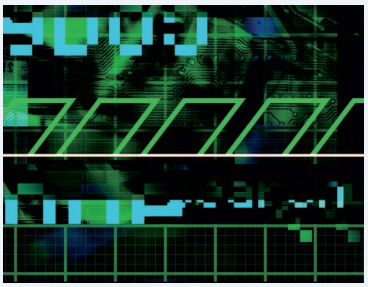
The Future of Simulation: A Field of Dreams?

Improving the infrastructure, benchmarking, and methodology of simulation—the dominant computer performance evaluation method—will result in higher efficiency and let architects gain more insight into processor behavior.



Joshua J. Yi
Freescale Semiconductor

Lieven Eeckhout
Ghent University

David J. Lilja
University of Minnesota, Minneapolis

Brad Calder
University of California, San Diego

Lizy K. John
University of Texas at Austin

James E. Smith
University of Wisconsin-Madison

Due to the enormous complexity of computer systems, researchers use simulators to model system behavior and generate quantitative estimates of expected performance. Researchers also use simulators to model and assess the efficacy of future enhancements and novel systems. Arguably the most important tools available to computer architecture researchers, simulators offer a balance of cost, timeliness, and flexibility.

For these reasons, architecture researchers have increasingly relied on simulators. Table 1 classifies the performance evaluation methods for papers appearing in the International Symposium on Computer Architecture—the flagship conference in computer architecture—in six selected years. In the conference's inaugural year, only two papers out of 28 (7.1 percent) were simulation-based, but that number steadily increased to 27.9 percent in 1985, 71.9 percent in 1993, 80 percent in 1997, and finally to 88 percent and 87 percent in 2001 and 2004, respectively.

These results indicate that simulation is a critical component in computer architecture research. While there is a heavy (and growing) dependence on general simulation, architects use a wide variety of methodologies, benchmarks, and data sets. As a result, several important questions arise regarding the simulation process.

Do popular simulators adequately model processor or system behavior? What are the gaps, if any, in simulation technology? What are the essential features of future simulators? How can we minimize the errors and improve the simulation speed of future simulators? Are benchmark suites truly representative of typical applications? How can we design benchmark suites that contain representative benchmarks, without being overly redundant? How can we properly weight benchmarks to represent realistic workloads? Are simulation methodologies sufficiently defined to permit independent replication of results? How can we add scientific rigor to simulation methodologies?

To encourage and foster consideration of these critical questions, a panel discussion at the International Symposium on Performance Analysis of Systems and Software in March 2004 focused on simulation infrastructure, benchmarks, and simulation methodology.

Published by the IEEE Computer Society

Most of these issues are geared toward researchers. However, some also apply to computer designers and developers who heavily rely on simulations to guide the design process. Moreover, techniques used in academia might eventually percolate into industry practices.

Table 1. Performance evaluation methodologies in papers appearing in the *Proceedings of the International Symposium on Computer Architecture*. The total is not necessarily the sum across the columns because some papers used more than one evaluation method. Adapted from Skadron.[1]

Year	Total papers	Simulation	Measurement	Mathematical modeling	Other
2004	31	27	3	1	0
2001	25	22	2	0	2
1997	30	24	6	0	0
1993	32	23	9	6	1
1985	43	12	1	14	16
1973	28	2	0	5	21

SIMULATION INFRASTRUCTURE

Collectively, in addition to striving for accuracy by modeling the processor in great detail, the developers of the current generation of simulators focus on flexibility. However, computer architects must explore relatively large design and application spaces, and detailed modeling and flexibility together pose problems that limit this exploration. The computer architecture community must devise solutions to ensure that these conflicting issues do not inhibit future computer architecture research and design.

Availability and diversity

Implementing and validating a simulator that faithfully models a state-of-the-art processor's behavior is extremely time-consuming and arduous. As a result, few researchers invest the effort to construct simulators suitable for open source distribution; SimpleScalar (www.simplescalar.com)—the simulator most researchers currently use—is a notable exception. The slow and infrequent development of open source simulators often results in using simulators that model unrealistic, dated, or even obsolete computer architectures.

