

Can Trace-Driven Simulators Accurately Predict Superscalar Performance?

Bryan Black, Andrew S. Huang, Mikko H. Lipasti and John Paul Shen

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh PA, 15213
{black, ahuang, mhl, shen}@ece.cmu.edu

Abstract

There are four crucial issues associated with performance simulators: simulator retargetability, simulator validation, simulation speed and simulation accuracy. This paper documents our experiences in developing performance simulators and our recent findings in using these simulators. We are concerned with all four of the crucial issues. Our first-generation tool, VMW, focused on achieving retargetability. Our second-generation tool, MW, significantly improved simulation speed. Recently we validated a PowerPC 604 simulator model, generated using MW, against an actual PowerPC 604 hardware system. We also present results on simulating extremely long traces on our PowerPC 620 model and highlight potential inaccuracies that can result from trace sampling. As processor complexity continues to increase at a rapid rate and microarchitectures continue to become more speculative, it is not clear whether the trace-driven paradigm of performance simulation can continue to effectively predict actual machine performance.

1. Introduction

Trace-driven performance simulators, or timing simulators, are used by microarchitects during the early design phase for exploring various design trade-offs in a quantitative fashion. Simulation results are used to confirm or correct designers' intuitions. There are four important issues regarding performance simulators; two concern the construction and two involve the use of the simulators. These are: simulator *retargetability*, simulator *validation*, simulation *speed* and simulation *accuracy*.

Retargetability not only allows a simulator to be reused for the next generation design, it also facilitates efficient design changes and hence the ease of design space exploration. Validating the correctness of a simulator is a very difficult task. One possibility is to validate a performance simulator against the hardware model. However, frequently in the early design phase the hardware model is not yet available. Furthermore, due to the complexity and details of the hardware model, only relatively short traces can be exercised. With increasing complexity of the processor design and the lengthening of the traces used, simulation time has been growing and simulation speed can become an issue. Ultimately it is the accuracy of the simulation results that determines the usefulness of these simulators. Other than the difficulty in validating the simulation model, there are other sources of inaccuracy, some of which may be due to fundamental limitations of the trace-driven paradigm.

This paper is a progress report of our efforts in developing performance simulators. We are concerned with all four of the above-stated issues. Our second-generation tool called MW is described in Section 2. In Section 3 our efforts in validating the MW-generated PowerPC 604 model against an

actual PowerPC 604 machine are documented. In Section 4 we present some of our recent experimental results on simulating extremely long traces and the potential inaccuracies that can result when trace sampling is used. The MW-generated PowerPC 620 model is used in this experimentation. We present our informal conclusions in Section 5.

2. Microarchitecture workbench

We began developing simulation tools for superscalar microarchitectures in 1992. Our goal was to develop tools that can be effectively used by both industry designers as well as university researchers and students. Two generations of such tools have been developed and we have made significant use of these tools in our research activities and classroom instruction. Quite a few copies of our first-generation tool have been distributed to a number of industrial companies.

2.1. First generation tool: VMW

Our first-generation tool called VMW (Visualization-based Microarchitecture Workbench) has been fully functional since 1994 [DS93, DS95]. The primary design objectives of VMW were retargetability and visualization support. VMW defined a set of machine description files, which once specified can be "compiled" into a working simulator. Design changes can be made easily by editing the files and a new simulator can be quickly generated by recompiling the files. Extensive visualization capabilities were also incorporated to allow the user to "see" the dynamic behaviors of the machine while simulation is taking place. These two design objectives led to capabilities that we called *simulator compilation* and *trace animation*.

Simulators for a number of real machines were implemented using VMW. These machines include the Alpha AXP 21064 [Dig92] and 21164 [BK95], the RS/6000 [Gro90], the PowerPC 601 [IBM93] and 620 [DNS95], and a multithreaded version of the PowerPC 620 developed at CMU. Once the architecture description files are in place, for each new microarchitecture, approximately 1-3 months are needed to write and debug the machine description files. Since the publication of the paper [DS95], close to 50 copies of the VMW source code have been distributed.

In early 1995 we began work on a second-generation tool by improving on VMW. The three major changes were: 1) removal of all the visualization capabilities; 2) redefining the C++ implementation framework; and 3) performing specific software optimizations. We discovered that while visualization was very helpful and interesting to a novice user, it quickly became a real overhead in terms of simulation time. We also re-architected the C++ implementation framework based on lessons learned in implementing several machine models. We also spent some effort in optimizing the code itself. We ended up calling the second-generation tool MW

(Microarchitecture Workbench) [HD95].

2.2. Second generation tool: MW

Similar to its predecessor, MW provides a framework that minimizes the amount of work necessary to produce a simulator and the amount of changes necessary to evaluate an alternative microarchitecture. This is done by placing most of the processor specification in the machine description files, which are read by the simulator at run time. The behavior specification of the processor is written in C++ and compiled to become part of the simulator. In this setup, minor changes in processor design (such as increasing the number of integer pipelines) can be evaluated by changing only the machine description files, resulting in a short cycle time for assessing the impact of a new microarchitectural feature.

