VALIDATING AN ARCHITECTURAL SIMULATOR

Erich M. NAHUM

CMPSCI Technical Report 96-40

September, 1996

Validating an Architectural Simulator

Erich M. Nahum
Department of Computer Science
University of Massachusetts at Amherst
nahum@cs.umass.edu

September 1996

Department of Computer Science Technical Report 96-40

Abstract

This paper reports on our experiences in building an execution-driven architectural simulator that is meant to *accurately* capture performance costs of a machine for a particular class of software, namely, network protocol stacks such as TCP/IP. The simulator models a single processor of our Silicon Graphics Challenge shared-memory multiprocessor, which has 100 MHz MIPS R4400 chips and two levels of cache memory. We describe our validation approach, show accuracy results averaging within 5 percent, and present the lessons learned in validating an architectural simulator.

1 Introduction

We have designed and implemented a execution-driven uniprocessor simulator for our 100 MHz R4400-based SGI Challenge [6]. The purpose of this simulator is to understand the performance costs of a network protocol stack running in user space on our SGI machine, and to guide us in identifying and reducing bottlenecks [11].

The primary goal of this simulator has been to *accurately* model performance costs for our SGI machine. Much of the simulation literature discusses the tradeoff between speed and accuracy, and describes techniques for making simulations fast. However, accuracy is rarely discussed (notable exceptions include [2, 4, 5]), and the tradeoff between accuracy and speed has not been quantitatively evaluated. Given that our simulator is meant to capture *performance* costs, it must be more than an emulator that duplicates the execution semantics of the hardware or counts events such as cache misses. The simulator should *perform* similarly to our actual hardware, i.e., an application taking N time units on the real hardware should take N simulated time units on the simulator. We quantify accuracy in terms of how closely an application's performance on the simulator comes to matching the performance on the real hardware.

Our simulator is designed for a specific class of software, namely, computer network communication protocols such as TCP/IP, and we use them to evaluate our simulator's accuracy. Network protocols

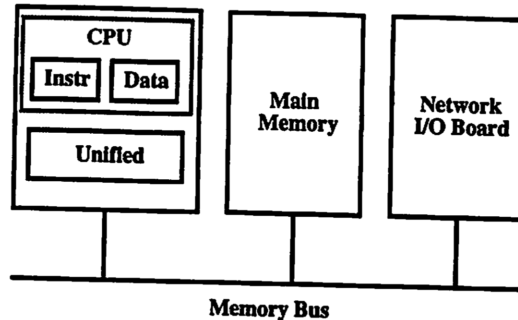

Figure 1: Machine Organization

most closely resemble integer benchmarks, and use essentially load, store, control, and simple arithmetic instructions. Our protocol benchmarks have cache hit rates ranging from 75-100 percent, and spend between 15 and 75 percent of time waiting for memory. We also evaluate accuracy on memory-intensive microbenchmarks from LMBench [10]. We have not evaluated accuracy on numeric (i.e., floating-point) benchmarks such as the SPEC 95 FP suite. On the workloads that we have tested, we find that the simulator predicts latencies that are, on average, within 5 percent of the actual measured latencies.

This paper reports on our experiences in constructing this simulator. We describe our simulator, enumerate our assumptions, show our approach to validation, and present accuracy results. We conclude with the lessons learned in validating an architectural simulator.

2 Architectural Simulator

Our architectural simulator is built using MINT [15], a toolkit for implementing multiprocessor memory reference simulators. MINT interprets a compiled binary directly and executes it, albeit much more slowly than if the binary was run on the native machine. This process is called *direct execution*. MINT is designed for use with MIPS-based multiprocessors, such as our SGI machines, and has support for the multiprocessor features of IRIX. Unlike several other simulation packages, it only requires the binary executable of the program[1]. As a consequence, all the application source does not need to be available, and the application does not need to be modified for use in the simulator. This means that the same exact binary is used on both the actual machine and in the simulator.

A simulator built using MINT consists of 2 components: a front end, provided by MINT, which handles the interpretation and execution of the binary, and a back-end, supplied by the user, that maintains the state of the cache and provides the timing properties that are used to emulate a target architecture. The front end is typically called a *trace generator*, and the back end a *trace consumer*. On each memory reference, the front end invokes the back end, passing the appropriate memory address. Based on its internal state, the back end returns a value to the front end telling it whether to continue (for example, on a cache hit) or to stall (on a cache miss).