There is currently a limited simulation infrastructure for multiprocessor and multithreaded simulation. However, with the recent advent of chip multiprocessors, interest in multiprocessor simulation infrastructure has grown significantly.

While researchers are currently working on open source simulators, the computer architecture community must expend additional effort to provide a wider variety of open source simulators for uniprocessor, multiprocessor, and multithreaded systems. This will lower the barrier of entry for other researchers to make contributions, while stimulating the independent validation of proposed ideas, which will likely improve the overall quality of research for these topics. The use of modular simulation infrastructures such as Liberty,[2] which allows for the quick building of simulators by assembling several reusable components, might be an interesting avenue to explore.

Speed and levels of abstraction

A primary characteristic of many modern simulators is their *cycle accuracy*, or the high level of detail to which they model the processor. Although this level of detail is critical for the simulator's fidelity and accuracy, the associated tradeoff is decreased simulation speed. The slower simulation speed is especially limiting when exploring an enormous design space that is the product of large numbers of processor configurations, compiler optimizations, benchmarks, and data sets. While this is a severe problem for computer designers during design space exploration, researchers also face the limitation of slow simulation speeds. Given the preponderance of papers that rely entirely on cycle-accurate simulators, it appears that architects are not using the full range of simulation options—such as analytical modeling and statistical simulation—to efficiently traverse and characterize the design space. Consequently, these detailed simulation-based studies often incompletely characterize the design space, which subsequently detracts from the study's overall contribution.

To address the problem of efficiently traversing the design space, we recommend using and improving on a range of alternatives, especially during the early phases of the design cycle. These alternatives include

- analytical models,[3,4]
- statistical simulation,[5,6] and
- specialized trace-driven simulation.

Analytical modeling involves developing a limited number of formulas that summarize performance based on program characteristics and microarchitectural parameters. Statistical simulation combines analytical modeling and simulation to generate a synthetic trace based on several program characteristics and is subsequently simulated on a simple trace-driven simulator. The "Statistical Simulation" sidebar provides a detailed explanation of this process.

Trace-driven simulation is an old technique in which functional simulation is separated from detailed simulation. Trace-driven simulation yields a slight speed advantage over execution-driven simulation since trace-driven simulation does not functionally execute the pro-

Statistical Simulation

Lieven Eeckhout, Ghent University
Lizy K. John, University of Texas at Austin
James E. Smith, University of Wisconsin-Madison

Statistical simulation[1-4] reduces the runtime for a detailed microarchitecture simulation. The basic idea of statistical simulation is to

- measure key program characteristics, that is, collect a statistical profile of the program execution;
- generate a synthetic trace having the same characteristics; and
- simulate the synthetic trace.

The synthetic trace is statistically similar to the average overall program behavior, making it several orders of magnitude shorter than the original program execution while providing reasonable accuracy. Further, the simulation model is much simpler than the detailed model because instructions collapse into a small number of types; cache and branch predictor behavior is also modeled statistically.

Given that a statistical profile reflects the key properties of the program's execution behavior, statistical simulation can accurately estimate performance and power. In recent years, statistical simulation research has evolved to improve its accuracy by including additional levels of correlation among program characteristics. The most accurate statistical simulation frameworks known to date include statistical flow graphs to model paths of execution,[4] as well as accurate memory data flow models for delayed hits and cache miss correlation.[5] The goal of statistical simulation is not to replace detailed simulation but to serve as a useful complement to speed up the early stages of the design process. A faster simulation methodology allows more efficient exploration of a large design space.

Statistical profiling

Specialized functional or trace-driven simulation provides both microarchitecture-independent and microarchitecture-dependent characteristics for the statistical profile.