MW is based on the Visualization-based Microarchitecture Workbench (VMW), which provides a graphical user interface. In MW, the graphical interface has been stripped and the core modules have been optimized for improved simulation speed. Currently, simulators for the following machines have been implemented using MW: Alpha AXP 21064 and 21164, PowerPC 604 [IBM94] and PowerPC 620. We have also started implementing an MW model of the Pentium. MW is now the primary simulation vehicle used in our research. On a complex microarchitecture design, such as the PowerPC 620, the MW model can run 10-20 times faster than the corresponding VMW model.

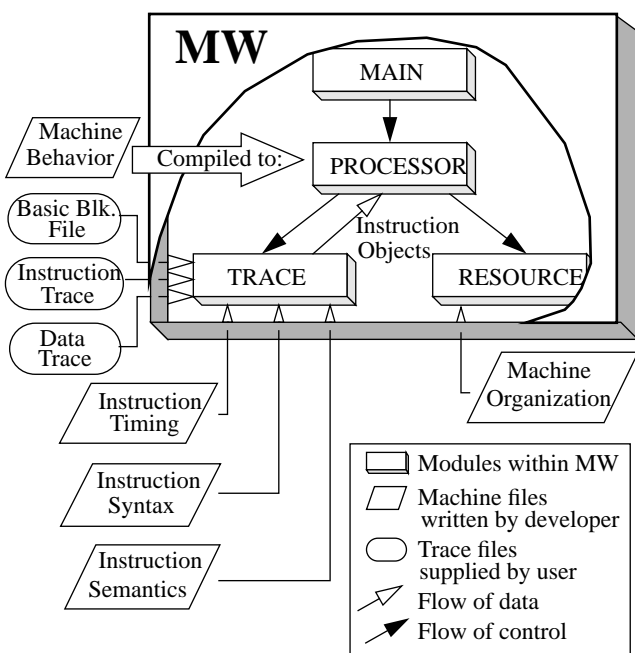


FIGURE 1. Major components of MW.

Figure 1 illustrates the major components of MW with an exploded view of the four primary modules inside MW. To build a simulator for a processor, the developer writes the five files represented by the rhombuses. The machine behavior file is written in C++; it is compiled and linked with other modules in MW to become part of the simulator.

When the simulator is executed, it goes through an initialization phase before starting the simulation. During this ini-

tialization phase, the other four machine files (in template form) are read by the `Trace` and the `Resource` modules. The machine organization file contains size and bandwidth information on the register files and instruction pipelines. Based on this information, the `Resource` module is able to determine all of the machine resources that are available to the executing instructions. The instruction timing file contains a cycle by cycle account of all resources used, read and written to by each instruction starting from the time that it is issued. This information is used during simulation to detect structural and data hazards. The instruction semantics file describes how operands and immediate values are to be extracted from each instruction. The instruction syntax file specifies the opcode fields to be used in instruction decoding.

When the simulation begins, the `main` module invokes the `Processor` module to simulate each cycle. The `Processor` module then asks the `Trace` module for the next instruction in the instruction trace. Using the three trace files, the `Trace` module is able to determine the next instruction in the instruction trace and its data address if the instruction is a load/store. The `Trace` module decodes the instruction using information from the instruction syntax file. It extracts the operands and immediate values using the instruction semantics file. It looks up the timing information for the instruction in the instruction timing file. When all this information about the instruction is located, the instruction is repackaged into an instruction object (`class InstrInfo`) and passed back to the `Processor` module.

The instruction object provides a simple interface through which the `Processor` module can access information that are necessary for simulation. For example, the instruction object can provide the instruction address for simulation of instruction cache. It can provide timing information, which contains the resource requirements for the instruction. The complete interface for `InstrInfo` is described in [HD95]. The `Processor` module uses the timing information to query the `Resource` module for the availability of the required resources. The resources may be unavailable due to data hazard or structural hazard. In that case, the `Processor` module will try to issue again in the next cycle. If the resources are available, the `Processor` module issues the instruction and informs the `Resource` module to mark those resources as used. This entire process is repeated until the end of the instruction trace.

2.3. Trace generation in MW

Currently MW has trace generation support for two architectures, the DEC Alpha and the IBM/Motorola PowerPC. For the Alpha we actually use the `ATOM` tool [SE94] from DEC. For the PowerPC we have developed our own trace generation tools. There is plan to develop a retargetable trace generation tool for MW. Typically the trace generation tool embeds instrumentation code into a program. When the instrumented program is executed the desired traces are produced.

There are four different ways to configure the interface between trace generation and simulation. One way is to generate the traces ahead of time and store them on disk. During simulation, disk files are accessed to obtain the traces. The second way is to generate the traces simultaneously with simulation and use the Unix file pipe mechanism to feed the traces to the simulator without storing the traces on disk. The third way is to implement trace generation and simulation as concurrent processes operating in the shared memory mode.

