Statistical Simulation: Adding Efficiency to the Computer Designer's Toolbox

STATISTICAL SIMULATION ENABLES QUICK AND ACCURATE DESIGN DECISIONS IN THE EARLY STAGES OF COMPUTER DESIGN, AT THE PROCESSOR AND SYSTEM LEVELS. IT COMPLEMENTS DETAILED BUT SLOWER ARCHITECTURAL SIMULATIONS, REDUCING TOTAL DESIGN TIME AND COST.

Lieven Eeckhout
Ghent University

Sebastien Nussbaum
Sun Microsystems

James E. Smith
University of Wisconsin-Madison

Koen De Bosschere
Ghent University

Computer system design is a time-consuming complex process, and simulation is essential to overall design activity. Simulation occurs at many levels, from circuit to system, and at different degrees of detail as the design evolves. The designer's toolbox holds evaluation tools, often used in combination: each tool has different complexity, accuracy, and execution-time properties.

Detailed models of register transfer activity typically conduct simulation at the microarchitecture level. These simulators track instructions and data on a per-cycle basis and typically provide detailed models for features such as instruction issue mechanisms, caches, load/store queues, and branch predictors, as well as their interactions. For input, microarchitecture simulators take sets of benchmark programs, including standard and company-proprietary suites. These benchmarks can each contain billions of dynamically executed instructions, and typical simulators run many orders of magnitude slower than real processors, producing a relatively long runtime for even a single simulation.

However, processor simulation at such a detailed level is neither always appropriate, nor necessary. For example, early in the design process, while exploring the design space and determining the high-level microarchitecture, too much detail only gets in the way. The initial definition of a processor microarchitecture requires basic design decisions. These decisions involve tradeoffs related to basic cycle time and instructions per cycle; cache and predictor sizing; and performance/power. At this stage of the design process, detailed microarchitecture simulations of specific benchmarks aren't feasible. For one, the detailed simulator itself takes considerable time and effort to develop. Second, benchmarks restrict the application space under evaluation to the specific programs represented by the benchmarks. To study a fairly broad design space, the number of simulation runs can be quite large. Finally, highly accurate performance estimates are illusory anyway, given the knowable level of design detail.

Similarly, for making system-level design decisions, where a processor (or several processors) might be combined with many other components, a very detailed simulation model is often unjustified or impractical. Even though the detailed processor microarchitecture might be known, the number of processors and the larger benchmark programs necessary for

Published by the IEEE Computer Society

0272-1732/03/$17.00 © 2003 IEEE

Table 1. Comparing existing simulation and modeling techniques.

Technique	Development time	Evaluation time	Accuracy	Coverage
Functional simulation	Excellent	Good	Poor	Poor
Specialized cache and predictor simulation	Good	Good	Poor	Poor
Full trace-driven simulation	Poor	Poor	Excellent	Excellent
Full execution-driven simulation	Poor	Poor	Excellent	Excellent
Modular full execution-driven simulation	Good	Poor	Excellent	Excellent
Sampled simulation	Poor	Fair	Good to excellent	Excellent
Analytical modeling	Excellent	Excellent	Fair	Poor
Statistical simulation	Good	Excellent	Good	Good

studying system-level behavior multiply many fold the simulation complexity.

Statistical simulation[1-4] can overcome many shortcomings of detailed simulation for those situations where detailed modeling is impractical or at least overly time consuming. Statistical simulation permits early exploration of the design space with relatively little new-tool development effort. In addition, it provides a simple way of modeling superscalar processors as components in large-scale systems where a high level of detail is unnecessary or practical.

Table 1 presents the practice of microprocessor design and the role statistical simulation can play in an overall simulation framework by comparing existing techniques in terms of development and evaluation times, accuracy in predicting overall performance, and level of microarchitecture detail or coverage.

Functional simulation models only the functional characteristics of an instruction set. It simulates instructions one by one, taking input operands and generating output values. These tools are typically most useful for determining if a design implements an instruction set correctly rather than its performance characteristics. Consequently, accuracy and coverage are poor (with respect to performance and implementation detail). However, development time is excellent because a functional simulator is usually already present at hardware development time (unless the processor implements a new instruction set). Functional simulators have a very long lifetime that can span many development projects. Evaluation time is good because no microarchitecture features need modeling. From a hardware development perspective, functional simulation is most useful because it can generate instruction and address traces—the functionally correct sequence of instructions and/or addresses that a benchmark program produces. Other simulation tools, including statistical simulation, can use these traces.

Specialized cache and predictor simulation takes program instruction and address traces, and simulates only cache or branch predictor behavior in isolation. Such simulations usually evaluate performance as miss rate or miss prediction rate. These tools are widely available, especially for cache simulation, and they can evaluate a variety of cache configurations simultaneously in a single simulation run.[5,6] Although development time and evaluation time are both good, coverage is limited because cache and predictor simulations model only certain specific aspects of a processor. And although the accuracy in terms of miss rate is quite good, overall processor performance accuracy correlates only roughly with these miss rates because so many other factors come into play. Hence, overall accuracy is poor (or perhaps not applicable).

Full trace-driven simulation takes program instruction traces and feeds the full benchmark trace into a detailed microarchitecture simulator. A trace-driven simulator separates the functional simulation from the timing simulation. This separation is often useful because designers must only perform functional simulation once, whereas they perform detailed timing simulations many times. This separation reduces the detailed simulation runtime.

One disadvantage of this approach is the need to store trace files, which can be quite large. Another disadvantage for modern superscalar processors is that they predict

branches and execute many instructions speculatively. These discarded instructions do not show up on a trace file from functional simulation, although they can affect cache and/or predictor contents.[7] Overall, full trace-driven simulation requires a long development time and long simulation runtimes, but both accuracy and coverage are excellent.

Full execution-driven simulation is similar to full trace-driven simulation, except it combines functional simulation with detailed timing simulation. Consequently, this type of simulation does not have to store trace files and can accurately simulate speculated instructions. Recently, execution-driven simulation has supplanted trace-driven simulation as the method of choice. An example in academia is SimpleScalar's out-of-order simulator,[5] computer companies use similar simulation tools.[8] Just as with trace-driven simulation, full execution-driven simulation is highly accurate, attains excellent coverage—the simulator models the microarchitecture in detail—but it is very time consuming to run and requires a long time to develop.