[1] MINT does not yet support dynamic linking; thus, the binary must be statically linked.

We have designed and implemented a back end for use with MINT to construct a uniprocessor simulator for our 100 MHz R4400-based SGI Challenge. Figure 1 shows the memory organization for this machine. The R4400 has separate 16 KB direct-mapped on-chip first level instruction and data caches with a line size of 16 bytes. Our SGI machine also has a 1 MB second-level direct-mapped on-board unified cache with a line size of 128 bytes. The simulator captures the cost of the important performance characteristics of the SGI platform. It supports multiple levels of cache hierarchy, including the inclusion property for multi-level caches, and models the aspects of the MIPS R4400 processor that have a statistically significant impact on performance, such as branch delays and load delay pipeline interlocks.

The primary goal of the simulator has been *performance* accuracy; thus, we have attempted to ensure the timing accuracy of the simulator. Accuracy generally depends on a number of issues:

- The accuracy of the *assumptions* made by the simulator, and their relevance to the real system,
- The accuracy to which *instruction costs* (e.g., branches, adds, multiplies) are modeled,
- The accuracy to which *memory references* (i.e., cache hits and misses) are modeled.

We describe our approach to each of these issues in turn.

2.1 Assumptions

Several assumptions are used, most of which are intrinsic to MINT:

- We ignore context switches, TLB misses, and potentially conflicting operating system tasks; the simulation essentially assumes a dedicated machine. This assumption is required by MINT and most other simulators.
- Unless otherwise specified, all instructions and system calls take one cycle. Certain exceptional library calls, such as malloc(), also take one cycle. This is described in more detail below. This assumption is required by MINT and most other simulators.
- We assume that the heap allocator used by MINT is the same as used by the IRIX C library, or that any differences between the two allocators does not impact accuracy. This assumption is required by MINT.
- We assume that virtual addresses are the same as physical addresses. For virtually-indexed caches such as those on-chip in the R4400, this is accurate. However, in our SGI systems, the second-level cache is physically indexed. The mapping between virtual and physical addresses on the real system is determined by the IRIX operating system. It is reported that IRIX 5.3 uses *page coloring* [3, 9] as a virtual-to-physical mapping strategy. In page coloring, whenever a new virtual-to-physical mapping is created, the OS attempts to assign a free physical page so that both the virtual and physical addresses map to the same bin in a physically indexed cache. This way, pages that are adjacent in virtual memory will be adjacent in the cache as well. However, if a free physical page that meets this criterion is not available, another free page will be chosen. Thus, our assumption matches the behavior of the

Instr	Num	%
mul.d	1	0.01
bgez	109	0.70
or	152	0.97
andi	180	1.15
beq	367	2.34
slti	1039	6.64
addu	1112	7.10
bne	1219	7.79
addiu	1610	10.28
st	2284	14.59
nop	3560	22.74
lw	3619	23.12
Total	15656	100.00

Table 1: Instruction Frequencies

operating system if the OS is successful in finding matching pages. Unfortunately, there is no way that we are aware of that allows us to determine the virtual-physical mappings and thus see how well the OS is finding matched pages. Since the OS may choose a non-matching page, the virtual address and the physical address may map to different bins in the L2 cache. Thus, an application executing in the simulator may experience conflicts between two lines in L2 that would not occur in the real system, and vice-versa. The impact on accuracy may be exacerbated by the inclusion property [1], which requires that, for coherency reasons, all lines cached in L1 must be held in L2. If a (possibly erroneous) L2 conflict forces a line to be removed, it must be invalidated in L1 as well.

2.2 Modeling Instruction Costs

One of the major assumptions in MINT is that all instructions and replaced functions (such as malloc() and uspsema()) execute in one cycle. For most instructions on RISC architectures, this is a reasonable assumption. For a few instructions, however, this assumption is incorrect. Integer divides, for example, take 75 cycles. However, MINT allows the user to change the times associated for each instruction via a supplied file that lists instructions and functions along with their simulated costs.

