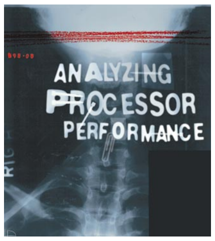
Cover Feature

Calibration of Microprocessor Performance Models



Microprocessor designers use performance models to predict performance and guide their design decisions. Experimental results in this article highlight the significant effort required to achieve a truly cycle-accurate performance model and the necessity for systematic validation of performance models to ensure their accuracy and usefulness.

Bryan Black
John Paul Shen
Carnegie Mellon University

The microprocessor industry can capably predict the clock frequency and functionality of first silicon—the first small set of chips made for a new design. However, predicting first silicon's *performance* on real programs remains a challenge.

Designers use performance models during the development of microprocessors to predict the average number of executed instructions per cycle (IPC). IPC measures instruction throughput and is a key factor in the overall performance of a microprocessor. Performance models, which are usually implemented in a high-level language such as C or C++, measure instruction throughput. They do so by capturing the timing essence of instruction execution; that is, by modeling the latency of instructions, the dependencies between instructions, and the allocation of limited resources in the microprocessor implementation.

Microprocessor designers use performance models to assist in design decisions.[1] These models allow the designer to evaluate new ideas, without the cost of fully implementing them in hardware or in a hardware description language, such as Verilog or VHDL (very high speed IC hardware description language). Often the validation of such performance models is not an algorithmic process and relies largely on "validation by inspection."

SOURCES OF ERROR

As in all design efforts, performance modeling is susceptible to many sources of error, including but not limited to modeling, specification, and abstraction errors. *Modeling errors* occur when the developer understands the modeling task but incorrectly codes the desired functionality. *Specification errors* occur when the developer, misinformed about the correct functionality, models the wrong behavior. To keep performance models simple and fast, the details of some features are abstracted. An *abstraction error* occurs when the developer implements a feature at a higher level of abstraction without maintaining equivalent instruction timing. Abstraction errors also result from features that are not implemented but end up having significant impact on instruction execution timing.

VALIDATION BY INSPECTION

Much like functional validation of hardware in the past, validating performance models today relies mostly on inspection. The programmer-intensive inspection/validation process involves analyzing the simulation process and performing sanity checks on the simulation results.

Inspection may require single stepping through the execution cycles of small code sequences and observing the performance model's internal state. The developer bears responsibility for sighting incorrect behavior as the code sequences execute. Sanity checking uses an array of instrumentation counters that are built into the performance model. These counters gather statistics, such as the IPC, cache miss rate, instruction dispatch rate, and the average and peak resource usage. They also provide summary statistics about the behavior of a code sequence.

Developers devise simple code sequences to exercise the boundary conditions of all resources in the performance model. Simulating these code sequences and analyzing the resulting counter statistics provide clues to finding incorrect behavior.

Longer code sequences and benchmarks are also used for sanity checks. Designers run simulations of large benchmarks, using different microarchitecture configurations for each run. They then analyze the counter statistics. If they find the expected behavior, they gain increased confidence in the performance model. If the counter statistics do not match intuitive expectations, and designers cannot determine an explanation, they search the performance model for possible errors.

Performance model validation involves generating test cases, stimulating the model under test, and comparing execution results to a known reference.

Although these inspection techniques effectively debug a model, without a systematic validation method, it is difficult to place a high level of confidence on the results produced by these performance models. Our calibration of a performance model against actual hardware demonstrates the risks of relying solely on inspection-based validation. We propose a systematic method for calibrating a performance model against a register transfer level (RTL) implementation of the microprocessor—a gate-level model of the hardware. Our results suggest the potential effectiveness of the proposed calibration method.

PERFORMANCE MODEL DEVELOPMENT

We chose a performance model for the IBM/Motorola PowerPC 604 as an experimental vehicle. The model was implemented in the Microarchitecture Workbench (MW),[2-4] a performance-modeling environment developed at Carnegie Mellon University. MW provides a framework that minimizes the amount of work required to develop a performance model and the number of changes necessary to evaluate new microarchitecture features. MW has been used to model several microprocessors, including the Digital Alpha 21064 and 21164 and the PowerPC 601, 604, and 620. Carnegie Mellon has distributed MW to several other universities and commercial companies.