Traces are passed from one to the other via the shared memory. The fourth way to configure the interface between trace generation and simulation involves the use of the Unix socket mechanism across a local area network. We first look at the relative speeds of trace generation and simulation in terms of CPU time, before we discuss the four interface configurations in more detail. Table 1 lists the time that it takes to generate traces and perform simulation relative to the time it takes to run the original uninstrumented program. These measurements are taken on a DEC 3000/400 (which uses a 133 MHz AXP 21064) running a 21064 simulator (a relatively simple MW-generated model). For more complicated processors, the simulation time can be even higher, but the trace generation times should stay the same.

Table 1 Relative run times for trace generation and simulation.

		Run time relative to original executable
Trace generation:	File mode	500X - 1,000X
	Shared memory mode	50X
	Socket	50X - 100X
Simulation:		10,000X

Table 2 lists the four possible interface configurations between trace generation and simulation. In the disk configuration, the trace is generated only once and stored on disk. The problem with this method is the large amount of disk space required to store the traces. In the file pipe mode, two FIFO files (.itr and .dtr) are created on the local disk. The instrumented executable is run at the same time as the simulator. While one writes to the FIFO files, the other reads from them. This configuration requires no disk space, but the trace has to be generated for each simulation, increasing the total simulation time by up to 10%. In the shared memory configuration, the simulator sets up a region in memory as shared memory. The trace is generated for each simulation and is written to the shared memory. Because writing to shared memory is much more efficient than writing to the file system, generating the trace concurrently with the simulation adds less than 1% to the total simulation time.

Table 2 Four configurations for trace generation and simulation interface.

Configuration	# of trace generations for n simulations	Disk space required?	Simulation time is increased by
Disk	1	yes	0%
File pipe	n	no	10%
Shared memory	n	no	< 1%
Socket	n	no	0%

The shared memory configuration seems to be the ideal way to do simulation. Because the trace is generated as the simulation occurs, there is no need to store the trace. Also the amount of CPU time used by the trace generation process represents an insignificant portion of the total simulation

time. However this configuration imposes the constraint that the host machine (the one performing the simulation) must be of the same ISA as the target machine (the one that is being simulated). We have implemented and experimented with the first three configurations in the past.

Recently, we have implemented the fourth configuration that uses the Unix socket mechanism across a local area network. In this configuration trace generation and simulation occur simultaneously but on different machines on the network. The machine generating the trace (with the same ISA as the target machine) feeds the trace on the fly to the machine doing the simulation (can be of any ISA). Because the trace is generated on a machine other than the one that the simulation is running on, the trace generation does not add any additional time to the simulation. Currently we use a large number of Intel Pentium Pro (150MHz) platforms for the tedious simulation runs. Traces are generated by Alpha and PowerPC machines, each of which can simultaneously generate traces for multiple programs and feed the traces, via the socket code, to multiple Pentium Pro systems. By harnessing dozens of Pentium Pro systems on the network, we can carry out many long simulation runs simultaneously. Currently, this is our preferred configuration for simulation.

3. Machine model validation

Much like functional verification in the past, today's performance model validation is predominantly an ad-hoc process, with no rigorous methodology. Without rigorous validation, strong confidence cannot be placed on the results produced by these performance models. Robert Colwell says "watch out for artifacts and inaccurate results" or "don't always believe everything it says" when speaking of performance models [CL95]. In our machine model validation experiment, we adapt the standard industry functional verification methodology to performance model validation.

Typical functional verification involves billions of cycles of random test patterns, and man years of micro-test generation and coverage testing. All of these patterns have an expected functional result and are executed on the processor's hardware model throughout development. We propose leveraging this functional work to validate the processor's performance model. If the performance model were linked with the functional model, the engineer and random test generator would create the tests and the hardware model would provide an expected execution cycle time. This process verifies a high level (not necessarily cycle for cycle equivalent) performance model, that tracks with the design process.

The following subsections will demonstrate the success of such a validation methodology. Since we do not have access to functional models or the facilities to execute them, our functional model will be replaced with the actual hardware. If done correctly, embedded hardware counters provide equivalent results, with a more complicated interface but blazing execution times. The following discussion uses the PowerPC 604 as its microarchitecture base. The MW simulator tool is used to implement the PowerPC 604 microarchitecture model and a Power Series 850 (100 MHz) AIX system is used as the hardware reference.

3.1. PowerPC 604 simulation model

Our MW model is based on published reports on the PowerPC 604 [DS94, SDC94, IBM94] and accurately models all aspects of the microarchitecture, including branch prediction, fetching, dispatching, register renaming, out-of-order

issue and execution, result forwarding, the non-blocking cache hierarchy, alias detection, instruction refetching, and in-order completion. The 604 can fetch, dispatch, and complete up to four instructions per cycle, and can issue and execute up to six instructions per cycle.

3.2. PowerPC 604 hardware counters

The PowerPC 604 has a set of hardware counters called the performance monitor. [IBM94: section 9] provides a detailed description of the performance monitor features. There are two 32 bit counters (PMC1 and PMC2) and a 32 bit monitor mode control register (MMCR0). The MMCR0 register determines the signals to be monitored by each count register, and the conditions under which to increment the counters.

The performance monitor can count basic information such as cache miss, TLB miss, instruction dispatching, instruction finish, instruction completion, and load/store miss. A subset of the PMC1 and PMC2 counter options are outlined in Table 3.