To see which instructions are actually being used by our applications, we developed a MINT back-end tool that counts dynamic instruction use and prints out a histogram. Table 1 presents an example of instruction frequencies for a program. This allows us to focus our attention on making sure that the time values for these instructions are correct. In the above example, it is much more important that the time for a branch (bne, beq) be correct than for a floating point multiply (mul.d), since branches occur orders of magnitude more frequently. We then wrote micro-benchmarks stressing use of those instructions and functions in order to measure their cost on our system. We also wrote a similar tool to count the use of replaced functions and to count data sizes (e.g., byte, word) of loads and stores.

Instruction or Function	Time in Cycles
div	75
divu	75
mult	2
multu	2
uspsema	113
usvsema	113
malloc	70
free	70
sginap	6000
gettimeofday	1580

Table 2: Instructions that take more than 1 cycle

Unfortunately, only a small subset of timing values are available in the MIPS R4000 Microprocessor Users Manual [7]. For instructions not listed there, we needed to construct micro-benchmarks to determine their cycle times. Table 2 presents the cycle times of instructions, functions, and system calls used that do not follow the single-cycle instruction assumption. These values are fed into MINT via the cycle file. The list in Table 2 is far from complete; these happen to be the instructions and functions that our benchmarks use. MINT's original notion of time was solely in cycles. We convert this notion into real time by setting a cycle time in nanoseconds.

2.3 Pipelining in the R4000

Part of the single-cycle assumption of MINT means that the R4000 pipeline is not modeled precisely. The single cycle assumption implies that the pipeline never stalls, which is not true under certain conditions. For example, if an instruction loads a value from memory into a register, and the subsequent instruction uses that register, the latter instruction will stall. This stalling is called a *pipeline interlock* [8]. MINT does not yet model pipeline interlocks.

To accurately model load delay pipeline interlocks, we instead keep track of loads in the back end. In the R4000, loads have a 2-cycle load delay. This means we need to keep track of at most 2 registers, since only 1 load can be issued each cycle, and we need only track a loaded register for 2 cycles. Whenever a register is loaded, we keep track of it and allow it to *age* over time. On each instruction, we look at the registers used and compare them with those recently loaded. If any of the registers are identified as having not completed their load delay, the instruction will stall one or two cycles as appropriate.

Branches and jumps can also cause stalls in the MIPS pipeline. Jumps, or unconditional branches, take 4 cycles. Branches are more complex than jumps in that they take 4 cycles if the branching condition is true, and 2 cycles otherwise. In either case, one cycle is exposed to the compiler as a branch delay slot, which the compiler will attempt to fill with a useful instruction if possible, or a NOP otherwise.