Microarchitecture-independent characteristics include the statistical flow graph, the instruction types, and the inter-instruction dependencies (through registers as well as memory). Microarchitecture-dependent characteristics include locality metrics such as branch predictability and cache behavior. Since locality metrics are difficult to model in a microarchitecture-independent way, the pragmatic approach is to simultaneously compute them for a wide range of branch predictors and cache configurations in a single profiling run.

Synthetic trace generation

After generating the statistical profile, the trace generator produces the synthetic trace. Synthetic trace generation uses random numbers between zero and one to select a program characteristic from its cumulative distribution function. A synthetically generated instruction consists of its type (arithmetic logic unit, load, store, or branch), its dependencies, and whether it causes an I-cache miss. In the case of load or branch instructions, the instruction indicates whether it causes a d-cache miss or is a mispredicted

gram, while execution-driven simulation combines both functional and detailed simulation in a single run.

Allowing the simulation of any program given the correct tracing infrastructure is trace-driven simulation's main advantage. Pure execution-driven simulation requires the emulation of system calls in the simulator, which is tedious to implement and costly to maintain; and simulators often support only a small set of system calls, which limits the type of programs that can be simulated.

In comparison, tracing the minimal amount of information needed to replay a program's execution allows the simulation of any application without having to emulate the operating system effects;[7] and the amount of storage required for these traces is reasonable. Allowing the generation of trace samples to be distributed and letting others reproduce the same program execution for simulation are another two advantages of trace-driven simulation.

On the other hand, trace-driven simulation's main drawback is that it might not capture the mispredicted path of execution, if those instructions or data values are not present in the trace. However, excluding the execution of mispredicted instructions does not typically have a significant effect on simulation accuracy. Although trace-driven simulation currently seems to be less popular than execution-driven simulation, we recommended reevaluating the tradeoffs between the two.

Compared with cycle-accurate simulation, analytical modeling and statistical simulation are typically much faster, but less accurate. Although these two alternatives might be less accurate than cycle-accurate simulation, their higher simulation speed earlier in the design process's evaluation cycle is more important for at least two reasons.

First, although the absolute accuracy of these alternatives might not be as high as the absolute accuracy of cycle-accurate simulation, their relative accuracy is usually sufficient to track significant changes in the processor's performance. In other words, since these alternatives can detect significant trends in the performance, their level of accuracy is sufficient for the design space exploration that occurs early in the design cycle.

branch, respectively. The synthetic trace typically contains 100,000 to 1,000,000 instructions.

Statistical processor modeling

The final step is simulating the synthetically generated trace using a simplified processor model. This simulator models instruction types and dependencies in the same way that detailed simulators do; but modeling cache misses and branch mispredictions is quite different.

The simulator models miss events statistically according to their miss rates. In case of an I-cache miss, the simulator stops fetching instructions for a number of cycles equal to the miss delay. In case of a D-cache load miss, the simulator assigns the access latency of the next level in the memory hierarchy; the remaining miss latency is assigned in a delayed hit. In case of branch misprediction, the simulator flushes the pipeline when the mispredicted branch gets executed.

Because the synthetic trace simulator does not need to explicitly model caches or branch predictors, the simulation model is both simpler to write and runs faster than a conventional detailed simulator. Coupled with the very short traces, the simulation times for this simulator are several orders of magnitude lower than their detailed counterparts.

Applications

Statistical simulation has several applications. The most obvious is uniprocessor power and performance design. Experiments[3,4,6,7] show that statistical simulation achieves excellent relative accuracy, making it extremely useful for early design stage exploration. Workload characterization and program analysis is a second application because the key features modeled in a statistical simulation framework are the ones more relevant to determining overall performance. Third, statistical simulation will be even more efficient for large system-level explorations and design studies than detailed cycle-by-cycle simulations.