To overcome the difficulty of building detailed cycle-accurate simulators, recent work has presented simulation frameworks built along the philosophy of modular components. Examples of such modular, full execution-driven simulation tools are Asim[8] and Liberty.[9] Within such an infrastructure, designers can easily reuse, extend, and modify architectural components to quickly build complex performance models. Consequently, the development time is good once the general infrastructure is available.

Unfortunately, modular, full execution-driven simulation does not address the problem of long simulation times. A partial solution to this problem is to carefully select a portion (or portions) of a full benchmark program and then simulate only that portion. Computer designers and researchers often implement this sampled-simulation approach by either selecting one single large sample or multiple small samples. Selecting a representative sampled trace is an important issue with possible pitfalls. For example, samples from the beginning of a program are likely to contain nonrepresentative initialization code. The more general problem is that the execution of a computer program consists of several program phases. As such, an effective sampled trace should represent each major program phase.[2,10-12]

Another approach, at the opposite end of the spectrum, is to use *analytical modeling*. With analytical modeling, several equations approximate the microarchitecture's behavior. Conceptually, analytical model development should be simpler than developing a detailed simulation model and should provide valuable insight. However, researchers have only derived reasonably accurate analytical models for simple (for example, in-order) microarchitectures. They have yet to develop accurate analytical modeling techniques that cover superscalar architectures and many parameters. The coverage of this approach is poor.

Between detailed simulation and analytical models lies *statistical simulation*.[1-4,13] Statistical simulation is very simple. First, the designer computes (or otherwise derives) a statistical program profile, using simple trace-based tools, such as specialized cache and predictor simulators. This statistical profile is a collection that records the distributions of important program characteristics. The statistical profile subsequently generates a synthetic instruction trace that executes on a simple trace-driven simulator. Because the synthetic trace is randomly generated from the statistical profiles, the simulation process quickly converges to a set of performance projections. As such, very short synthetic traces (several hundred thousand instructions or fewer) can yield accurate performance estimates. The computation time for this approach is several orders of magnitude less than for full benchmark simulation.

Furthermore, the trace-driven statistical simulator itself is simpler because it does not have to model all of a microprocessor's details. This sort of simulation statistically model several microarchitectural mechanisms that do not need precise hardware modeling. So the lines of code for a synthetic trace simulator—a few hundred in the ones we have developed—are significantly fewer than for a fully detailed simulator, which takes at least several tens of thousands of lines. Finally, program statistics that drive the synthetic trace generation do not necessarily have to come from a specific benchmark program. Rather, the designer can choose them to cover the full range of expected program behavior.

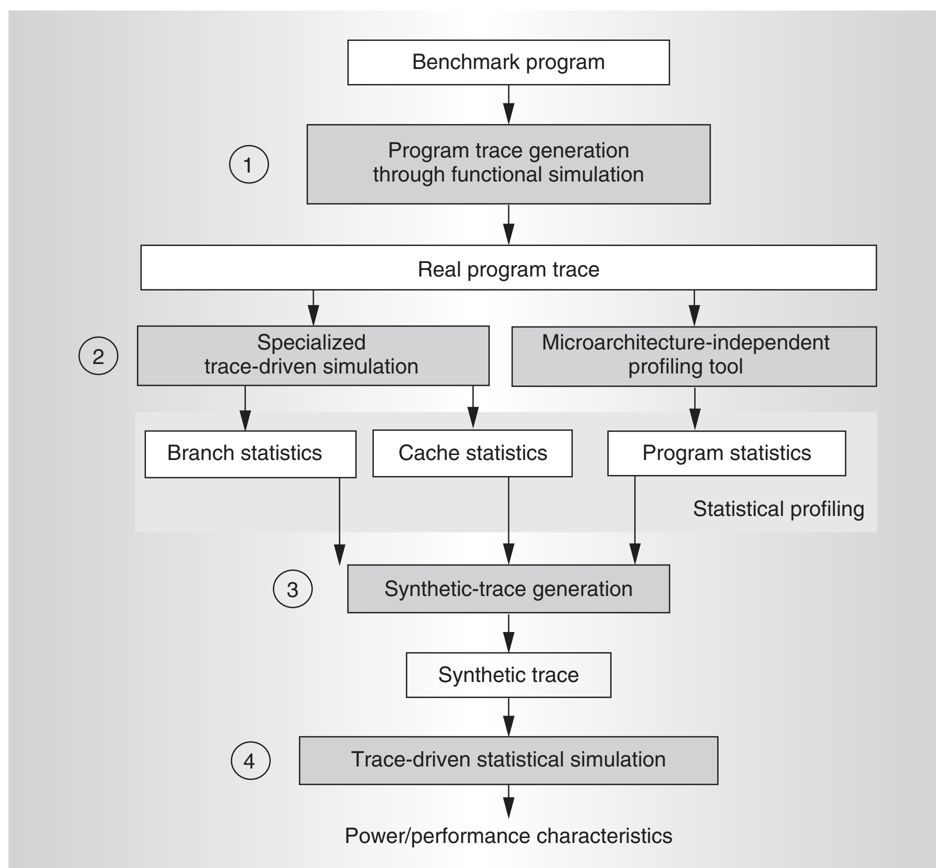

Figure 1. Statistical simulation general framework.

The drawback of using a simple statistical simulator is that the level of detail or the coverage during statistical simulation is less than for full and sampled trace simulations. For example, instructions can be collapsed into a single class; statistical simulation might treat all simple ALU instructions as a single class. As such, detailed measurement of specific instruction types becomes impossible. Obviously, this depends on the statistical simulation's level of detail; such simulations could model specific instruction types, if desired. However, modeling additional details lengthens development and evaluation times. Furthermore, statistical simulation is useful in identifying a region of interest for further analysis through slower, more detailed simulations.

A second drawback of statistical simulation is that employing it for a radically different system than the one for which it has been validated would require revalidating the model.