The PowerPC 604[5-7] is a superscalar microprocessor capable of out-of-order execution. It is a *four-wide processor*, which means it can fetch, dispatch, and complete up to four instructions per cycle. Because the 604 has six execution units, it can issue (that is, initiate execution) and finish the execution of up to six instructions per cycle.

Our performance model can capably model all the key microarchitecture features of the PowerPC 604, including branch prediction, instruction fetch and dispatch, register renaming, out-of-order instruction issue and execution, execution result forwarding, the nonblocking cache hierarchy, load/store alias detection, instruction refetching, and in-order completion.

Because the model is trace-driven, it does not actually simulate mispredicted branches. It does accurately model the penalty cycles due to a misprediction because the 604 performs a pipeline flush (it empties instructions from the pipeline and starts over) on a branch mispredict. However, our performance model does not account for potential instruction cache pollution caused by loading the cache with instructions on the mispredicted path.

PERFORMANCE MODEL VALIDATION

Validation involves generating test cases, stimulating the model under test, and comparing execution results to a known reference. Functional validation in microprocessor development validates the hardware structural model (RTL model) against a known behavioral model. A behavioral model simulates the instruction set execution of a microprocessor. Because such validation is limited by the test cases executed, designers expend great effort to generate a wide variety of test suites. Test suites in functional validation range from explicitly and randomly generated instruction sequences to those that are handwritten. We adopted existing functional validation techniques and associated test suites to validate our performance models. By design, functional-validation test suites exercise the functionality of an RTL model, although they neglect instruction timing. However, these same test suites have instruction execution timing results that functional validation simply does not verify. We used these same test suites to validate the timing of instruction execution in a performance model.

Performance model validation reference

All validation methods require a reference to determine if a test sequence passes or fails. We focused on validating the correctness of the PowerPC 604 performance model implemented on the MW. The ideal reference is the RTL model developed during the implementation of a new microprocessor. Unfortunately, we do not have access to the RTL model of the PowerPC 604. Instead of the RTL model, we used the actual hardware as the reference machine. More specifically, we chose an IBM Power Series 850 AIX 4.1.3 system as the reference machine. We can use an actual machine because the PowerPC 604 chip provides a set of hardware-embedded counters. These counters provide adequate observability into the execution of code—to extract cycle counts, instruction count, completion rate, and other statistics.

PowerPC 604 hardware counters. The set of hardware-embedded counters is called the performance monitor,[7] which in the PowerPC 604 includes two 32-bit counters and a 32-bit control register. The two counters count *events* during execution, and the control register determines which event each counter will monitor. The PowerPC 604's performance monitor can count many events, such as misses in the cache and translation look-aside buffer (TLB), instruction dispatches, instruction finishes, instruction completions, and load/store miss latencies. Different event-counting modes allow the user to count only supervisor code, user code, or specially marked processes.

Hardware counter interface. The hardware counters are implemented as special registers accessible only in supervisory mode. Special supervisor instructions provide read/write access to the counter and control registers. Because these registers are accessible only in supervisory mode, we developed a set of AIX (Advanced Interactive Executive, IBM's Unix) dynamically loadable pseudodevices to interface user code to the supervisor instructions. The header of the pseudodevice initializes register state, flushes the machine pipeline, and then configures and starts the performance monitor counters. Execution of the test sequence begins, and a trailer stops the counters and returns the count results to the calling function. This device provides a clean interface to the performance monitor at the cost of a small, fixed overhead.

Performance model validation test patterns

We defined five different test suites to target specific portions of the PowerPC 604 performance model. *Alpha tests* exercise instruction latency by executing each instruction one at a time. *Beta tests* check pipeline dependencies within an instruction type—for example, data forwarding and register renaming—by executing each instruction two to 100 times back to back. *Gamma tests* execute each instruction next to every other instruction, testing pipeline dependencies between instruction types and data forwarding across functional units. *Random test sequences* of up to 100 instructions exercise the interactions among instructions and the different components of the microarchitecture. Finally, *handwritten patterns* test microarchitecture features for which tests are difficult to generate automatically or those not sufficiently covered by random sequences.

These five test suites are only a small portion of the functional validation process used by IBM and Motorola for PowerPC microprocessors. A complete validating process involves dozens of automatically generated and specially directed test suites and years of random test sequence execution. Although we use only the five test suites to illustrate the need for performance model validation, the calibration method suggested by our work advocates the use of all functional validation test suites for performance model validation.