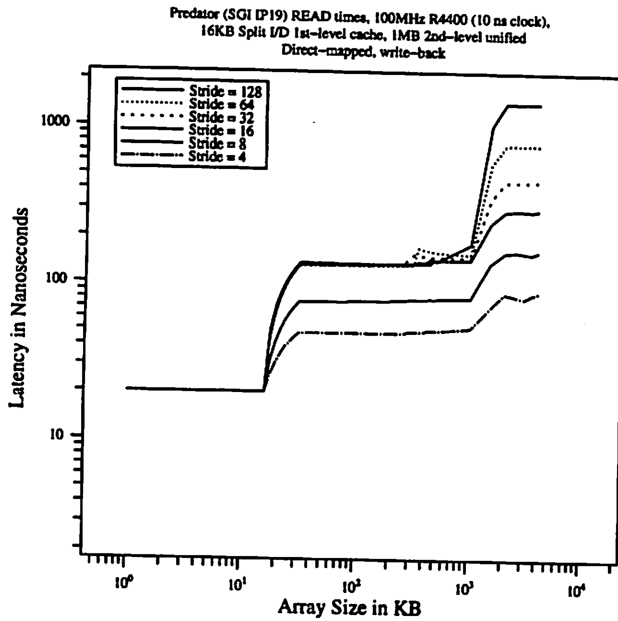

Figure 2: Actual Read Latencies

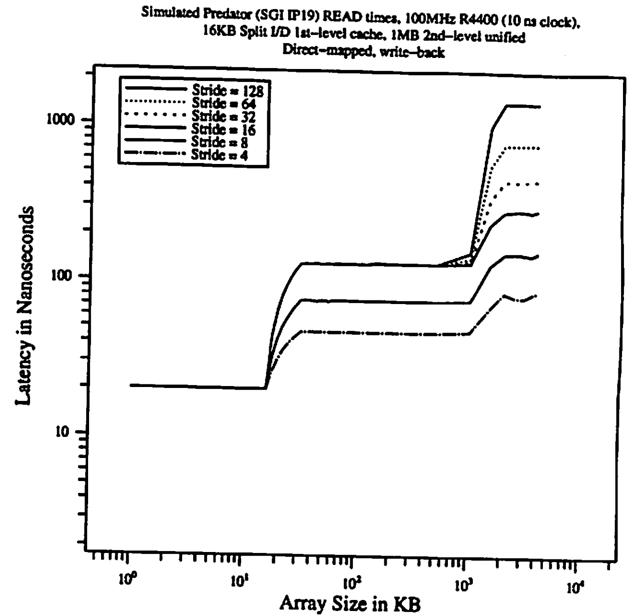

Figure 3: Simulated Read Latencies

MINT will charge a cost for the instruction in this visible delay slot. This implies that we should set the cost value for a jump at 3 cycles, and the cost for branch instructions to 3 or 1 cycles depending on whether the branch is taken. However, MINT cannot distinguish between these cases, and we are only allowed to set a single value for a branch instruction.

To accurately model the dynamic costs of branches, the back end keeps track of the program counter on every instruction and data reference. When the PC does not change to the next sequential instruction (i.e., changes by anything other than 4), the back end delays by 2 cycles to emulate the branch delay cost. This accurately covers the cost of both jump instructions and all taken branches, so that we do not need to change their cycle time values.

2.4 Modeling Memory References

We used the memory striding benchmarks from LMBench [10] to measure the cache hit and miss latencies for all three levels of the memory hierarchy: L1, L2, and main memory. These in turn gave us values with which to parameterize the back-end cache simulator. Table 3 lists the cycle times to read and write the caches on the 100MHz SGI Challenge (IP19). Note that references which hit in L1 incur no additional penalty beyond the cost incurred by the instruction in isolation, e.g., an integer add instruction that hits in the instruction cache and whose operands are all available costs only 1 cycle.

The same memory stride programs were then run in the simulator, using the table of modified instruction times, to ensure that the simulated numbers agreed with those from the real system. Figure 2 shows the LMBench read memory access time as a function of the area walked by the stride benchmark, as run on our 100 MHz R4400 SGI Challenge. We call this graph a *memory signature*. The memory signature illustrates the access times of the first level cache, the second-level cache, and main memory. When the area walked by the benchmark fits within the first level cache (i.e., is 16 KB or less), reading a byte in the area results

Layer in Hierarchy	Read time	Write time
L1 Cache	0	0
L2 Cache	11	11
Challenge Bus	141	147

Table 3: Read and write times in cycles

Layer in Hierarchy	Real time	Simulated time	Diff (%)
L1 Cache	20	20	0
L2 Cache	131	134	2
Challenge Bus	1440	1440	0

Table 4: LMBench real and simulated values in μsec.

in a first-level cache hit and takes 20 nanoseconds. When the area fits within the second level cache (i.e., is between 16 KB and 1 MB in size), reading a byte results in a second-level cache hit and takes 134 nanoseconds. If the area is larger than 1 MB, main memory is accessed, and the time to read a byte is 1440 nanoseconds. Note that the scales in Figure 2 are logarithmic in both the x and y axes.

Figure 3 shows the memory signature of the same binary being run on the simulator for the same machine. Table 4 lists the numbers from Figures 2 and 3 in textual form for comparison. As can be seen, the simulator models the cache memory behavior very closely.

2.5 Summary of Validation Sequence

In general, we use a sequence of actions to validate a particular application. This sequence can be repeated several times depending on the application. As each new application is introduced:

- It is instrumented to see its instruction usage.
- If any previously unexamined instructions are used in a significant fashion, appropriate micro-benchmarks for those instructions are produced and timings ascertained.
- The table of instruction times is updated to reflect the newly determined values.
- The stride benchmarks are re-run inside the simulator to ensure that the memory latencies are still accurate.
- Finally the application is run on the cache simulator.