Based on experiments, we observe that statistical simulation attains good accuracy, requires relatively short development time, and provides very short evaluation time and good coverage. As such, statistical simulation is a useful addition to a designer's evaluation toolbox.

Statistical simulation

The overall process of statistical simulation consists of the four steps shown in Figure 1: program trace generation, statistical profiling, synthetic-trace generation, and trace-driven simulation.

Initially, we assume designers have generated a program trace via functional simulation (to avoid the storage of long trace files, the trace output can go directly into the profiling tools). Then, a microarchitecture-independent profiling tool and specialized cache/predictor simulators analyze the program trace.

The microarchitecture-independent profiling tool analyzes only the functional operation of instructions, extracting statistics that summarize program characteristics. Primarily, these characteristics are distributions,

including the instruction mix and the data dependencies among instructions. Such characteristics are independent of the microarchitecture on which the instructions execute; however, they do depend on the instruction set architecture and compiler.

The specialized cache/predictor simulation tools extract statistics about the branch and cache behavior from the program trace. These cache and predictor miss rates measure locality-related events that are difficult to model through microarchitecture-independent characteristics. There are several fast tools that can collect such cache and branch predictor statistics.

The complete set of statistics (for the program, branch, and cache) thus derived is a *statistical profile*. Once designers compute a statistical profile, the next step is to generate a synthetic trace with the same statistical properties, by construction, as the original trace that served as the statistical profile's source. The synthetic trace simulates on a trace-driven simulator. In simulating this synthetic trace, performance characteristics (like instructions issued per cycle) typically converge after a few hundred thousand instructions.

After initial collection of the statistics, designers can efficiently explore the complete design space. For instance, the instruction window size, the number of in-flight instructions, instruction execution latency, or the number of pipeline stages are all variable. In addition, the statistics themselves can be varied to touch on points in the program space that specific benchmarks might not cover.

Statistical profiling

Finding a viable statistical profile is key to successful statistical simulation. The performance characteristics of the synthetic trace should be comparable to those of the original trace from which it derives.[2] In our baseline model, a statistical profile includes four distribution classes:

- *Instruction mix.* The profile includes a distribution of instruction mix by type and the number of operands per instruction type. We typically use 10 instruction types, such as loads, ALU instructions, conditional branches.
- *Interinstruction dependencies.* This distribution measures the probability that an instruction depends on a preceding instruction through a register value or through memory. In our approach, we only consider read-after-write dependencies because other types of dependencies (write-after-write and write-after-read) are removed dynamically by today's superscalar, out-of-order microarchitectures. However, designers can add statistics for other types of dependencies if the hardware does not automatically remove them.
- *Cache behavior.* For statistical profiling, we measure the probabilities of L1 and L2 cache misses, and data and instruction cache misses.
- *Branch predictor behavior.* These characteristics include branch (target) prediction accuracies for the various branch classes (conditional branch, jump, function call, or return).

Synthetic-trace generation and simulation

Given the statistical profile, we can generate the synthetic trace. Generating a random number between 0 and 1, and then mapping through the cumulative distribution function selects a particular point on a distribution. The generation of a synthetic trace proceeds in the following order:

1. Determine the instruction type and the number of source operands.
2. For each instruction source operand, determine the preceding instruction that produces the value. This will create a read-after-write dependency between the preceding instruction and the current instruction. Do the same for memory operations, that is, make loads dependent on preceding stores (for example, a load follows a store to the same address).
3. If the current instruction is a load, determine whether it will cause a data cache miss.
4. If the current instruction is a branch instruction, determine whether the branch predictor will correctly predict the branch's outcome.
5. For each instruction, determine whether instruction fetch will cause an instruction cache miss.

As the generator produces synthetic instructions, it streams them into the simulator. One simplification of the synthetic trace simulator is that it does not need to model caches or branch predictors. During simulation, the simulator models the performance impact of these structures as follows:

- For a load miss, the simulator assigns an appropriate latency when the load executes. For example, in case of an L1 cache miss, it will assign the access time of the L2 cache. For an L2 cache miss, it will assign the memory access time.
- For a branch misprediction, the simulation will flush the pipeline when the mispredicted branch executes.
- For an instruction cache miss, the simulator will stop fetching instructions for a specified number of clock cycles.

Absolute accuracy

An important performance metric obtainable through simulation is the average number of instructions that execute per cycle or IPC. We use this metric to evaluate the overall accuracy of statistical simulation. To do so, we define the absolute IPC error:

$$\text{Absolute IPC error} = (IPC_s - IPC_r) / IPC_r$$

IPC_r is the reference IPC measure by detailed execution-driven simulation; IPC_s comes from statistical simulation. The IPC prediction error of a baseline statistical simulation model typically ranges from 10 percent for modest-resource superscalar microprocessors (at most 32 in-flight instructions and an issue width of 4) to 15 percent for wide-resource microprocessors (128 in-flight instructions and an issue width of 8).[1,3,4] There are several reasons for this inaccuracy.

- The synthetic-trace simulator is less detailed than the detailed architectural simulator and such a simplified model can introduce errors. This is expected and part of the tradeoff that dramatically reduces development and simulation time.
- In the current statistical simulation framework, we assume the statistical independence of the various program characteristics. For example, the data cache behavior is statistically independent of the instruction-level parallelism (ILP). In reality, a correlation might exist between various program characteristics, which can impact overall performance. Modeling this correlation can lead to increased accuracy as the "Program characterization" sidebar explains.
- Statistical simulation does not account for time-dependent program behavior. For example, we use one single statistical profile for the complete benchmark program run. However, measuring several statistical profiles and generating synthetic traces for different program phases might increase the accuracy of statistical simulation, but will consume additional evaluation time.

Relative accuracy

In the context of design space explorations, a performance model's relative accuracy is more important than its absolute accuracy (in this case, absolute IPC error, defined earlier). A measure of relative accuracy would indicate the ability of a performance estimation technique to predict performance trends, for example, the degree to which performance changes as the designer varies a microarchitectural parameter. So, if statistical simulation can provide good relative accuracy, then it is useful for making design decisions. For example, a designer might want to know if the performance gain due to increasing a particular hardware resource justifies the increased hardware cost.