Automatic test suite generation. The alpha, beta, and gamma test suites are generated by an automatic test pattern generator (ATPG) that is built into the MW framework. The ATPG tool extracts information about the instruction set architecture from the performance model and generates executable code sequences that fall into the alpha, beta, and gamma categories of test sequences.

The ATPG also generates executable random code sequences. All source and destination operands are randomly selected with the exception of branch instructions and load/store instructions. Branches are not allowed to leave the program space and loop for a random amount of time less than a preprogrammed maximum. Load/store instructions access random data locations from an allocated data space.

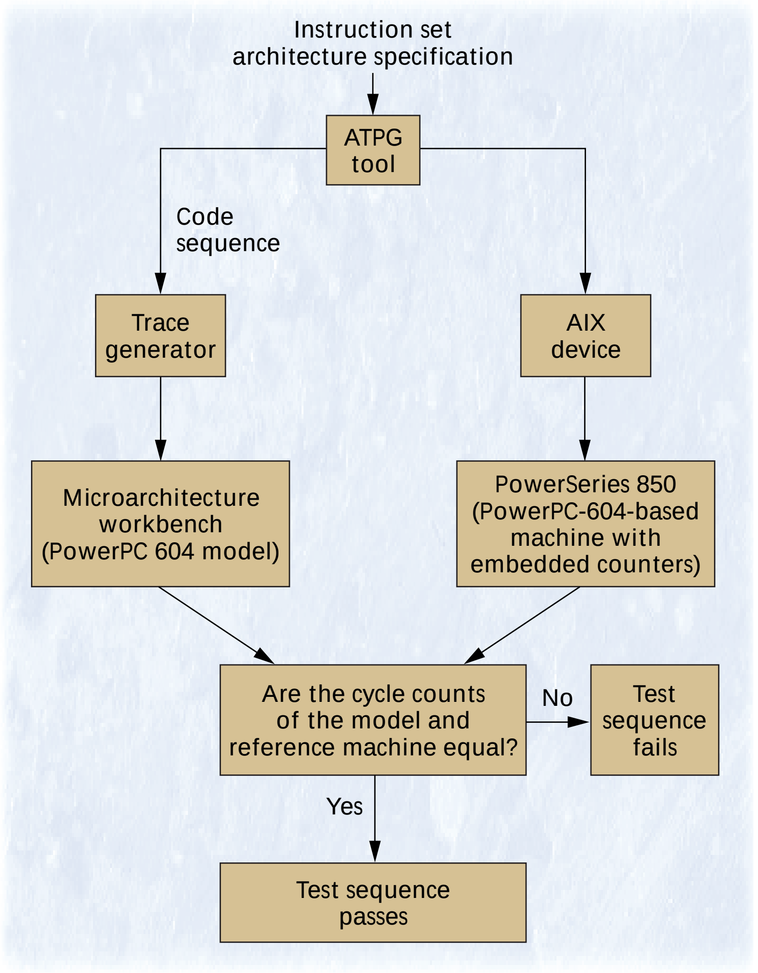

Instruction set architecture specification
ATPG tool
Code sequence
Trace generator
AIX device
Microarchitecture workbench (PowerPC 604 model)
PowerSeries 850 (PowerPC-604-based machine with embedded counters)
Are the cycle counts of the model and reference machine equal?
No
Test sequence fails
Yes
Test sequence passes

Figure 1. Performance model validation process.

Handwritten test suite. Handwritten tests are usually designed to exercise boundary conditions as well as obscure functional states that a random test generator may not exercise. These tests improve test coverage by stimulating certain signals in the hardware implementation. As with hardware models, performance models have boundary conditions and behaviors involving obscure states. Proper validation of a performance model must include handwritten tests.

The validation of the PowerPC 604 performance model includes several such 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 pinpoint the hardware's behavior.

Putting it all together

Figure 1 illustrates the experimental framework used to validate the PowerPC 604 performance model. The ATPG tool directly accesses the MW instruction set architecture specification files and generates executable code sequences for use in validation. These code sequences are pushed through a trace generator and the resulting trace is executed on the MW performance model, which yields an execution cycle count. Concurrently, we add the same code sequences

Table 1. Infant model validation results.