References

1. M. Oskin, F. Chong, and M. Farrens, "HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Design," *Proc. 27th Ann. Int'l Symp. Computer Architecture*, ACM Press, 2000, pp. 71-82.
2. S. Nussbaum and J. Smith, "Modeling Superscalar Processors via Statistical Simulation," *Proc. 10th Ann. Int'l Conf. Parallel Architectures and Compilation Techniques*, IEEE CS Press, 2001, pp. 15-24.
3. L. Eeckhout et al., "Statistical Simulation: Adding Efficiency to the Computer Designer's Toolbox," *IEEE Micro*, Sept./Oct. 2003, pp. 26-38.
4. L. Eeckhout et al., "Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies," *Proc. 31st Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, 2004, pp. 350-361.
5. D. Genbrugge, L. Eeckhout, and K. De Bosschere, "Accurate Memory Data Flow Modeling in Statistical Simulation," *Proc. 20th Ann. Int'l Conf. Supercomputing*, IEEE CS Press, 2006.
6. S. Eyerman, L. Eeckhout, and K. De Bosschere, "Efficient Design Space Exploration of High Performance Embedded Out-of-Order Processors," *Proc. Conf. Design Automation and Test in Europe*, European Design and Automation Association, 2006, pp. 351-356.
7. A. Joshi et al., "Evaluating the Efficacy of Statistical Simulation for Design Space Exploration," *Proc. IEEE Symp. Performance Analysis of Systems and Software*, IEEE Press, 2006, pp. 70-79.

Another key advantage of analytical modeling and statistical simulators is that their implementation requires a fraction of the time of the typical cycle-accurate simulator, which increases their utility early in the design cycle. When the goal of simulation is to gain insight into the processor's behavior or to explore the design space, adding fine-grained detail into the simulator's processor model might not be worth the additional modeling time. In addition to dramatically decreasing the simulation speed, using a simulator that is too detailed might cause the researcher or designer to get bogged down in the details and miss the big picture.

In fact, one of the overriding conclusions of the 2004 panel discussion was that simulations tend to cause the computer architect to focus more on the quantitative measurement itself than on the insight that the measurement yields.

Reduced simulation time techniques

Because the simulator's speed can limit the amount of design space exploration, researchers and designers use reduced simulation time techniques such as truncated execution, reduced input sets, and sampling.

In the sampling category, researchers have recently introduced sophisticated statistically-based simulation techniques such as SimPoint[8] and *sampling microarchitecture simulation* (SMARTS),[9] which can drastically reduce the simulation time with little loss in simulation accuracy. The "Statistical and Representative Sampling" sidebar (on the next page) provides further information on sampling-based simulation techniques. To further reduce the simulation time of a single test case, in conjunction with the aforementioned techniques, researchers can distribute the simulation of samples across a cluster of machines for simulation in parallel.[10]

Researchers must address two issues to produce accurate and fast results from sampling techniques. First, they must load the architectural state efficiently, which they can achieve through reduced checkpointing.[11] Second, they should construct the hardware state, such as cache and branch predictor, as accurately as possible. They can achieve this by using warm-up techniques such

Statistical and Representative Sampling

Brad Calder
University of California, San Diego

Sampling is an established method for representing a data set using a fraction of the data. A sample is a contiguous interval of dynamic instructions during program execution.

Sampling techniques are split into two general types—statistical sampling and representative sampling. Statistical sampling samples the execution in a random or systematic pattern without special consideration of the sample selection. Representative sampling involves carefully choosing samples to uniquely represent repetitive patterns in a program's execution.

Statistical sampling

Computer architecture researchers have proposed several different sampling techniques to estimate a program's behavior. Statistical sampling has a rigorous mathematical foundation based on the central limit theorem.

Subhasis Laha and colleagues[1] introduced the use of random sampling to evaluate cache memory performance. They compared the sampled mean's accuracy and examined the distribution of random sampling to show that it matched that of the real trace.

Thomas Conte[2] pioneered the use of statistical sampling in processor simulation. He applied sampling methods to cache and instruction traces with good accuracy. In more recent work, Conte and colleagues[3] provided a framework that took random samples from the execution. They computed the samples' statistical metrics such as standard deviation, probabilistic error, and confidence bounds to predict the estimated results' accuracy, and statistically analyzed the metric of interest such as *instructions per cycle*.