Benchmark	Simulated	Real	Error (%)
TCP Send, cksum off	76.63	78.58	2.48
TCP Send, cksum on	147.84	146.66	-0.81
UDP Send, cksum off	18.43	15.97	-15.40
UDP Send, cksum on	71.99	70.30	-2.41
TCP Recv, cksum off	58.06	62.65	7.33
TCP Recv, cksum on	190.47	198.39	3.99
UDP Recv, cksum off	33.80	32.84	-2.95
UDP Recv, cksum on	161.78	158.84	-1.85
Average Error			4.65

Table 5: Macro benchmark times (μsec) and relative error

3 Validation Results

Table 5 lists the current set of benchmarks, with their corresponding real and simulated latencies in microseconds, and the relative error. Error is defined as

$$Error = \frac{(Simulated\ Value - Real\ Value)}{Real\ Value} * 100$$

A negative error means the simulator *underestimates* the real time; a positive value means it *overestimates* the real time. The *average* error is calculated as the mean of the absolute values of the individual errors. This is to prevent positive and negative individual values from canceling each other out. Note the average error is within 5 percent, with the worst case error being about 15 percent.

Table 6 presents more accuracy results, this time modifying the protocol benchmarks by adding copies (COPY ON), or by executing the CORDed versions of the executables. CORD [13] is a binary re-writing tool that uses profile-guided code positioning [12] to reorganize executables for better instruction cache behavior. An original executable is run through Pixie [14] to determine its runtime behavior and profile which procedures are used most frequently. CORD uses this information to re-link the executable so that procedures used most frequently are grouped together. This heuristic approach is meant to minimize the likelihood that "hot" procedures will conflict in the caches, both in the L1 instruction cache and in the L2 unified cache.

One side affect of this is that CORDed executables tend to have better accuracy than regular executables. Since one of the simulator's assumptions is that virtual addresses are the same as physical addresses, which is not true for the L2 cache, part of the simulator's accuracy depends on modeling conflicts in L2 correctly. The larger the number of conflicts, the more likely the simulator will not capture their cost accurately. Similarly, the fewer the number of conflicts, the less impact the conflicts have on performance, and the less impact the virtual = physical assumption has on accuracy. For example, one protocol benchmark, the UDP Send without checksumming, has the worst accuracy on the simulator with an error of 15 percent. In this benchmark, all the L1 data cache misses are caused by evictions that are the result of a conflict in the L2

Benchmark	Simulated	Real	Error (%)
TCP Send COPY ON Cksum OFF	201.27	200.47	-0.40
TCP Send COPY ON Cksum ON	268.55	264.58	-1.50
UDP Send COPY ON Cksum OFF	131.83	126.19	-4.47
UDP Send COPY ON Cksum ON	184.55	185.39	0.45
TCP Recv COPY ON Cksum OFF	248.10	258.06	3.86
TCP Recv COPY ON Cksum ON	313.47	327.21	4.20
UDP Recv COPY ON Cksum OFF	218.35	217.68	-0.31
UDP Recv COPY ON Cksum ON	278.24	267.21	-4.13
CORD TCP Send Cksum OFF	72.64	68.72	-5.70
CORD TCP Send Cksum ON	148.42	144.76	-2.53
CORD UDP Send Cksum OFF	12.54	13.49	7.10
CORD UDP Send Cksum ON	66.05	65.32	-1.12
CORD TCP Send COPY ON Cksum OFF	197.10	190.59	-3.42
CORD TCP Send COPY ON Cksum ON	268.97	251.41	-6.98
CORD UDP Send COPY ON Cksum OFF	126.02	127.36	1.05
CORD UDP Send COPY ON Cksum ON	178.54	176.03	-1.43

Table 6: Macro benchmark times (μsec) and relative error

cache[2]. We believe this is the main cause of inaccuracy in this benchmark. The CORDed version of this executable does not exhibit this behavior, and its 7 percent error is half that of the regular executable.

Due to time constraints, we could not run every permutation of every benchmark. However, given the range of cache hit rates and instructions used that have been exercised by the simulator, we feel confident that it is very accurate for this class of applications.