Pattern type	Instruction count	No. of failing tests*	Passing tests (percentage)
Alpha	245	120	51.0
Beta	2,940	2,040	30.6
Gamma	29,890	29,890	0.0
Handwritten	500 (approximately)	500 (approximately)	0.0
Random sequences**	N/A	N/A	0.0

* Pass/fail based on a zero-cycle difference.

** Random sequence numbers were never recorded due to 100-percent failure.

Table 2. Samples of errors found by applying the proposed method to the infant model.

Error type	Description
Specification	Floating-point multiply-add record instructions need a latency of six cycles and reserve all pipeline stages (documented as three cycles).
	Complex instructions that modify the overflow bit need to take one extra cycle (not specified).
	The Isync instruction should wait for both the completion buffer and the write-back buffer to empty before execution.
	The Isync instruction needs a latency of five cycles (documented as one cycle).
Modeling	Mtfsb0 and mtfsb1 dispatching need to dispatch to the floating-point unit and not the complex integer unit.
	Instruction finish adds an extra cycle to latency and should be part of the last cycle of execution.
	Load instructions execute in a single cycle; it should be two cycles.
	The complex integer unit is not fully pipelined.
	An incorrect number of branches exposed to prediction during decode and dispatch.
	Branch execution corrects later branch mispredictions.
	Reservation stations issuing out-of-order did not have age influence.
	Load with update should forward update results after address generation.
	The data cache reloads the cache line immediately after a miss should stall for memory latency.
Abstraction	Branch misprediction paths are not simulated.
	Data-dependent execution is not simulated.
	The memory hierarchy assumes a perfect level-two cache.
	The PowerPC 604 bus unit is modeled as a delay and not as a complex state machine.

to the AIX pseudo-device described earlier. This AIX device brackets each code sequence with embedded performance counter control. A read of the device executes the code sequence on an actual PowerPC 604 system. The AIX device returns the number of execution cycles on the actual hardware. If both cycle counts are equal, the test sequence passes. This framework produces large numbers of automatically generated sequences, which can be strictly random or directed toward certain microarchitectural features. We then measured the accuracy of the PowerPC 604 performance model by the passing percentage of test sequences.

INFANT MODEL VALIDATION

The initial MW-based performance model we developed is called the *infant model*. To illustrate the need for systematic performance model validation, we validated the infant model by inspection only. Using inspection over a long debugging period, we found and repaired many bugs. During this time, we ran single-stepping studies of the simulation process and sanity checks on the simulation results.

It was thought that the resultant infant model would accurately model the PowerPC 604, and continuing the debugging process should find the few remaining bugs. After this validation by inspection, we applied our proposed systematic performance model method, which uses the five test suites. Table 1 shows the results of applying these test suites to the infant model. A failure is any test for which there is a difference between the cycle counts of the performance model and the hardware reference machine.

The infant model failed badly on most of the test suites. It correctly modeled the latency (alpha tests) of only 51.0 percent of all the instructions. Even worse, it correctly modeled instruction pipelining (beta tests) for only 30.6 percent of all the instructions. The gamma tests, handwritten tests, and random sequences had a 100 percent failure rate. Given such poor results, there can be little confidence in the results produced by this infant model.

CHILD MODEL VALIDATION

We used the test suite results to debug the infant model. After this debugging process, we called the infant model a *child* to reflect its status as a more mature performance model. Bugs infected every aspect of the microarchitecture and ranged from incorrect instruction latencies to instructions dispatching to the wrong functional units. Table 2 lists some of the more interesting errors found during validation. Table 2 clearly shows that errors of all types play a role in the accuracy of performance models. Such a wide range of failures indicates the need for a systematic validation method. The significant number of modeling errors further demonstrates that inspection methods will not fully debug a performance model. Table 3 shows the number of tests or test sequences in each test suite, with the current number of passing and failing sequences.

Accurate modeling of instruction latency (alpha tests) increased from 51.0 to 95.9 percent. Pipeline modeling (beta tests) improved from 30.6 to 75.4 percent. When we relaxed our criteria and allowed a sin-

Table 3. Child model validation results.