Conte and colleagues specified two sources of error in their sampling technique—nonsampling bias and sampling bias. Nonsampling bias results from improperly warming up the processor. Sampling bias, on the other hand, is fundamental to the samples, since it quantifies how accurately the sample average represents the overall average. Two major parameters influence this error—the number of samples and the size of each sample in instructions.

The sample size in processor simulation is the number of dynamic instructions that a sample encompasses. The smaller the sample size, the faster the potential simulation time, but this comes at the cost of increased overhead and complexity because of the need for accurate sample warm-up.[4]

To determine the amount of samples to take, the user determines a particular accuracy level for estimating the metric of interest. The benchmark is then simulated and N samples are collected, N being an initial value for the number of samples. Error and confidence bounds are computed for the samples, and, if they satisfy the accuracy limit, this estimate is good. Otherwise, more samples ($> N$) must be collected, and the error and confidence bounds must be recomputed for each collected sample set until the accuracy threshold is satisfied. The SMARTS[5] framework proposes an automated approach for applying this sampling technique.

as *memory reference reuse latency*[12] or by constructing a particular hardware state from a generic stored hardware state. As an example, constructing cache state from the state of a larger cache is both easy to do and accurate.[11] These recent contributions make producing both accurate and fast results from these sampling techniques possible.[13]

Uniprocessor simulation has been the target for most of this sampled simulation research, yet little research has been directed toward multiprocessor and multithreaded system simulation. Researchers must investigate sampling techniques because extending the uniprocessor's efficient architecture and hardware state construction techniques to multiprocessor and multithreaded system simulation is especially challenging.

BENCHMARKING

Ideally, a benchmark suite collectively represents the commonly used programs in a particular application space, such as general-purpose, online-transaction-processing, scientific, multimedia, and embedded applications. However, as is the case for simulators, potential problems might either limit the utility of those benchmarks or, worse yet, result in misleading conclusions.

Representativeness and subsetting suites

The more serious of the two benchmarking problems is the uncertainty in the representativeness of typical benchmark suites. More specifically, while certain benchmark suites are not necessarily more representative than others, the representativeness of most benchmark suites is unknown. For example, the question is not whether the benchmarks in the Standard Performance Evaluation Corporation (SPEC) CPU 2006 benchmark suite—the current de facto standard for general-purpose computing—are evenly distributed over the entire application space or have similar characteristics. Rather, it is that the representativeness of the SPEC CPU 2006 benchmark suite is unknown.

A related problem is the subsetting of benchmark suites for simulation. The key questions are not whether researchers are subsetting benchmark suites or whether they should be subsetted, but rather are they subsetting benchmarks in a justifiable way, is the resulting subset

Representative sampling

Representative sampling contrasts with statistical sampling in that it first analyzes the program's execution to pick a representative sample for each unique behavior in the program's execution. The key advantage of this approach is that having fewer samples can reduce simulation time and also allows for a simpler simulation infrastructure.

SimPoint[6] is an infrastructure that chooses a small number of representative samples that, when simulated, represent the program's complete execution. SimPoint relies on the fact that a program's execution is a series of repeating patterns (loops and procedure calls), which are recurring behaviors called phases. This structured behavior greatly benefits simulation, since only a single sample needs to be simulated from each repetitive pattern or phase.

To accomplish this, SimPoint breaks a program's execution into fixed-length intervals of execution, for example, 10 million instructions. An instruction interval is defined to represent a contiguous stream of instructions during program execution. A vector, in which every dimension represents a static code construct, such as branch edges, basic blocks, loops, or procedures, represents each interval. The dimension is incremented when each code construct is executed during that interval's execution. Therefore, the vector represents the proportion of code executed during that interval and forms a code signature for it.