4 Sample Output

Here we present a sample output from the simulator, in this case from the send-side UDP experiment with checksumming disabled. The sample gives an idea of the information captured by the simulator.

```
[L1 I Cache] Stats:  (  542598 invalidates,   271299 evicts,   439325653 cycles)
Operation:    Number        Hits (   %  )    Misses (   %  )     Cycles ( %  )
   READ:  196159176  188291498 ( 95.99)   7867678 (  4.01)          0 ( 0.00)
  WRITE:          0          0 (  0.00)         0 (  0.00)          0 ( 0.00)
READ_EX:          0          0 (  0.00)         0 (  0.00)          0 ( 0.00)
  TOTAL:  196159176  188291498 ( 95.99)   7867678 (  4.01)            ( 0.00)

[L1 D Cache] Stats:  (  542598 invalidates,   271299 evicts,   439325653 cycles)
```

[2]The numbers are given in the example in Section 4. Note the number of evictions in the L1 data cache is the essentially the same as the number of misses.

Operation:	Number	Hits (%)	Misses (%)	Cycles (%)
READ:	49920127	49648826 (99.46)	271301 (0.54)	0 (0.00)
WRITE:	36084989	36084989 (100.00)	0 (0.00)	0 (0.00)
READ_EX:	0	0 (0.00)	0 (0.00)	0 (0.00)
TOTAL:	86005116	85733815 (99.68)	271301 (0.32)	(0.00)

[L2 U Cache] Stats: (0 invalidates, 0 evicts, 439325653 cycles)

Operation:	Number	Hits (%)	Misses (%)	Cycles (%)
READ:	8138979	7596379 (93.33)	542600 (6.67)	83560169 (19.02)
WRITE:	0	0 (0.00)	0 (0.00)	0 (0.00)
READ_EX:	0	0 (0.00)	0 (0.00)	0 (0.00)
TOTAL:	8138979	7596379 (93.33)	542600 (6.67)	83560169 (19.02)

[IP19 Bus] Stats: (0 invalidates, 0 evicts, 439325653 cycles)

Operation:	Number	Hits (%)	Misses (%)	Cycles (%)
READ:	542600	542600 (100.00)	0 (0.00)	76506600 (17.41)
WRITE:	0	0 (0.00)	0 (0.00)	0 (0.00)
READ_EX:	0	0 (0.00)	0 (0.00)	0 (0.00)
TOTAL:	542600	542600 (100.00)	0 (0.00)	76506600 (17.41)

Instruction	Number (%)	Cycles (%)
loads:	49920127 (25.45)	49920127 (11.36)
load stalls:	28215096 (56.52)	33912375 (7.72)
stores:	35542391 (18.12)	35542391 (8.09)
branches:	16007752 (8.16)	37711672 (8.58)
jumps:	13564950 (6.92)	40694850 (9.26)
control:	29572702 (15.08)	78406522 (17.85)
adds:	32014393 (16.32)	32014393 (7.29)
subtracts:	2984289 (1.52)	2984289 (0.68)
muls:	271299 (0.14)	542598 (0.12)
divs:	1111 (0.00)	83325 (0.02)
shifts:	4069485 (2.07)	4069485 (0.93)
logicals:	12751053 (6.50)	12751053 (2.90)
sets:	5968578 (3.04)	5968578 (1.36)
immediates:	1899093 (0.97)	1899093 (0.43)
loads:	49920127 (25.45)	83832502 (19.08)
stores:	35542391 (18.12)	35542391 (8.09)
control:	29572702 (15.08)	78406522 (17.85)
ariths:	59959301 (30.57)	60312814 (13.73)
nops:	20891134 (10.65)	20891134 (4.76)
others:	273521 (0.14)	273521 (0.06)
mem:	0 (0.00)	160066769 (36.43)
cpu:	0 (0.00)	279258884 (63.57)
total:	196159176 (100.00)	439325653 (100.00)

As can be observed, the simulator lists both how many times an event happened as well as what percentage of the total time to which the event contributed. For example, we see that control operations make up 15 percent of the instruction usage, but contribute 17.85 percent of the total cycles. On average, a branch takes roughly 2.7 cycles. This tells us even if branches cost a single cycle, our application performance would only improve about 10 percent.