Pattern type	Instruction count	Zero-cycle difference: No. of failing tests	Zero-cycle difference: Passing tests (percentage)	One-cycle difference: No. of failing tests	One-cycle difference: Passing tests (percentage)
Alpha	194	8	95.9	0	100.0
Beta	20,358	5,017	75.4	3,105	84.7
Gamma	60,552	12,244	79.8	11,005	81.8
Handwritten	about 500	0	100.0	Not applicable	Not applicable
Random sequences	93,779	63,782	32.0	26,688	71.5

Table 4. Benchmark instruction counts.

Benchmarks	Input	Length (no. of instructions): Hardware	Length (no. of instructions): Trace	Difference (percentage)
cjpeg	128 × 128-pixel black-and-white image	2,771,141	2,771,012	–0.02
eqntott	SPEC92 modified reference input	18,866,003	17,903,424	–5.10
gperf	-a -k 1-13 -D -o scrabble 200-word dictionary	2,315,408	2,315,201	–0.01
grep	-c "st*mo" 1/2 SPEC92 compress input	7,819,185	7,817,130	–0.03
mpeg	Four frames with dithering	9,039,253	9,039,010	0.00
quick	Sort of 5,000 random elements	739,022	738,895	–0.02

gle-cycle difference to be a passing test, 100 percent of the alpha test and 84.7 percent of the beta tests passed. The gamma tests showed a pass rate of 79.8 percent for a zero-cycle difference and 81.8 percent for a single-cycle difference. Random sequences began passing on the child model, with a passing rate of 32.0 percent for a zero-cycle difference and 71.5 percent for a single-cycle difference. The results of this validation process clearly reveal that achieving a reliable and accurate performance model requires significant effort. Even our much-improved child model requires further validation to ensure that it can accurately predict performance.

ANALYSIS OF VALIDATION RESULTS

Our results demonstrate the effectiveness of the proposed validation method at finding bugs in performance models. However, to verify its effectiveness at improving model accuracy, we had to test the performance model with longer code sequences. We therefore selected a small set of real benchmarks to run on the reference machine as well as on the infant and child performance models. Obviously, the smaller the difference between cycle counts of the reference machine and the performance model, the more accurate the performance model. We executed real code on the hardware, extracted traces of the code execution, and ran the traces on both the infant and child performance models. The primary source of error in this experiment lay in trace gathering. Does the tracing tool accurately trace the benchmarks' runtime execution?

Benchmark instruction count correlation

Dynamic instruction count is a microarchitecture-independent metric common to both the hardware execution and the trace simulation. We assumed that if the two had the same instruction count, the trace accurately captured the runtime execution of the benchmark. Table 4 lists the benchmarks used to verify this validation process along with their input data sets. The table includes the instruction counts for both the hardware execution and the trace-driven performance simulation. These numbers demonstrate a strong correlation between the two instruction counts. Two sources of error can account for the small discrepancies:

- *Tool overhead.* The hardware counters and the tracing tool have fixed overheads of 621 and 557 instructions. The overhead results from extra function calls and code to set up the hardware counters and trace gathering. However, these constant fixed overheads are comparable and resulted in the net addition of 64 instructions to the hardware instruction counts.
- *Trace gathering.* The tracing tool we used is designed to trace library calls, however, it is unclear how far the trace extends into each library call. The hardware counters count a user process up to the point it switches into supervisor mode. Some additional instructions in the hardware count result from this discrepancy.

The fact that the hardware counter's instruction counts are consistently higher than those of the simulation traces supports these assertions. With the exception of the eqntott benchmark, a strong correlation exists between the two instruction counts. Therefore, we would expect the performance models to yield reliable cycle count results for cjpeg, grep, gperf, mpeg, and quick.

Table 5. Benchmark cycle count results.

Benchmarks	Hardware reference model	Infant model	Child model	Infant discrepancy (percentage)	Child discrepancy (percentage)
cjpeg	2,686,552	2,758,667	2,629,786	2.68	–2.11
eqntott	17,660,335	12,837,171	15,985,200	–27.30	–9.49
grep	2,774,036	2,768,619	2,968,911	–0.20	7.02
gperf	6,452,720	6,796,622	6,554,997	5.32	1.56
mpeg	8,182,928	7,797,112	8,000,382	–4.71	–2.23
quick	775,600	738,895	799,504	–4.73	3.08

Benchmark cycle count correlation

Table 5 summarizes the cycle counts from the benchmark executions and performance model simulations. As expected, the simulation cycle counts for five of the benchmarks (all except eqntott) correlate well—within 5.32 percent for the infant model and 7.02 percent for the child model. Eqntott shows a significant error for the infant model. It contains a significant load/store bug which we discovered early in the validation process for the child model. This accounts for eqntott's extremely poor results on the infant model and the improved results on the child model.