SimPoint then compares two vectors by computing their distance from each other. Vectors close to each other are grouped into the same phase, or cluster, using the *k*-means algorithm from machine learning. Only one interval is chosen from a cluster for detailed simulation because intervals with similar code signatures have similar architectural metrics. Simulating each of the representative samples together—one from each cluster—creates a complete and accurate representation of the program's execution in minutes.[4]

References

1. S. Laha, J. Patel, and R. Iyer, "Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems," *IEEE Trans. Computers*, Nov. 1988, pp. 1325-1336.
2. T. Conte, "Systematic Computer Architecture Prototyping," doctoral dissertation, Dept. Electrical and Computer Eng., Univ. of Illinois at Urbana-Champaign, 1992.
3. T. Conte, M. Hirsch, and W. Hwu, "Combining Trace Sampling with Single Pass Methods for Efficient Cache Simulation," *IEEE Trans. Computers*, June 1998, pp. 714-720.
4. M. Van Biesbrouck, L. Eeckhout, and B. Calder, "Efficient Sampling Startup for Sampled Processor Simulation," *Proc. Int'l Conf. High Performance Embedded Architectures and Compilers*, Springer, 2005, pp. 47-67.
5. R. Wunderlich et al., "SMARTS: Accelerating Microarchitectural Simulation via Rigorous Statistical Sampling," *Proc. 30th Ann. Int'l Symp. Computer Architecture*, ACM Press, 2003, pp. 84-95.
6. T. Sherwood et al., "Automatically Characterizing Large Scale Program Behavior," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, 2002, pp. 45-57.

of benchmarks representative of the entire suite, and should some benchmarks in the subset (or in the suite for that matter) be weighted more than others?

Also, of the classification techniques used to cluster benchmarks before subsetting, do some techniques yield more meaningful classifications than others? If the subset of benchmarks that the computer architect selects is not representative of the entire suite, when it should be, such as for general-purpose computing, the conclusions the architect draws might be completely different from the conclusions that would have been drawn had the entire suite been used.

Ideally, a benchmark suite should cover the desired application space with as few benchmarks as possible. Implicit in this stipulation is that redundancy—in terms of multiple benchmarks covering the same subspace—should be minimized. To help accomplish this, the computer architecture community must propose additional methods of characterizing and classifying benchmarks. Currently that includes, but is not limited to, characterizing benchmarks based on their performance bottlenecks[14] or their principal components, that is, their architectural characteristics such as cache miss rate, branch prediction accuracy, amount of instruction-level parallelism, and so forth.[15] Other options include classifying benchmarks based on their characteristics such as computation or memory-boundedness and integer and floating point.

To address these two problems, computer architecture researchers must continue to develop characterization and classification methods that are more accurate or efficient than current methods, providing important building blocks for developing benchmark suites that efficiently span an application space.

Simulation time

The second problem we identified was the benchmarks' length, which translates into simulation time. For some benchmarks, such as those from SPEC CPU 2006, the intended purpose was to use the entire suite to measure the performance of real processors. For many reasons, including portability and standardization, architecture researchers have adopted these benchmarks for simulation.

However, due to the tremendous speed difference between native execution and simulation, simulating a complete benchmark takes several orders of magnitude longer than hardware execution, which makes simulating the entire benchmark suite impossible for most studies. On the other hand, some benchmarks might be too abbreviated if they exclude the initialization and clean-up sections of the code, and it is important to have representative samples from all phases of a program's execution.

To reduce the simulation time of industry-standard benchmarks to a tractable level, we again recommend using sampling-based techniques such as SimPoint and SMARTS.

SIMULATION METHODOLOGY

Broadly, simulation methodology is defined as the specific steps of setting up and running simulations, and then analyzing the results. Using poor simulation methodology might affect simulation results or limit the insight gained from simulations.

Ad hoc simulation

The first problem with current simulation methodology is often ad hoc simulation. For example, when evaluating the efficacy of a processor enhancement, the architecture researcher must choose processor and memory parameters to configure the simulator, select a subset of benchmarks, use a reduced-time simulation technique to minimize the overall simulation time, and analyze the reasons for any observed improvements. At best, however, researchers typically neglect to describe the justification for these decisions or state their underlying assumptions; at worst, they choose parameters and a subset of benchmarks haphazardly or do not characterize their results in a systematic or statistically rigorous manner.