We define relative error as

$$\text{Relative IPC error} = \frac{IPC_{s,B}}{IPC_{s,A}} - \frac{IPC_{r,B}}{IPC_{r,A}}$$

with $IPC_{s,A}$ and $IPC_{s,B}$ the IPC numbers of the synthetic trace when running on two different microarchitectures A and B; $IPC_{r,A}$ and $IPC_{r,B}$ are the similarly defined IPCs for reference traces. This definition quantifies the difference in IPC increase or decrease of the synthetic trace versus the IPC increase or decrease of the reference (real) trace.

Applications

Three applications of statistical simulation are uniprocessor performance modeling, high-

Program characterization

An interesting application for statistical simulation is program characterization. When validating the statistical simulation methodology in general and the characteristics included in the statistical profile in particular, program characterization makes clear which program characteristics must be in the profile for good accuracy. That is, this process distinguishes program characteristics that influence performance from those that do not. Two enhancements increase the accuracy of statistical simulation and reveal important program characteristics.

The first enhancement comes from the notion that cache misses occur in clusters or bursts. In other words, as a function of time, cache misses tend to occur in close proximity to other misses.[1] Modeling this clustered cache-miss behavior significantly improves the accuracy of statistical simulation. For example, enhancing statistical simulation by modeling intermiss gaps (instead of using average miss rates) reduces the 15 percent IPC prediction error for wide-resource microprocessors to approximately 10 percent.[2] That is, we model a bursty cache-miss behavior instead of a uniformly distributed one.

The high correlation of various program characteristics is the basis for the second enhancement. Correlating the instruction mix, the interoperation dependencies, cache-miss rates, and branch misprediction rates to the basic-block size can lead to significantly higher accuracy in performance prediction.[3] Figure A shows the IPC prediction error for several microarchitectural configurations by varying the window size, issue width, and the fetch queue size. Within each set, the 12 bars represent different improved characterization models, for instance, with additional complexity in the statistical profile or more correlation among various program characteristics. Each bar represents the average IPC error for the SPECint95 benchmark suite. This graph shows that the four rightmost bars in each set attain the highest accuracy for each microarchitectural configuration. These improved program models correlate cache and branch statistics and basic-block size to varying degrees and in various ways. For instance, one method of correlation is to measure different branch prediction and cache statistics for each basic-block size. The results in the rest of this article are the baseline results, that is, the results with the statistical profile as discussed in the main paper.

References

1. J. Voldman et al., "Fractal Nature of Software-Cache Interaction," *IBM J. Research and Development*, vol. 27, no. 2, Mar. 1983, pp. 164-170.
2. L. Eeckhout, "Accurate Statistical Workload Modeling," PhD thesis, Ghent Univ., Belgium, Dec. 2002; http://www.elis.ugent.be/~leeckhou/.
3. S. Nussbaum and J.E. Smith, "Modeling Superscalar Processors Via Statistical Simulation," *Proc. 2001 Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 2001), IEEE CS Press, 2001, pp. 15-24.

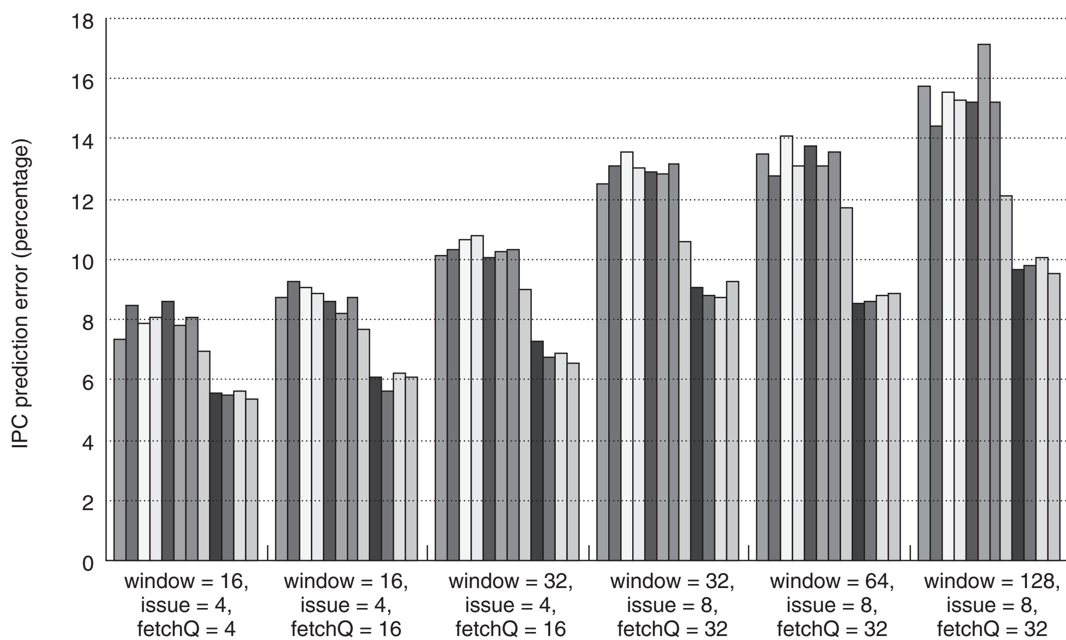

Figure A. Increasing the accuracy of statistical simulation reveals important program characteristics.