Final analysis

Debugging the infant model using the proposed validation method improved the child model's accuracy, reducing the cycle-count difference on these benchmarks from an average of 7.49 to 4.25 percent. So applying our validation method improves both the accuracy of and our confidence in the performance model. However, Table 5 shows that for individual benchmarks the percentage of error changed drastically from the infant to the child performance model. For cjpeg, grep, and quick, the error even changed sign. For grep, the magnitude of the error actually increased.

These interesting data points conflict with what we typically expect of the performance model validation process. We initially viewed the performance validation process as always monotonically improving toward a small asymptotic error rate. The graph on the left in Figure 2 illustrates this naive notion. We assumed that as debugging time increases and as we add details to a performance model, its results would become more and more accurate.

After analysis of our results, we observed that the performance model results changed sporadically with each bug fix or addition of detail. We discovered that bugs can arbitrarily increase or decrease the performance model's predicted cycle counts. Careful analysis of this validation process leads us to believe that performance model results bounce significantly as debugging and validation progresses. Only systematic validation with large numbers of generated patterns can mature a performance model from infant to adult status. As the validation process advances, the test suites become more advanced, further improving the performance model. Only when the error range is consistently less than the desired level—for example, less than 2 percent—for all benchmarks, that is, attaining adulthood, can we place confidence in a performance model. The graph on the right of Figure 2 illustrates the maturing of a performance model from infancy, through the childhood and teenage periods and finally to adulthood.

Microprocessor designers use performance models to assist in design decisions by adding new features and comparing the execution results to a baseline model. Such *A-B testing*, a normal practice in industry to determine the relative merits of two design options, implicitly makes the naive assumption that performance models asymptotically and rapidly approach correctness. If, as we have observed, the actual behavior is sporadic, *designers cannot make quality design decisions without an adult model.* The problem lies in understanding the true performance effect of a new feature. In an unstable performance model, model changes may inadvertently remove existing bugs, introduce new bugs, or reduce the performance impact of an existing bug. This effect can mislead designers into implementing features that do not actually improve performance or not implementing features that would.

This article presents experimental results on calibrating a performance model against actual hardware, and based on these results suggests a systematic method for validating performance models. This method involves calibrating the performance model against the hardware RTL model throughout the design cycle. The procedure and automatic test sequences used for functional validation are applied to performance model validation. This will speed up the maturation of the performance model. It is noted that at the beginning of the design cycle an RTL model is not available. During this early period of the design, inspection techniques are necessary. Once the development of the RTL model begins, the calibration of the performance model can begin. By tracking the performance model with the hardware RTL model, the performance model can provide accurate performance projections throughout the remainder of the design cycle.

Our results highlight the difficulty in developing an accurate performance model. As microarchitecture complexity continues to increase, especially with the

