packets, so it saw a minor 1% change in bandwidth. On the other hand, the stream (transmit) benchmark saw a 6% change in bandwidth. Since the webserver does a combination of receives and transmits, it saw a 3% change. Acknowledging writes earlier reduced the time spent in the device driver and interrupt handler, bringing the utilization breakdown more in line with the real system.

6.2 IPR Latency

Alpha CPUs have a variety of internal processor registers (IPRs) that are accessed with special PALcode instructions. Among other things, IPRs store information about the most recent TLB miss and provide access to TLB entries. The PAL-resident software TLB miss handler on Alpha CPUs makes heavy use of these IPRs. While attempting to model the TLB latency, we realized that our initial model treated IPR accesses as integer ALU operations, meaning that several of them could execute in each cycle with a one-cycle latency. In contrast, the real hardware takes three cycles per access. Also, although the Alpha documentation does not specifically mention it, we assume that only one IPR access can be performed at a time. Fixing this modeling error causes a 2%-3% change in bandwidth across all benchmarks.

6.3 TLB Miss Penalty

To further match the observed TLB miss penalty, we added an arbitrary delay of 20ns to the invocation of the TLB miss handler. Eliminating this delay causes a 10% change in bandwidth for the maerts benchmark. The stream benchmark is only affected slightly by this change.

6.4 Link Delay

Our link delay parameter controls the amount of time elapsed from when a bit is put onto the wire at the sender's NIC to when it is received at the other end. This delay is in addition to the delay incurred due to the size of the packet and the bandwidth of the link. We modeled several link delays ranging from 0us to 1us and found the change in bandwidth to be negligible.

7 Related Work

Bedicheck [1] validated the Talisman simulator, designed to model the Meerkat prototype multicomputer, against the final hardware. He achieved remarkable accuracy, but beyond a wide range of microbenchmarks he reports only on a few small kernel without any OS or I/O activity. Meerkat was based on the in-order, single-issue Motorola 88100 CPU, so detailed CPU modeling was not required.

Gibson *et al.* [2] validated the various simulation models used to develop the Stanford FLASH prototype (including SimOS-based full-system models) against the prototype hardware itself. In accordance with the focus of the FLASH project, the workloads used for validation were parallel compute-bound applications from the SPLASH-2 suite [18], which did not involve significant OS or I/O activity. Interestingly, they found (as we did) that the overhead of software TLB miss handling was a significant discrepancy between their simulator and the real machine. They also found that a fast simple in-order processor model often gave results as accurate as a more detailed out-of-order model.

Desikan *et al.* [4] created and validated a model of a Alpha 21264 microprocessor against a Compaq DS-10L workstation. They spent considerable effort modeling detailed aspects of the 21264 processor core, such as replay traps and clustered execution, resulting in an error of less than 2% for CPU-bound microbenchmarks. However, their inability to model complex interactions within the memory system caused their error on SPEC CPU2000 macrobenchmarks to average 18%. This error reflects in part issues that are effectively non-deterministic and cannot be accurately matched in a simulator, such as physical page mapping effects on cache and DRAM page conflicts and DRAM refresh timing.

8 Conclusion and Future Work

We validated the performance modeling capabilities of the M5 simulator by comparing M5 models of two Compaq Alpha XP1000 servers with their real-world counterparts. We compared network bandwidth and CPU utilization for two network-intensive micro-benchmarks and a macro-benchmark. The M5 models were able to get within 15% of the real machines' bandwidth on most benchmarks. Furthermore, the simulation results accurately reflected the impact of varying CPU frequency. We feel that this level of accuracy is quite good given the relatively generic and imprecise models used in the simulator, and the fact that our only major effort in tuning for the reference machine was to correlate memory and device register latencies. We believe that the discrepancies in some of the runs are largely due to our simulated system processing two times more transmit interrupts per packet. If we can address this problem, our results should match the real Alpha hardware even more closely.

Our short-term future work involves both verifying and correcting these discrepancies and employing additional macrobenchmarks for comparison. One longer-term enhancement to this study would be perform multiple runs of each workload while inserting small random delays into our memory system [19], thus determining whether or not the small differences we see tweaking various simulator parameters are simply artifacts of timing randomness or are statistically significant. Another future opportunity deals with investigating the applicability of simple in-order CPU models with a configurable IPC instead of a larger detailed CPU model [2]. This configuration would allow runs to be much longer while continuing to provide reasonable fidelity.

References

[1] R. C. Bedichek, "Talisman: Fast and accurate multicomputer simulation," in *Proc. 1995 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, 1995, pp. 14–24.

[2] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich, "FLASH vs. (simulated) FLASH: Closing the simulation loop," in

Proc. Ninth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX), Oct. 2000, pp. 49–58.

[3] B. Black and J. P. Shen, "Calibration of microprocessor performance models," *IEEE Computer*, vol. 31, no. 5, pp. 59–65, May 1998.

[4] R. Desikan, D. Burger, and S. W. Keckler, "Measuring experimental error in microprocessor simulation," in *Proc. 28th Ann. Int'l Symp. on Computer Architecture*, 2001, pp. 266–277.

[5] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, "Complete computer system simulation: The SimOS approach," *IEEE Parallel & Distributed Technology*, vol. 3, no. 4, pp. 34–43, Winter 1995.

[6] L. Schaelicke and M. Parker, "ML-RSIM reference manual," http://www.cse.nd.edu/~lambert/pdf/ml-rsim.pdf.

[7] P. S. Magnusson *et al.*, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[8] D. Burger, T. M. Austin, and S. Bennett, "Evaluating future microprocessors: the SimpleScalar tool set," Computer Sciences Department, University of Wisconsin–Madison, Tech. Rep. 1308, July 1996.

[9] R. E. Kessler, "The Alpha 21264 microprocesor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, March/April 1999.

[10] High Performance Technical Computing Group, "Exploring Alpha Power for Technical Computing," April 2002, http://h18002.www1.hp.com/alphaserver/download/wp_alpha_tech_apr00.pdf.

[11] L. W. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," in *USENIX Annual Technical Conference*, 1996, pp. 279–294. [Online]. Available: citeseer.csail.mit.edu/mcvoy96lmbench.html

[12] Hewlett-Packard Company, "Netperf: A network performance benchmark," http://www.netperf.org.

[13] Standard Performance Evaluation Corporation, "SPECweb99 benchmark," http://www.spec.org/web99.

[14] Apache Software Foundation, "Apache HTTP server," http://httpd.apache.org.

[15] P. Barford and M. Crovella, "Generating representative web workloads for network and server performance evaluation," in *Measurement and Modeling of Computer Systems*, 1998, pp. 151–160.

[16] L. R. Hsu, A. G. Saidi, N. L. Binkert, and S. K. Reinhardt, "Sampling and stability in TCP/IP workloads," in *Proc. First Annual Workshop on Modeling, Benchmarking, and Simulation*, June 2005, pp. 68–77.

[17] P. S. Panchamukhi, "Smashing performance with OProfile," IBM DeveloperWorks, Oct. 2003, http://www-106.ibm.com/developerworks/linux/library/l-oprof.html.

[18] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. 22nd Ann. Int'l Symp. on Computer Architecture*, June 1995, pp. 24–36.

[19] A. R. Alameldeen and D. A. Wood, "Variability in architectural simulations of multi-threaded workloads," in *Proc. 9th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2003, pp. 7–18.