Table 3 PowerPC 604 hardware counter options

Counter	Feature Description
PMC1 & PMC2	Processor Cycles
PMC1 & PMC2	Instructions Completed
PMC1 & PMC2	Instructions Dispatched
PMC1	Instruction cache miss
PMC2	Data cache miss
PMC1	Branch miss-prediction
PMC1	Length of load data cache miss service (Cycles) Threshold event
PMC1	Length of store data cache miss service (Cycles) Threshold event
PMC1	Number of integer instructions completed
PMC1	Number of floating point instructions completed
PMC2	Number of branches completed
PMC2	Number of loads completed

To configure the counters for a desired event the bit encoding of the specified event is inserted into the corresponding field of the MMCR0 register. In addition to counter control the MMCR0 register handles all threshold events, PMC2 count trigger, and contains the configuration for both the PMC1 and the PMC2 registers. Table 4 contains a detailed list of the MMCR0 configuration options.

The PowerPC 604 counters can be enabled to count supervisor mode, user mode, only processes that are marked by the MSR[PM] bit, and only processes that are not marked by the MSR[PM] bit. Combinations of these modes are possible. For example, one can observe all supervisor and user processes that are marked. The MSR[PM] bit is a special bit in the machine state register (MSR) that when set marks a process. Each processor has its own MSR value. Threshold events are events with different values. Table 3 has two threshold events, the load and store miss service times. The MMCR0 threshold field is set to a value and the PMC1 counter field configured to one of these two events. This creates a condition on the event such that the threshold condition

must be met before the counter is incremented. For example, setting the threshold to 20 and the PMC1 counter configuration to 9, the PMC1 counter will only count load misses that take 20 cycles or more to service. This feature allows the user to create a histogram of load miss service times. The final fea-

Table 4 MMCR0 bit settings

Bits	Description
0	Disable counting
1	Disable counting while in supervisor mode
2	Disable counting while in user mode
3	Disable counting while the MSR[PM] is set
4	Disable counting while the MSR[PM] is clear
10 - 15	Threshold value of 0 to 63
18	Trigger counting of PMC2 after PMC1 has become negative
19 - 25	PMC1 configuration
26 - 31	PMC2 configuration

ture of the MMCR0 register is the trigger of the PMC2 counter when the PMC1 counter goes negative. This allows the user to count events late in execution after another event has occurred. For example, the cache miss rate can be observed at a certain point of execution by counting the cycles until then and triggering the PMC2 counter at that moment.

3.2.1. Hardware counter interface

The PowerPC 604 hardware counters are implemented as special registers accessible only in supervisory mode. The "mfspr" instruction is used to modify the PMC1, PMC2, and MMCR0 registers, while the "mfspr" instruction is used to read the current state of these registers. Since these registers are accessible only in supervisory mode, a set of AIX dynamically-loadable pseudo devices are built to interface user code to the supervisor "mfspr" and "mfspr" instructions.

3.2.2. Benchmark instrumentation

A small library is created that provides "C" functions for reading and writing all the special registers used in the PowerPC 604 performance monitor. This library also includes data structures that define the counter options and counting configurations. To use the hardware counters this library is included in the compile options and a simple script instruments the "main function" of the benchmark source code. The instrumented code reads command line options to configure the counters and counting conditions. The source code is then encapsulated by a hardware counter initialization block and a termination block. The initialization block consists of simple library calls that preset and start the counters. The termination block stops the counting, reads the current state of the counters and writes the results to a specified file.

The instrumentation process can be easily adapted to encapsulate any desired portion of the source code. Each start and stop of the hardware counters introduces an overhead of 621 instructions using a Power Series 850 (100 MHz) PowerPC 604 system running AIX 4.1.3.

3.3. Validation tests

The validation methodology for the MW PowerPC 604 model has two sources for test code sequences. The first is a random test program generator, designed to produce a huge number of test instructions without human intervention. The second is a small set of hand generated test sequences, designed to target specific architectural features of the PowerPC 604 implementation.

3.3.1. Random test program generator

A random test program generator (RTPG) is built into the MW simulator to provide a validation test program source for MW models. The RTPG mode of MW requires the Instruction Syntax and Instruction Semantics files, described in section 2.2. These files completely specify the ISA of the simulation model. The RTPG mode produces a random set of instructions and outputs executable assembly code. The assembly code output is turned into an MW trace using any one of a number of possible tracing programs. The trace is then run as input for the simulation model, and results are compared with hardware execution.