5 Improving Accuracy

There are several possibilities for improving the accuracy of the simulator:

- *modeling the pipeline more accurately.* This would improve accuracy on more complex events, particularly combinations of instructions that affect the pipeline. The interaction between the pipeline and instructions with overlapped execution is not captured fully. For example, a multiply instruction placed in a branch delay slot is probably not modeled precisely in terms of how much (or little) the pipe is stalled. However, adding this amount of detail requires a large amount of work either to the MINT front end or to the back-end simulator. It is not clear what the overall impact on accuracy would be, or whether the amount of work would be worth the effort.
- *virtual addresses vs. physical addresses.* We can examine more carefully the assumption that physical addresses are the same as virtual addresses. However, this assumption is usually correct in kernel code. In addition, without access to more information about IRIX's policies for assigning virtual pages to physical ones, we cannot know whether we are improving accuracy.
- *improving L2 write accuracy.* At the moment, L2 write costs are over-estimated by the simulator by about 30 percent. This appears to be a consequence of the presence of a write buffer and the lack of modeling pipeline effects. This might be able to be fixed; however, write misses that hit in L2 are extremely rare, so improving this will probably have little impact on overall accuracy.

What is interesting is how well the machine is being modeled despite many features not being represented. For example, the TLB, store buffer, and write buffer are all ignored, with apparently little impact on overall accuracy.

Our belief is that improving accuracy will only be necessary upon introducing new classes of software, most obviously floating-point benchmarks. Modeling pipeline slips and stalls will become more important for multiple-cycle instructions such as multiplies and divides.

6 Lessons Learned

Several general lessons were learned (or re-learned) over the process of constructing and validating the simulator. Some of these are general software engineering principles, but mostly they relate to accuracy and validation.

- *Frequency is key.* The frequency of events plays a key role in the overall accuracy. This can be thought of as the 90/10 rule or RISC approach to validation. One way to think of validation is trying to minimize error, where error is defined as follows:

$$Error = \sum_{i=1}^{n} freq(E_i) * (real(E_i) - sim\,(E_i))$$

Here, E_i is event i in the system, $freq(E_i)$ is the frequency of event i, $real(E_i)$ is the real cost of event i, and $sim(E_i)$ is the simulated cost of event i. Examples of events include uses of a particular instruction, cache misses, or taken branches.

Obviously, the frequency of a particular event is application-dependent. To know the frequency of various events, one must be aware what the application is doing, i.e., its dynamic behavior at run time. As mentioned before, modeling events that happen frequently is crucial. Similarly, events that happen occasionally can be modeled less carefully; however, their cost must be taken into account. For example, L2 write misses are very rare, but their cost is so high (146 cycles) that they must be accounted for.

Events that *never* happen can be ignored, or assigned simple costs. For example, our simulator completely neglects floating point costs, but given that not a single floating point instruction is executed, this neglect does not impact our accuracy. It does, however, save us *time*: it eliminates the need to write micro-benchmarks to measure event costs, obviates the requirement to implement appropriate functionality in the simulator, and improves the simulator performance by not wasting cycles testing for events that never happen.

The disadvantage of this is that the simulator is effectively tuned to the application, in that different applications can have different frequencies of events. However, the simulator can be tuned for a new application as necessary.

- *The law of diminishing returns applies.* A corollary of the frequency lesson, accuracy tends to get harder and harder to achieve over time. For example, adding the functionality to model the load delay pipeline took a couple of days to design, implement, test and debug. Adding this feature changed the accuracy of some of the checksummed protocol benchmarks from 20 percent to 5 percent. However, it only improved the *average* accuracy of the whole suite of benchmarks by one percent, and slowed down the simulator by a factor of 2-3.

- *Use tools.* Events that never happen can be ignored, however, one must *be certain* that they never happen! Writing special-purpose tools to determine frequency of cases is extremely useful.

- *Use microbenchmarks.* Microbenchmarks that provoke certain types of behavior both allow the unit cost of those events to be measured and give a means to test the accuracy of the simulator on those events. This makes the process of validating the simulator essentially self-correcting. Of course, one should not bother writing micro-benchmarks for events that never happen.