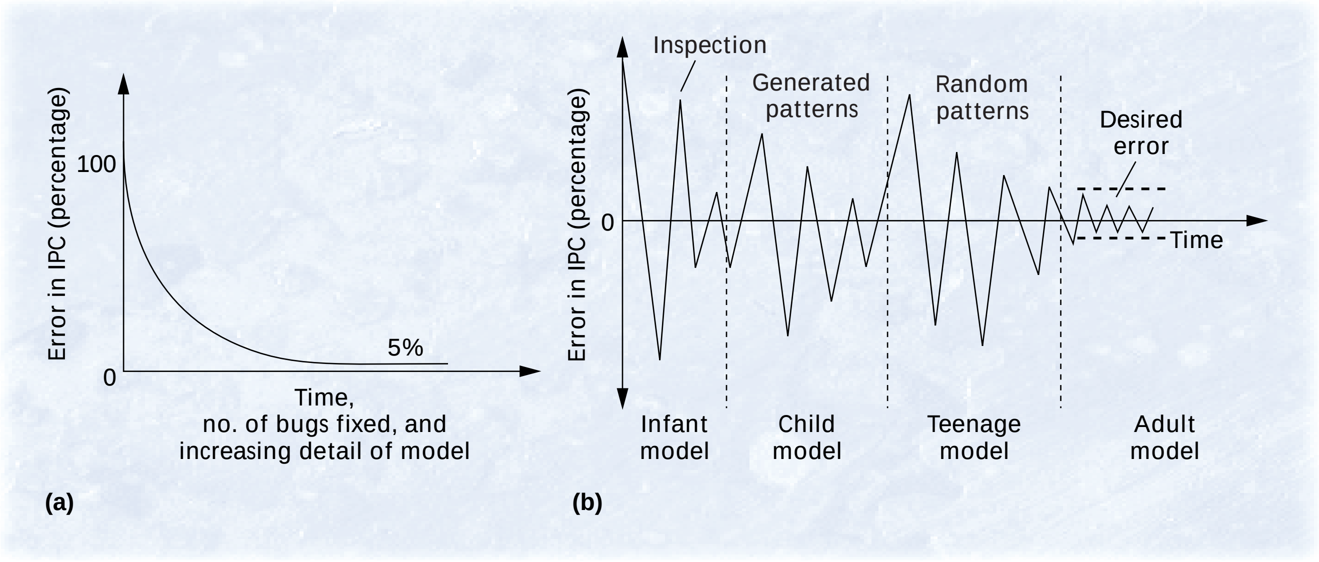


Figure 2. Conflicting views of the maturation of a performance model: (a) simplistic view in which inaccuracies decrease monotonically and (b) realistic view in which results oscillate and then slowly converge.

incorporation of aggressive speculation techniques,[8-10] accurate performance modeling and the validation of performance models will continue to be a great challenge. ❖

Acknowledgments

This research has been supported by the National Science Foundation (CCR 9423272) and the Office of Naval Research (N00014-95-1-1112 and N00014-96-1-0347). We have also benefited from the generous donation of numerous Pentium Pro systems from Intel. We especially thank 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. Finally, the reviewers' comments were extremely helpful.

References

1. A. Poursepanj, "The PowerPC Performance Modeling Methodology," *Comm. ACM*, June 1994, pp. 47-55.
2. A. Huang and T. Diep, *MW Developer's Guide*, Tech. Report CMuART-95-1, Carnegie Mellon Univ., Pittsburgh, Aug. 1995.
3. T. Diep and J.P. Shen, "VMW: A Visualization-Based Microarchitecture Workbench," *Computer*, Dec. 1995, pp. 57-64.
4. B. Black et al., "Can Trace-Driven Simulation Accurately Predict Superscalar Performance?" *Proc. IEEE Int'l Conf. Computer Design*, IEEE Computer Soc. Press, Los Alamitos, Calif., pp. 478-485.
5. K. Diefendorff and E. Silha, "The PowerPC User Instruction Set Architecture," *IEEE Micro*, Oct. 1994, pp. 30-41.
6. S.P. Song, M. Denman, and J. Chang, "The PowerPC 604 RISC Microprocessor," *IEEE Micro*, Oct. 1994, pp. 8-17.
7. *PowerPC 604 RISC Microprocessor User's Manual*, IBM Microelectronics Division, 1994.
8. M. Lipasti, C. Wilkerson, and J.P. Shen, "Value Locality and Data Speculation," *Proc. 7th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, Computer Soc. Press, Los Alamitos, Calif., pp. 138-147.
9. S. McFarling, *Combining Branch Predictors*, Tech. Note TN-36, Western Research Lab, Digital Equipment Corp., June 1993.
10. E. Rotenberg, S. Bennett, and J. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," *Proc. 29th Int'l Symp. Microarchitecture*, Dec. 1996, pp. 24-34.

Bryan Black is a PhD candidate in Carnegie Mellon University's Electrical and Computer Engineering Department. His research interests span computing systems and tropical fruits. He spent four years at Motorola as a design engineer and was a member of the PowerPC 604 design team before returning to CMU for his PhD. Black received a BS and an MS from CMU in electrical and computer engineering.

John Paul Shen is a professor in Carnegie Mellon University's Electrical and Computer Engineering Department and heads the Carnegie Mellon Microarchitecture Research Team (CMuART). Shen received a BS from the University of Michigan and an MS and a PhD from the University of Southern California, all in electrical engineering. He is an IEEE fellow.

Contact Shen at the CMuART, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213; shen@ece.cmu.edu