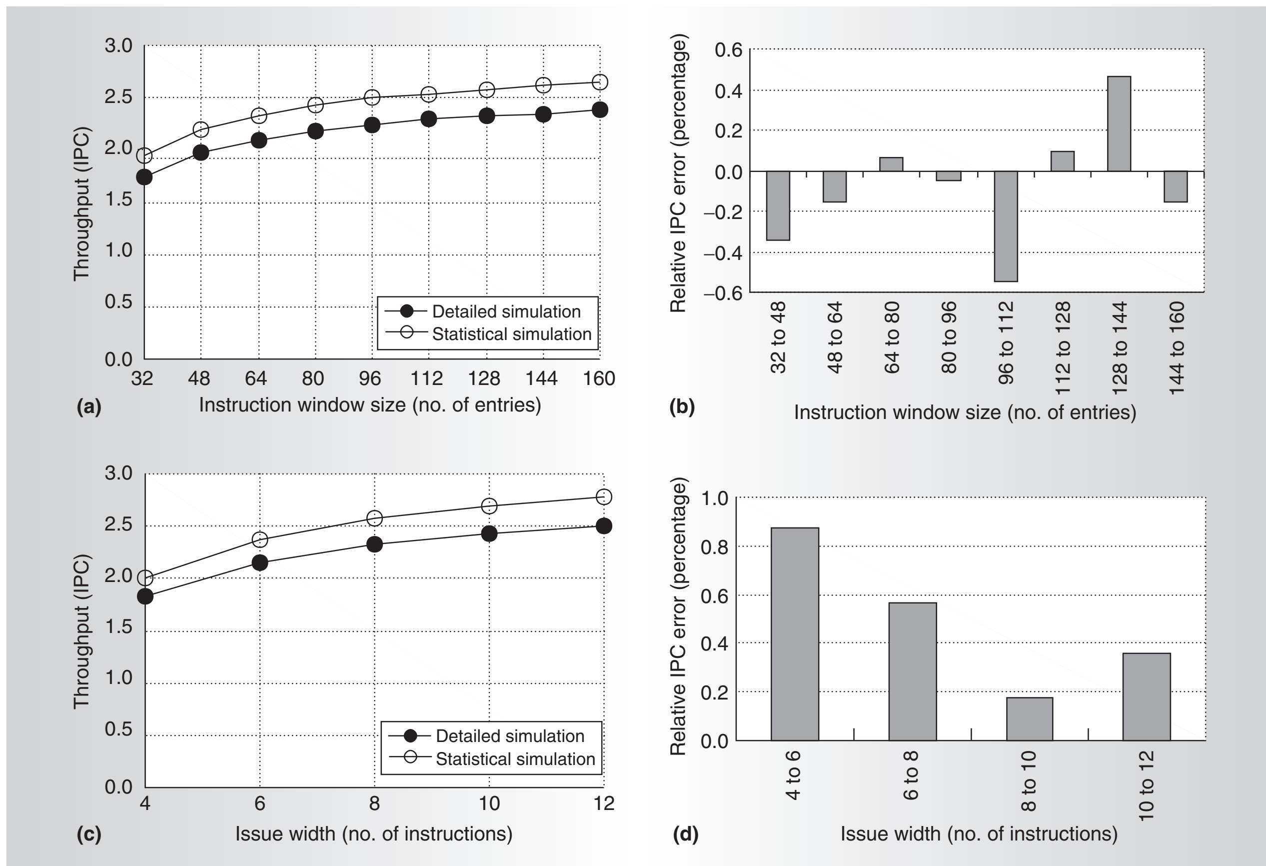

Figure 2. Relative performance prediction accuracy of statistical simulation. For an 8-issue out-of-order machine, statistical simulation yields these absolute (a) and relative (b) IPC errors as a function of instruction window size. For a 128-entry window out-of-order machine, simulation yields these absolute (c) and relative (d) IPC errors as a function of issue width.

level power modeling, and system evaluation.

Uniprocessor performance modeling

The most obvious application of statistical simulation is evaluation of uniprocessor performance. Statistical simulation does not aim to replace detailed cycle-accurate simulations. Rather, it aims to provide an efficient look at the design space, and to provide guidance for decision-making early in the design process.

Figure 2 illustrates the relative accuracy of statistical simulation for the SPECint95 benchmarks. Figures 2a and 2b show the average IPC for SPECint95 as a function of the instruction window size. Figures 2c and 2d show IPC as a function of the issue width, the maximum number of instructions that can be scheduled for execution per cycle. Figures 2a and 2c show absolute IPC numbers. These graphs show that statistical simulation results in a 10-percent absolute IPC prediction error. Although this absolute prediction error would be quite reasonable in many applications, statistical simulation attains a much better relative accuracy. Figures 2b and 2d show that the relative IPC error is less than 0.9 percent in all cases. Furthermore, the runtime for obtaining these data was many orders of magnitude faster than for detailed simulation.

High-level power modeling

Power dissipation and energy consumption are and will continue to be key issues when designing microprocessors. For laptop computers and handheld devices, battery life is a major design constraint. For high-end systems, power consumption is also becoming a major design issue:[14] higher clock frequencies, larger die sizes, and more complex microarchitectures significantly increase power consumption. This, in turn, increases packaging and cooling cost. Consequently, power consump-

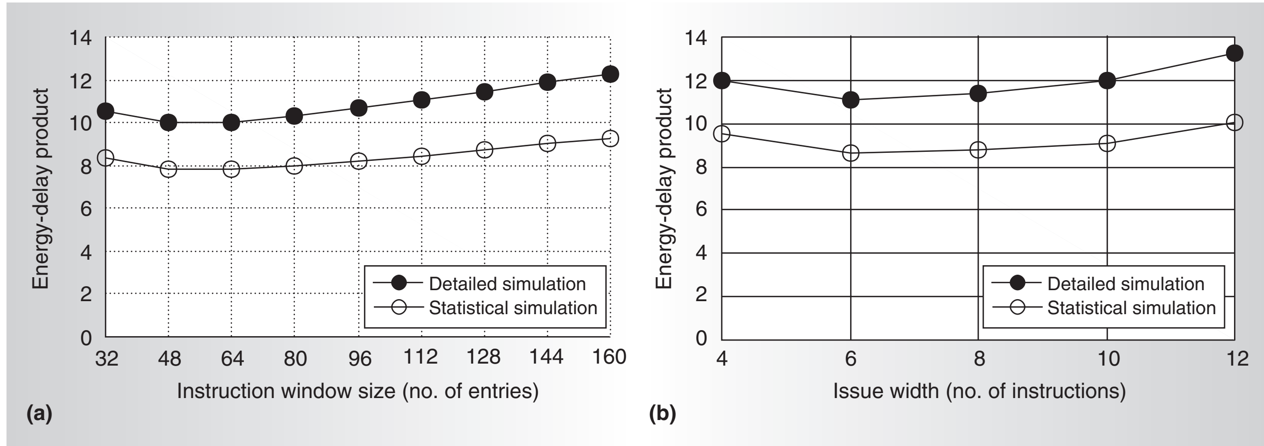

Figure 3. High-level power modeling through statistical simulation: Energy delay product as a function of instruction window size (a) and issue width (b).

tion requires consideration early in the design process. Recent work addresses this issue by integrating power modeling techniques into architectural simulators. In other words, power models of various processor structures are combined with counters that measure the activity of the structures at the architectural level and calculate the microprocessor's total power consumption. Wattch is an example of such a simulator.[15] Because these power modeling methodologies derive from detailed architectural simulations, they suffer from the same disadvantages: long simulation times and the need to develop complex simulation models.