Hardware execution is achieved using the AIX pseudo devices described in Section 3.2.1. Instead of instrumenting the code as described in Section 3.2.2., the RTPG mode inserts the randomly generated code directly into the device driver code. The device driver is compiled and installed into the OS for each random pattern. Although execution time is expensive, the hardware results have no overhead and produce exact cycle counts for comparison with the simulation model. The core of the device driver created by RTPG is

```
# Initialize registers
mtspr          PMC1, r0          # Initialize counter 1
sync           # Flush data memory system
isync          # Flush machine pipelines
mtspr          MMCR0, r4        # Start counters

# Random code sequence
mtspr          MMCR0, r5        # Stop counters
mfspr         r3, PMC1         # Retrieve count value
blr           # Return to the C portion of the device

# Data area for loads and stores follows
```

FIGURE 2. RTPG device driver core

illustrated in Figure 2. The random code sequence is bracketed by the start and stop instructions for the PowerPC 604 hardware counters. The “`mtspr MMCR0, r4`” instruction is functionally implemented in the PowerPC 604 simulator. The simulator records all performance monitor options (from Table 3 and [IBM94]: section 9) between the start and stop instructions. Once simulation is completed the recorded data is written to a file for the comparison. Modeling the “`mtspr MMCR0, r4`” functionally removes any discrepancy in fetching before and after the “`mtspr`” instructions.

On request RTPG can exclude any instructions in the ISA. Branches are not allowed to leave the program space and loop for a random amount of time less than a pre-programmed maximum. Load and store instructions load random data from an allocated data space following the device driver code. Future RTPG work will involve biasing the random patterns to exercise specific portions of the micro-architecture implementation.

3.3.2. Handwritten micro test programs

In any comprehensive functional validation methodology, engineers are required to create micro tests that target specific microarchitecture features. These tests are usually designed to exercise boundary conditions and obscure functional states that a random test generator may not exercise. These tests are also used to improve “test coverage” by stimulating certain signals in the functional implementation. As with functional models, performance models have boundary conditions and obscure state based behavior. Proper validation of a performance model must include these micro tests.

The validation of the PowerPC 604 simulator included several micro test sequences. These test sequences are mostly branch and load/store tests, designed to exercise the branch paradigm and memory hierarchy. These tests are inserted into the performance monitor device drivers. Hardware counter statistics (including: branch misprediction, load miss latencies, Icache miss, and Dcache miss counts) are gathered to pinpoint the behavior of the hardware.

3.4. Validation results

To verify the effectiveness of the validation process a small set of benchmarks are executed on the hardware system as well as simulated with the PowerPC 604 model. Obviously, the smaller the delta between the two cycle counts the more accurate the simulation model. This process has two sources for error. The first involves the actual sequence of instructions executed. The instructions in the simulated trace may not be identical to the sequence of actual instructions executed on the hardware. The second is due to the inaccuracy involving the embedded counters. The hardware counters may count other instructions executed that are not part of the benchmark code. The closer correlation between the *instruction counts* of the simulated trace and the executed trace will ensure a more reliable and correlated *cycle counts* on both the simulator and the hardware.

3.4.1. Hardware to trace correlation

Dynamic instruction count is a microarchitecture independent metric that is common to both the hardware execution and trace simulation. It is assumed, if the instruction count is the same between a hardware execution and a trace simulation, that the trace has accurately captured the runtime effects of the benchmark. It also follows that the hardware counters counted only the execution of the benchmark code. The benchmarks used to verify this validation process are listed in Table 5. The table also includes the instruction counts for both the hardware execution and the trace driven simulation. These numbers demonstrate that there is strong correlation between the two instruction counts. There are three sources of error that can account for these small discrepancies.

Tool overhead:

Both the hardware counters and the trace gathering tool have a fixed overhead of 621 and 557 instructions respectively. This overhead offsets and effectively adds 64 instructions to the hardware instruction counts.

Trace gathering:

The trace gathering introduces an unknown element to the instruction and cycle counts. The tracing tool used in this study is designed to trace library calls, however it is unclear how extensive the trace reaches into each library call. The hardware counters count all user processes up to the point

they switch to supervisory mode. We expect some additional instructions in the hardware count due to this difference.

Hardware system interrupts:

The PowerPC 604 performance monitor allows the user to mark processes and count only those processes. However, the AIX operating system masks the machine state register bit which marks the process. We are forced to run the benchmark executions counting all user process. Therefore, the instruction and cycle counts may include interrupts for other user processes. All hardware executions are performed in single user mode to reduce this effect. Unfortunately, the counts still varied slightly. This introduces an unknown perturbation on the instruction counts in Table 5.

Table 5 Benchmark instruction counts

Benchmarks	Length (Instructions)		%
	Hardware	Trace	
cjpeg 128x128 BW image	2,771,141	2,771,012	-0.02
eqntott (SPEC92) Modified reference input	18,866,003	17,903,424	-5.1
grep grep -c "st*mo" 1/2 SPEC92 compress input	2,315,408	2,315,201	-0.01
gperf gperf -a -k 1-13 -D -o Scrabble 200 word dict.	7,819,185	7,817,130	-0.03
mpeg Berkeley MPEG decoder 4 frames w/ dithering	9,039,253	9,039,010	0.0
quick Sort of 5000 rand. Elem.	739,022	738,895	-0.02

Supporting the above stated explanations for the instruction count discrepancies, the hardware counter instruction counts are consistently greater than the simulation traces. With the exception of the eqntott benchmark there is strong correlation between the two counts. Therefore, we expect very reliable cycle count results for cjpeg, grep, gperf, mpeg, and quick. The 5.1% difference in instruction counts for the eqntott benchmark is unexplained at this time, but must fall under one or more of the error conditions mentioned above.