In addition to having the potential of significantly skewing the simulation results—for example, by choosing a poor set of parameters—ad hoc simulation methodology might also obscure important conclusions. In a similar vein, inadequately documenting or justifying the simulation methodology makes independent replication of simulation results—and improvements to the original work—difficult, if not impossible.

To address the problem of ad hoc simulation methodology, computer architecture researchers must add more scientific rigor to their studies. In particular, the two specific recommendations are to comprehensively document and justify the simulation methodology and to make it more statistically rigorous. Carefully documenting the simulation methodology ensures reproducibility of results, and providing justification for the choices offers additional insight into the researcher's thought processes.

Further, by incorporating statistical rigor into simulation methodology, the researcher can run simulations efficiently, reduce the errors in the simulation methodology, and gain insight into the behavior of the processor enhancement being evaluated. Researchers are developing more statistically rigorous approaches to choosing processor and memory parameters, efficiently reducing the design space, characterizing and classifying benchmarks, and analyzing the effects of enhancements.[14,15]

Reproducibility and comparability are key problems with current simulation methodology.

Simulation workloads

Researchers often use a wide range of processor and memory subsystem configurations for their simulations. Consequently, the large number of benchmarks, data sets, simulation techniques such as truncated execution and reduced input sets, and benchmark suite subsetting techniques lead to wide variation in the workloads that architecture researchers use for their simulations. Regardless of whether the processor and memory configurations are "realistic" or if the researcher selects the most "appropriate" workload, directly comparing the results from two different simulation studies is difficult when using a wide range of configurations and workloads. Therefore, reproducibility and comparability are key problems with current simulation methodology.

To address these issues, the research community must try to achieve consensus on well-balanced processor and memory hierarchy configurations, common subsets of benchmarks, and common data sets. In addition, making available checkpoints and traces of benchmark execution ensures that the same application-level behavior occurs during simulation for reproducible results.[7] However, it is important to include future applications in the subset of benchmarks to ensure that future application characteristics are well represented.

Currently, simulation is the approach of choice for quantitatively evaluating computer architecture performance. However, problems with the current simulation infrastructure, benchmarking, and simulation methodology can affect simulation accuracy or the time required for simulation, which might potentially affect the conclusions drawn.

We make five key recommendations. First, the state of simulation technology must improve further to allow for efficient multiprocessor and multithreaded simulation and the simulation of popular operating systems and applications. Second, researchers must propose more efficient simulation methods and continue to extend sampling-based simulation techniques for multiprocessor and multithreaded simulations. Third, architects should use

higher-speed alternatives to cycle-accurate simulation to efficiently traverse large design spaces. Fourth, benchmark suites should contain representative and nonredundant benchmarks that are efficient and fast to simulate. Fifth, simulation methodology must be more robust and statistically based to increase the likelihood of independent validation and to facilitate comparability.

The underlying motivation for these recommendations is to gain insight into the behavior of processors in the environments where they will actually be used. Researchers must encourage the use of analytical modeling and statistical theory in performance evaluation, and training for these tools and evaluation techniques should be included in computer architecture education. To extend the historical rate of performance improvement in architectural enhancements, the computer architecture community must implement these recommendations and address other simulation-related questions. ■