We propose to meld statistical simulation with architectural power modeling, easily accomplishing this by integrating architectural power models into the synthetic trace simulator. In earlier work, we integrated Wattch[15] into the statistical simulation framework.[1] Using this tool, we can estimate the power dissipation of a microarchitecture executing a synthetic trace. The power prediction error is less than 5 percent for wide-resource processors; the relative accuracy is less than 1.6 percent. By combining the performance and power dissipation predictions, it's possible to measure a microarchitecture's energy-efficiency.[14] A metric that combines power and performance can measure energy efficiency; a popular power/performance metric for microprocessors is the energy-delay product (EDP):

$$EDP = \frac{energy}{instruction} \times \frac{cycles}{instruction}$$

$$= \left(\frac{1}{IPC}\right)^2 \times \frac{energy}{cycle}$$

Figures 3a and 3b show EDP for the SPECint95 benchmarks as a function of the instruction window size and the issue width. These graphs show that statistical simulation can identify a region of energy-efficient microarchitectures—those with potentially minimal EDP.

System evaluation

For larger systems containing several processors—symmetric multiprocessors (SMPs), clusters of computers, and even large networks—simulation time becomes a much bigger problem because of the number of processors that the designer must simulate simultaneously. Typically, benchmark problems for such systems are also much larger, and there might be additional design choices.

Modeling cache coherence, sequential consistency, private versus shared virtual memory pages, and processor synchronization extend the statistical simulation framework to SMP systems.[13] To do so, we computed the following additional program statistics (an earlier work provides a more elaborate discussion[13]):

- *Cache line ownership distribution.* Our simulation accounts for the probability that a store does not own the cache line it targets.
- *Store distributions.* We calculate two distributions, one measuring the numbers

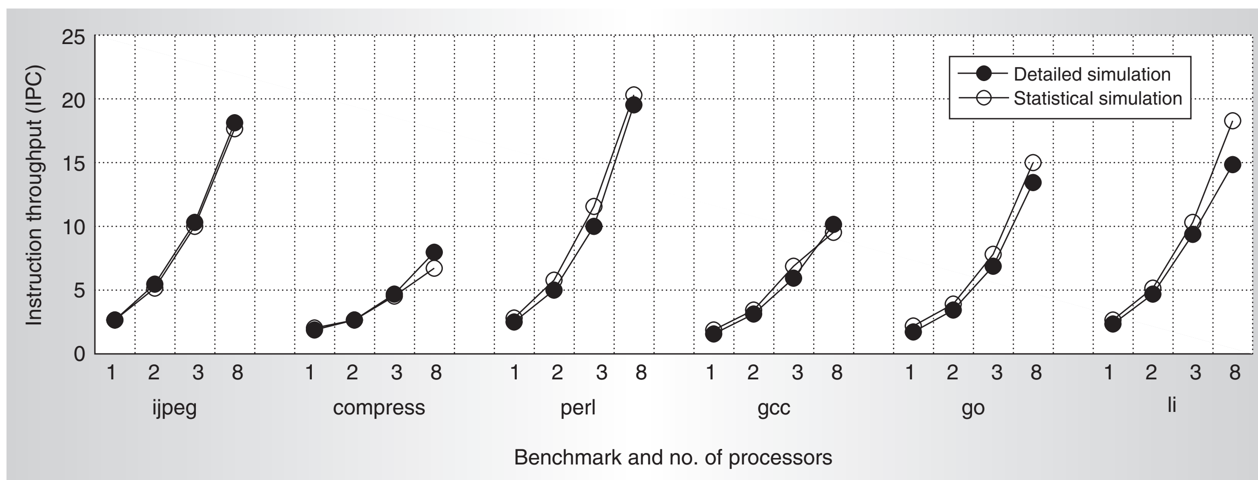

Figure 4. Reference (a) and estimated (b) instruction throughputs for multiprogrammed SPECint95 workloads as a function of the number of processors.

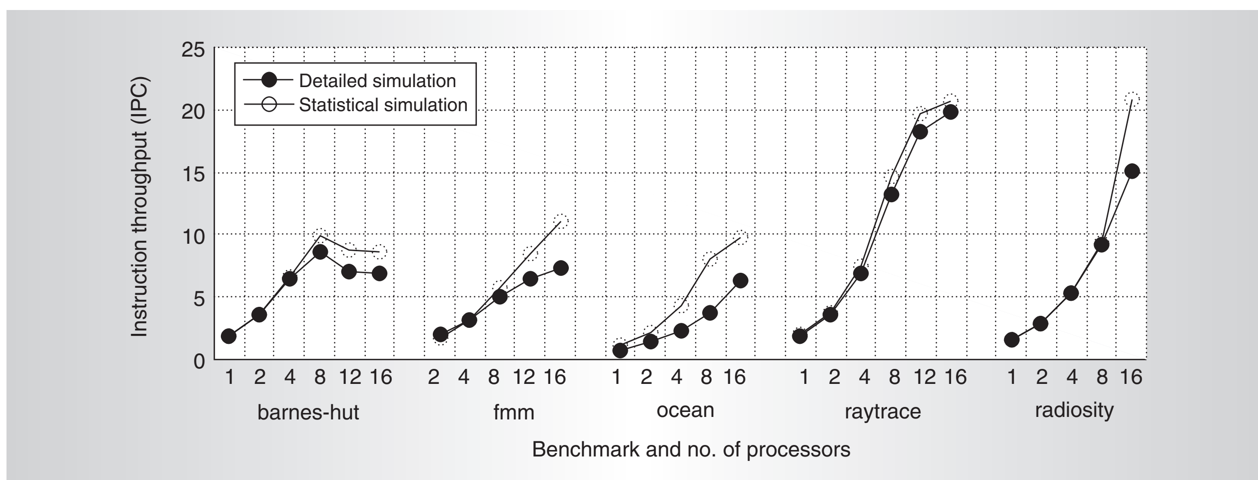

Figure 5. Reference (a) and estimated (b) instruction throughputs for parallel workloads as a function of the number of processors.

of consecutive stores to private pages and the other, to shared pages.