- *Build in lots of self-checking code.* The simulator has huge amounts of assertion checks and tracing print statements that are defined by #ifdef's. These checks are normally compiled away for speed. However, turning them on explicitly tested all assumptions and that variables were in consistent states. Particularly, whenever a major component was added to the simulator, we would first run things with the full set of checks turned on to make sure any assumptions had not been violated by the new functionality. Some of these assumptions were type issues that would be addressed by using a more type-safe language than C, such as Modula-3. However, most were assumptions about which state (out of several possible correct ones) a variable (such as a cache line) was in.

- *Automate, automate, automate.* This greatly reduces the opportunities for error, and makes it easy to regenerate results when the simulator changes. We wrote many scripts and post-processing tools to

do things such as calculate the relative error.

- *Iterate until satisfied.* An implementer can go through the cycle of adding functionality, re-executing benchmarks, and re-parameterizing the simulator with the new results until the accuracy is satisfactory. We iterated through the validation process roughly 15 times.

The combination of these factors along with the accuracy results gives us great confidence in the resulting code.

7 Summary

This paper has reported our experiences in building an execution-driven architectural simulator that is meant to accurately capture performance costs for a particular class of software, namely, network protocol stacks. The simulator models a single processor of our Silicon Graphics Challenge shared-memory multiprocessor, which has 100 MHz MIPS R4400 chips and two levels of cache memory. We have presented our approach to validation and shown average accuracy of within 5 percent for our class of applications. We have described the lessons learned in validation, chief of which is that modeling frequent events accurately is key.

Acknowledgements

This work benefited from discussions with Amer Diwan, Kathryn McKinley, Eliot Moss, Jack Veenstra, and David Yates. Jim Kurose, Eliot Moss, Don Towsley and Jack Veenstra gave many useful comments on earlier drafts of this paper.

This research supported in part by NSF under grant NCR-9206908, and by ARPA under contract F19628-92-C-0089. Erich Nahum is supported by a Computer Measurement Group Fellowship.

References

[1] Jean-Loup Baer and Wen-Hann Wang. On the inclusion property for multi-level cache hierarchies. In *Proceedings 15th International Symposium on Computer Architecture*, pages 73–80, Honolulu Hawaii, June 1988.

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

[3] Edouard Bugnion, Jennifer M. Anderson, Todd C. Mowry, Mendel Rosenblum, and Monica S. Lam. Compiler-directed page coloring for multiprocessors. In *Proceedings of the Seventh International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Cambridge MA, October 1996.

[4] Brad Calder, Dirk Grunwald, and Joel Emer. A system level perspective on branch architecture performance. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 199–206, Ann Arbor, MI, November 1995.

[5] Amer Diwan, David Tarditi, and Eliot Moss. Memory-system performance of programs with intensive heap allocation. *ACM Transactions on Computer Systems*, 13(3):244–273, 1995.

[6] Mile Galles and Eric Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. Technical report, Silicon Graphics Inc., Mt. View, CA, May 1994.

[7] Joe Heinrich. *MIPS R4000 Microprocessor Users Manual (2nd Ed.)*. MIPS Technologies, Inc., Mt. View, CA, 1994.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach (2nd Edition)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1995.

[9] Richard E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10(4):338–359, November 1992.

[10] Larry McVoy and Carl Staelin. LMBENCH: Portable tools for performance analysis. In *USENIX Technical Conference of UNIX and Advanced Computing Systems*, San Diego, CA, January 1996.

[11] Erich M. Nahum, David J. Yates, James F. Kurose, and Don Towsley. Cache behavior of network protocols. In preparation, August 1996.

[12] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (PLDI)*, pages 16–27, White Plains, NY, June 1990.

[13] Silicon Graphics Inc. Cord manual page, IRIX 5.3.

[14] Michael D. Smith. Tracing with Pixie. Technical report, Center for Integrated Systems, Stanford University, Stanford, CA, April 1991.

[15] Jack E. Veenstra and Robert J. Fowler. MINT: A front end for efficient simulation of shared-memory multiprocessors. In *Proceedings 2nd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Durham, NC, January 1994.