3.4.2. PowerPC 604 simulator results

Table 6 summarizes the cycle count results for the benchmarks. All simulation cycle counts for the benchmarks with strong correlation of instruction counts are within 5.32% of the actual hardware cycle counts. Again eqntott shows a significant error. At this time it is not well understood why the eqntott cycle counts are 27% off. The 5% instruction count error is not significant enough to account for this error.

The results in Table 6 show promise for this new methodology, but are not conclusive. At the time of publication this new validation methodology, borrowing from functional verification techniques, is in its infancy. The RTPG tool is still finding errors in the simulation model, and is unable to

Table 6 Benchmark cycle counts

Benchmark	Cycle Count		Discrepancy in %
	Hardware	Simulation	
cjpeg	2,686,552	2,758,667	2.68
eqntott	17,660,335	12,837,171	-27.31
grep	2,774,036	2,768,619	-0.2
gperf	6,452,720	6,796,622	5.32
mpeg	8,182,928	7,797,112	-4.71
quick	775,600	738,895	-4.73

log large numbers of successful, i.e. perfectly matching, instruction executions between model versions. Further development of this methodology is ongoing, and its final results will include a larger diverse set of benchmarks and micro tests, along with significant RTPG cycles. These results are expected and will be presented at the conference.

4. Trace sampling inaccuracy

Various trace sampling techniques (e.g. [ITB96]) have been proposed to reduce both the time and space requirements of trace-driven performance modeling. These techniques attempt to capture the behavior of the original program trace in a smaller, shorter trace that takes less space and time to store and process, without sacrificing too much simulation accuracy. To evaluate the feasibility of trace sampling, we collected detailed intermediate simulation results from our PowerPC 620 model [DNS95] for some of our longer traces and qualitatively analyzed them for amenability to two commonly-used approaches to trace sampling: truncation and time-domain sampling.

4.1. PowerPC 620 simulation model

Our model is based on published reports on the PowerPC 620 [DNS95, LTT95], and accurately models all aspects of the microarchitecture, including branch prediction, fetching, dispatching, register renaming, out-of-order issue and execution, result forwarding, the non-blocking cache hierarchy, alias detection, instruction refetching, and in-order completion. The 620 can fetch, dispatch, and complete up to four instructions per cycle, and can issue and execute up to six instructions per cycle.

4.2. Trace sampling results

Table 7 Trace sampling benchmarks

Name	Description	Length (# instr)	IPC	Branch Mispred
cc1	SPEC92 gcc 1.35	146.1M	1.24	5.1%
compress	SPEC92 1/2 of in	38.8M	1.16	8.8%
perl	SPEC95 tiny	105.2M	1.02	4.3%
sc	SPEC92 short	78.5M	1.26	1.9%
swm256	SPEC92 5 iter	43.7M	0.85	0.1%
xlisp	SPEC92 6 queens	52.1M	1.16	5.1%
Total		464.5M		

To evaluate trace sampling, we collected the branch misprediction rate and the sustained instructions-per-cycle

(IPC) metric for the six benchmarks summarized in Table 7 at intervals of two million cycles, and plotted both the cumulative average and the average for just the last two million cycles against time