- *Lock distributions.* Lock distributions help us to statistically simulate synchronization through critical sections. Statistical simulation uses these distributions during statistical simulation to compute a target critical section. In addition, we maintain two statistical profiles: one updated when executing outside a critical section and one updated when executing inside a critical section. The synthetic trace generator then produces instructions based on these two statistical profiles.
- *Distributions for synchronization barriers.* Synchronization barriers cause two problems. First, they often occur at coarse granularity. As such, they are few in number but cover long time periods. Second, barriers typically involve all the processors, which is difficult to model in a statistically independent way in each processor. To address these issues, we scale down the amount of work done between barriers in proportion to the length of the statistical simulation.

Figures 4 and 5 show that it is possible to achieve accurate performance estimates on multiprogrammed workloads (SPECint95) as well as on synchronized parallel scientific

Workload space exploration

A statistical profile in statistical simulation consists of several program characteristics. These characteristics are easily variable and it's possible to measure the influence of these parameters on overall performance. However, varying the distributions in a statistical profile is sometimes impractical because of the numerous probabilities that require specification. A limited set of parameters that specifies complete program behavior would be useful. This is achievable within the statistical simulation framework by approximating measured distributions with theoretical distributions.

For example, the distribution of interoperation dependencies through register values exhibits power law properties—the probability density function (PDF) of this distribution follows a straight line when plotted in a log-log diagram.[1] This type of a distribution is a power law distribution or a *Pareto* distribution: $P[X = x] = \alpha x^{-\beta}$ with $1 > \alpha > 0$ (the intersect of the PDF with the y axis) and $\beta > 0$ (the slope of the PDF in a log-log graph). The results show that this approximation is more accurate than a previously proposed exponential approximation method. By fitting this power law to the measured distributions, you can estimate theoretical parameters α and β.

Figure B displays the SPECint95 as well as the Instruction Benchmark Suite (IBS) traces as a function of α and β, where α and β derive from the power law PDF of the interoperation dependencies. We can make two interesting conclusions from this graph. First, the interoperation dependencies seem to be quite different for the SPECint95 benchmarks than for the IBS traces. All but one of the IBS traces concentrate in the middle of the graph whereas the SPECint95 benchmarks are more toward the graph's left.

Second, this information can identify weak spots in a workload. For example, this graph reveals that none of these benchmarks have $0.28 < \alpha < 0.33$. There are two possibilities to address this lack of benchmark coverage: either search for real benchmarks or generate synthetic traces with the desired program properties. The latter option is easy because the program characteristics in a statistical profile can freely vary. In addition, these program properties can vary independently from each other.

Reference

1. L. Eeckhout and K. De Bosschere, "Hybrid Analytical-Statistical Modeling for Efficiently Exploring Architecture and Workload Design Spaces," *Proc. 2001 Int'l Conf. Parallel Architectures and Compilation Techniques*

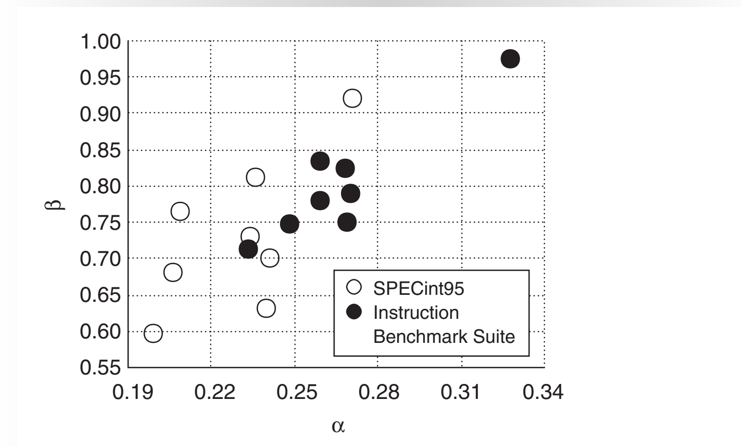

Figure B. Workload space as a function of α and β and for the SPECint95 and the IBS traces.

workloads (Splash 2). Figure 4 shows the detailed simulation and the statistical simulation instruction throughput for multiprogrammed workloads; in these initial experiments, we assume a simple one-bank memory system. In general, the performance estimates for multiprogrammed workloads running on the statistical model are fairly accurate (less than 15 percent error with respect to detailed simulation), but it shows larger errors in running the benchmarks **`compress`** (underestimated by 17 percent) and **`li`** (overestimated by 23 percent) on eight processors. This graph shows that statistical simulation accurately predicts important performance trends.

Figure 5 shows instruction throughput for parallel workloads and a four-bank memory system, giving results for the detailed and statistical simulations. In contrast to the earlier one-bank memory system, the four-bank memory system employs interleaving on a cache block basis, putting less pressure on the shared-memory system. These results show that accurate performance estimates are obtained for benchmarks **`barnes-hut`**, **`raytrace`**, and **`radiosity`**. However, it shows less accuracy for **`fmm`** and **`ocean`**. In spite of the reduced absolute accuracy for some benchmarks, this simple SMP model sufficiently identifies the number of processors after which there is a diminishing performance return. For example, detailed simulations of **`barnes-hut`** show a strong performance degradation from eight to 16 processors, which also appears in the statistical simulation results. Likewise, statistical simulation identifies the point of diminishing performance return for **`ocean`** after eight processors and for **`raytrace`** after 12 processors. Statistical simulation can predict performance trends sufficiently and accurately and is therefore a useful tool for quickly investigating design options. We obtained these results with only two to 10 minutes of simulation time.