References

1. K. Skadron et al., "Challenges in Computer Architecture Evaluation," *Computer*, Aug. 2003, pp. 30-36.
2. M. Vachharajani et al., "Microarchitectural Exploration with Liberty," *Proc. 35th Ann. Int'l Symp. Microarchitecture*, IEEE CS Press, 2002, pp. 271-282.
3. T. Karkhanis and J. Smith, "Modeling Superscalar Processors," *Proc. 31st Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, 2004, pp. 338-349.
4. T. Sherwood, M. Oskin, and B. Calder, "Balancing Design Options with Sherpa," *Proc. Int'l Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, ACM Press, 2004, pp. 57-68.
5. S. Nussbaum and J. Smith, "Modeling Superscalar Processors via Statistical Simulation," *Proc. 11th Ann. Int'l Conf. Parallel Architectures and Compilation Techniques*, IEEE CS Press, 2001, pp. 15-24.
6. L. Eeckhout et al., "Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies," *Proc. 31st Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, 2004, pp. 351-362.
7. S. Narayanasamy et al., "Automatic Logging of Operating System Effects to Guide Application-Level Architecture Simulation," *Proc. Joint Int'l Conf. Measurement and Modeling of Computer Systems*, ACM Press, 2006, pp. 216-227.
8. T. Sherwood et al., "Automatically Characterizing Large Scale Program Behavior," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, 2002, pp. 45-57.
9. R. Wunderlich et al., "SMARTS: Accelerating Microarchitectural Simulation via Rigorous Statistical Sampling," *Proc. 30th Ann. Int'l Symp. Computer Architecture*, ACM Press, 2003, pp. 84-95.
10. S. Girbal et al., "DiST: A Simple, Reliable and Scalable Method to Significantly Reduce Processor Architecture Simulation Time," *Proc. 2003 ACM Sigmetrics Int'l Conf. Measurement and Modeling of Computer Systems*, ACM Press, 2003, pp. 1-12.
11. M. Van Biesbrouck, L. Eeckhout, and B. Calder, "Efficient Sampling Startup for Sampled Processor Simulation," *Proc. 2005 Int'l Conf. High Performance Embedded Architectures and Compilers*, Springer, 2005, pp. 47-67.
12. J. Haskins Jr. and K. Skadron, "Memory Reference Reuse Latency: Accelerated Warmup for Sampled Microarchitecture Simulation," *Proc. 2003 Int'l Symp. Performance Analysis of Systems and Software*, IEEE Press, 2003, pp. 195-203.
13. J. Yi et al., "Characterizing and Comparing Prevailing Simulation Methodologies," *Proc. 11th Ann. Int'l Symp. High-Performance Computer Architecture*, IEEE CS Press, 2005, pp. 266-277.
14. J. Yi, D. Lilja, and D. Hawkins, "A Statistically Rigorous Approach for Improving Simulation Methodology," *Proc. 9th Ann. Int'l Symp. High-Performance Computer Architecture*, IEEE CS Press, 2003, pp. 281-291.
15. L. Eeckhout, H. Vandierendonck, and K. De Bosschere, "Workload Design: Selecting Representative Program-Input Pairs," *Proc. 11th Ann. Int'l Conf. Parallel Architectures and Compilation Techniques*, IEEE CS Press, 2002, pp. 83-94.

Joshua J. Yi *is a performance analyst at Freescale Semiconductor. Yi received a PhD in electrical engineering from the University of Minnesota, Minneapolis. Contact him at jjyi@ece.umn.edu.*

Lieven Eeckhout *is an assistant professor in the Electronics and Information Systems Department at Ghent University, Belgium, and is supported by a postdoctoral fellowship of the Fund for Scientific Research-Flanders. Eeckhout received a PhD in computer science and engineering from Ghent University. Contact him at lieven.eeckhout@elis.ugent.be.*

David J. Lilja *is a professor of the Department of Electrical and Computer Engineering at the University of Minnesota, Minneapolis. Lilja received a PhD in electrical engineering from the University of Illinois at Urbana-Champaign. Contact him at lilja@ece.umn.edu.*

Brad Calder *is a professor in the Department of Computer Science and Engineering at the University of California, San Diego. Calder received a PhD in computer science from the University of Colorado, Boulder. Contact him at calder@cs.ucsd.edu.*

Lizy K. John *is an associate professor in the Department of Electrical and Computer Engineering at the University of Texas at Austin. John received a PhD in computer engineering from the Pennsylvania State University. Contact her at ljohn@ece.utexas.edu.*

James E. Smith *is a professor in the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison. Smith received a PhD in computer science from the University of Illinois.*