The results are plotted in Figure 3 and Figure 4. For all

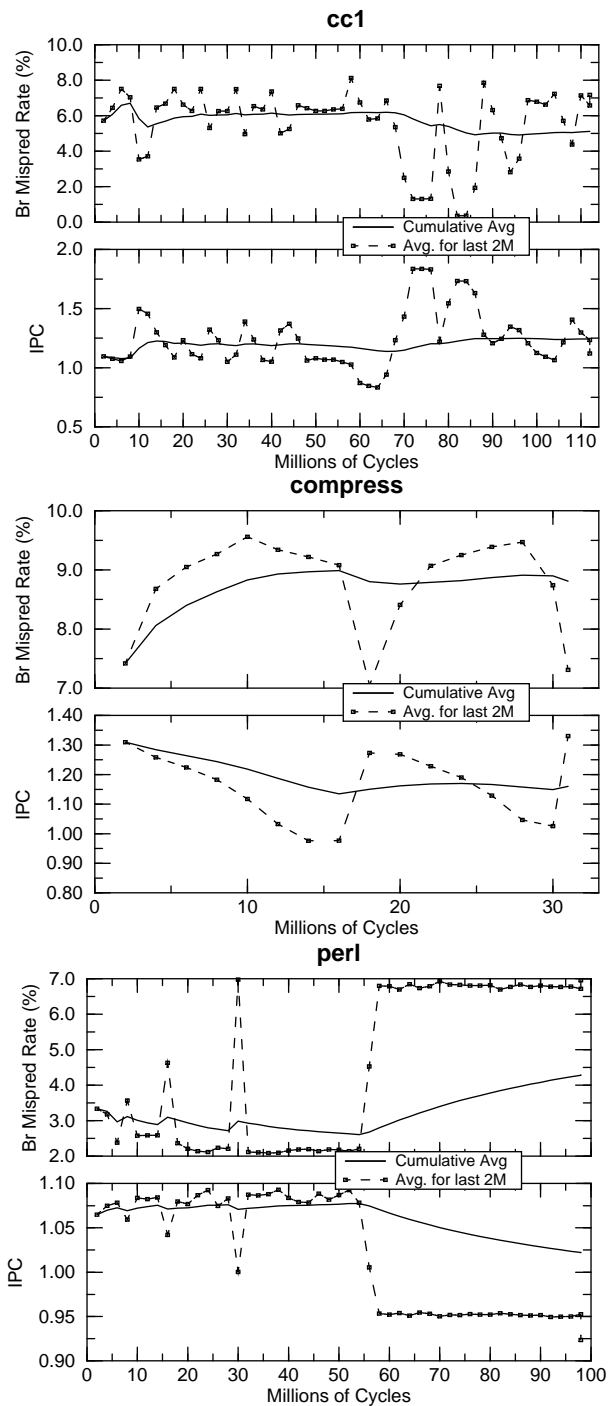


FIGURE 3. Results for cc1, compress, and perl

benchmarks, we see that one or both metrics vary significantly throughout the execution of the program. This in itself is discouraging for trace sampling, since none of the bench-

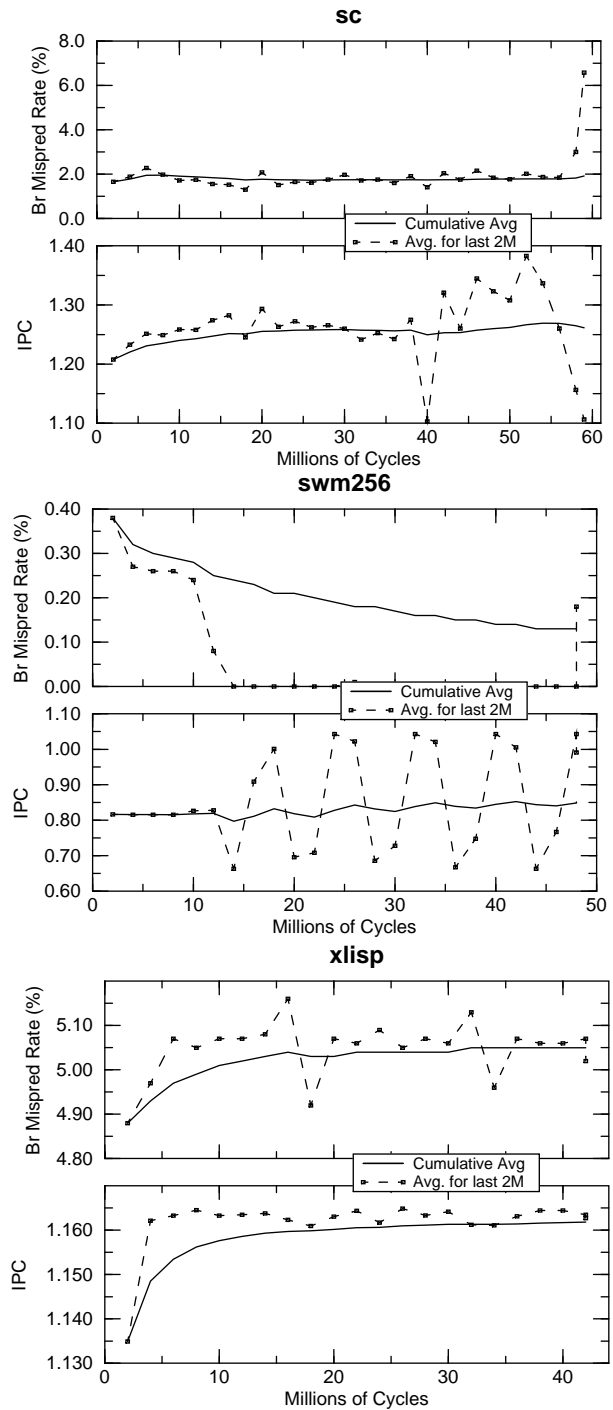


FIGURE 4. Results for sc, swm256, and xlisp

marks appear to reach a dominant steady state that could be used to approximate the behavior of the whole trace.

Three of the benchmarks--cc1, compress, and swm256--demonstrate periodic behavior in one or both metrics. For cc1, this behavior corresponds with the number and size of the procedures in the source input file used (gcc compiles one procedure at a time). Clearly, truncating the trace before the largest procedure (recog_6) could severely distort results, while time-domain sampling would be subject to interfer-

ence due to the periodic nature of each metric.

The periodic nature of compress can be attributed to the compression algorithm in use: it fills a table until it reaches a certain size, then starts over with a new one (this happens at around 16 million cycles). Again, both truncation and trace sampling could severely distort results.

The third benchmark that display periodic behavior is swm256. Here, the behavior corresponds to five iterations of the main loop. Again, the same pitfalls apply to truncation and trace sampling.

The results for perl clearly match two main phases of computation: first, a large dictionary is loaded into an internal structure, then this dictionary is searched for anagram matches. Here, truncation would fail badly, while sampling, if applied carefully to include both phases, could be quite successful.