The "Workload Space Exploration" sidebar presents another application of statistical simulation to system development.

Designing a new microprocessor is extremely time-consuming because of the numerous necessary simulations. Statistical simulation can speed up this design process significantly by identifying a region of interest in

a huge design space, permitting targeted analysis using detailed architectural simulations. This is true for uniprocessor as well as for multiprocessor power/performance modeling. In addition, statistical simulation is also useful for distinguishing important program characteristics from unimportant ones, and for identifying and addressing potential weak spots in the workload design. Now that we have a better understanding of the important program properties and processor characteristics, we plan to simplify performance modeling even further by tackling the problem of analytical modeling of superscalar processors. MICRO

Acknowledgments

Lieven Eeckhout is a postdoctoral fellow of the Fund for Scientific Research-Flanders (Belgium; F.W.O. Vlaanderen). James E. Smith and Sebastien Nussbaum were supported in part by National Science Foundation grant CCR-9900610, IBM, and Intel. The authors also thank the anonymous reviewers for their valuable comments, which significantly improved the quality of this article.

References

1. L. Eeckhout, "Accurate Statistical Workload Modeling," PhD thesis, Ghent Univ., Belgium, Dec. 2002; http://www.elis.ugent.be/~leeckhou.
2. V.S. Iyengar, L.H. Trevillyan, and P. Bose, "Representative Traces for Processor Models with Infinite Cache," *Proc. 2nd Int'l Symp. High-Performance Computer Architecture* (HPCA-2), IEEE CS Press, 1996, pp. 62-73.
3. S. Nussbaum and J.E. Smith, "Modeling Superscalar Processors Via Statistical Simulation," *Proc. 2001 Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 2001), IEEE CS Press, 2001, pp. 15-24.
4. M. Oskin, F.T. Chong, and M. Farrens, "HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Design," *Proc. 27th Ann. Int'l Symp. Computer Architecture* (ISCA 27), ACM Press, 2000, pp. 71-82.
5. T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *Computer*, vol. 35, no. 2, Feb. 2002, pp. 59-67.
6. R.A. Sugumar and S.G. Abraham, "Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Characterization," *Proc. 1993 ACM Conf. Measurement and Modeling of Computer Systems* (SIGMETRICS 93), ACM Press, 1993, pp. 24-35.
7. C. Bechem et al., "An Integrated Functional Performance Simulator," *IEEE Micro*, vol. 19, no. 3, May 1999, pp. 26-35.
8. J. Emer et al., "Asim: A Performance Model Framework." *Computer*, vol. 35, no. 2, Feb. 2002, pp. 68-76.
9. M. Vachharajani et al., "Microarchitectural Exploration with Liberty," *Proc. 35th Ann. ACM/IEEE Int'l Symp. Microarchitecture* (MICRO-35), ACM Press, 2002, pp. 271-282.
10. T.M. Conte, M.A. Hirsch, and K.N. Menezes, "Reducing State Loss for Effective Trace Sampling of Superscalar Processors," *Proc. 1996 Int'l Conf. Computer Design* (ICCD 96), IEEE CS Press, 1996, pp. 468-477.
11. R.E. Wunderlich et al., "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," *Proc. 30th Ann. Int'l Symp. Computer Architecture* (ISCA 30), ACM Press, 2003, pp. 84-95.
12. T. Sherwood et al.,"Automatically Characterizing Large Scale Program Behavior," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS X), ACM Press, 2002, pp. 45-57.
13. S. Nussbaum and J.E. Smith, "Statistical Simulation of Symmetric Multiprocessor Systems," *Proc. 35th Ann. Simulation Symp. 2002*, IEEE CS Press, 2002, pp. 89-97.
14. D. Brooks et al., "Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors," *IEEE Micro*, vol. 20, no. 6, Nov. 2000, pp. 26—44.
15. D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proc. 27th Ann. Int'l Symp. Computer Architecture* (ISCA 27), ACM Press, 2000, pp. 83-94.

Lieven Eeckhout is a postdoctoral researcher in the Department of Electronics and Information Systems at Ghent University, Belgium. His research interests include computer architecture, performance analysis, and workload characterization. Eeckhout has a PhD in Computer Science and Engineering from Ghent University.

Sebastien Nussbaum is a microprocessor architect at Sun Microsystems, working on the UltraSparc V processor, in Burlington, Mass. His research interests include statistical simulation of superscalar processors and multiprocessor systems. Nussbaum has a bachelor's degree in electrical and computer engineering from L'Ecole Supérieure d'Electricité, France; and a master's degree in computer engineering from the University of Wisconsin-Madison.

James E. Smith is a professor of Electrical Computer Engineering at the University of Wisconsin-Madison, where he teaches courses in computer architecture. His research interests include performance- and power-effective processors. Smith has a PhD in computer science from the University of Illinois. He is a member of IEEE and ACM.

Koen De Bosschere is a professor in the Faculty of Applied Sciences at Ghent University, where he teaches courses in computer architecture, operating systems, and declarative programming languages. His research interests include logic programming, system programming, parallelism, and debugging. De Bosschere has a PhD in applied sciences from Ghent University. He is a member of IEEE and ACM.

Direct questions and comments about this article to Lieven Eeckhout, ELIS-Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium; leeckhou@elis.ugent.be.

For further information on this or any other computing topic, visit our Digital Library at http://computer.org/publications/dlib.

SET
INDUSTRY
STANDARDS

Posix
gigabit Ethernet
enhanced parallel ports
wireless networks
token rings
FireWire

Computer Society members work together to define standards like IEEE 1003, 1394, 802, 1284, and many more.

HELP SHAPE FUTURE TECHNOLOGIES • JOIN A COMPUTER SOCIETY STANDARDS WORKING GROUP AT

computer.org/standards/