The remaining two benchmarks, sc and xisp, are relatively well behaved, and could be quite amenable to both truncation and sampling. However, even sc shows significant variations in both metrics towards the end of the program (these would not be accounted for with truncation), while xisp displays some periodic branch-misprediction behavior that could impact time-domain sampling.

In summary, these results demonstrate that arbitrary use of either trace truncation or time-domain sampling can lead to significant inaccuracies for both IPC and branch-misprediction metrics. Neither approach, unless carefully and systematically applied, is a reasonable substitute for simulating with the full trace.

5. Conclusions

We started out developing trace-driven performance simulators out of necessity in order to carry out our superscalar microarchitecture research. We soon discover that this task is a research topic in its own right. We have learned a great deal about microarchitecture performance simulation and believe that there is a great deal more work to be done, including more rigorous experimental research.

Chronologically we developed our PowerPC 620 model first. It was a frustration not being able to rigorously validate our 620 model against a hardware model or an actual hardware machine due to their unavailability. We then developed our PowerPC 604 model for the purpose of validating it against a PowerPC 604 machine, which was available. Given the MW framework, the 604 model took one person about two months to write. We have spent another two months of debugging up to the time that this paper was submitted for publication. The 604 validation results presented in Section 3 are only preliminary. We expect that the 604 model will be further fine tuned and more benchmarks will be processed. In any case, the results so far seem to indicate that trace-driven performance simulators can model actual hardware performance with reasonable accuracy. There is also the trade-off between model development time and model accuracy.

Because machines are getting much faster than standard benchmarks, e.g. SPEC, are also evolving to run for longer times involving much longer traces. For simulations with an overhead of four possibly five orders of magnitude, these extremely long traces become problematic for average researchers. In Section 4 we explore the possible inaccuracies that can result from trace sampling. Our sense is that arbitrary samplings (quite frequently done in published papers) are very questionable. How would one go about validating the representativeness of a sampled trace without the

data from the full trace as a reference? Running unvalidated sampled traces on unvalidated performance models generates results that can easily be considered as bogus.

Acknowledgments

The research efforts presented here have been supported by NSF (CCR 9423272) and ONR (N000149610347). Andrew Huang was supported by an IBM Graduate Fellowship. We have also benefited from the generous donation of a large number of Pentium Pro Systems from Intel to the computer engineering group. We would like to give special thanks to Marvin Denman at Motorola for sharing his knowledge of the PowerPC 604 microarchitecture, and Prithvi Rao of CMU for building the AIX pseudo device driver interface used in the MW PowerPC 604 validation process. We thank the session organizer, Pradip Bose of IBM, for inviting us to contribute this paper.

References

- [BK95] Peter Bannon and Jim Keller. Internal architecture of Alpha 21164 microprocessor. *COMPCON 95*, 1995.
- [CL95] Robert P. Colwell and Conrad Lai. Eternal Vigilance is the Price of Performance. "Pre-Silicon Performance Simulation" Workshop, *ISCA 1995*
- [Dig92] Digital Equipment Corporation. *DECchip 21064-AA Microprocessor Hardware Reference Manual*, 1992.
- [DNS95] Trung A. Diep, Christopher Nelson, and John P. Shen. Performance evaluation of the PowerPC 620 microarchitecture. In *Proceedings of the 22nd International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.
- [DS93] Trung A. Diep and John Paul Shen. EXPLORER: A retargetable and visualization-based trace-driven simulator for superscalar processors. In *Proceedings of the 26th Annual ACM/IEEE International Symposium on Microarchitecture*, 1993.
- [DS94] Keith Diefendorff and Ed Silha. The PowerPC User Instruction Set Architecture. *IEEE Micro*, pages 30-41, 1994
- [DS95] Trung A. Diep and John Paul Shen. VMW: A visualization-based microarchitecture workbench. *IEEE Computer*, 28(12):57-64, 1995.
- [Gro90] G.F. Grohoski. Machine organization of the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1):37-58, January 1990.
- [HD95] Andrew S. Huang and Trung A. Diep. MW developer's guide. August 1995.
- [IBM93] IBM Microelectronics Division, Essex Junction, VT. *PowerPC 601 RISC Microprocessor User's Manual*, 1993.
- [IBM94] IBM Microelectronics Division. *PowerPC 604 RISC Microprocessor User's Manual*, 1994.
- [ITB96] Vijay S. Iyengar, Louise H. Trevillyan, and Pradip Bose. Representative traces for processor models with infinite cache. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture*, pages 62-72, San Jose, California, February 3-7, 1996. IEEE Computer Society TCCA.
- [LTT95] David Levitan, Thomas Thomas, and Paul Tu. The PowerPC 620 microprocessor: A high performance superscalar RISC processor. *COMPCON 95*, 1995.
- [SDC94] S. Peter Song, Marvin Denman, and Joe Chang. The PowerPC 604 RISC Microprocessor. *IEEE Micro*, pages 8-17, 1994
- [SE94] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools", *Proc. of PLDI*, 1994.