SimFlex: Statistical Sampling of Computer System Simulation

Timing-accurate full-system multiprocessor simulations can take years because of architecture and application complexity. Statistical sampling makes simulation-based studies feasible by providing ten-thousand-fold reductions in simulation runtime and enabling thousand-way simulation parallelism.

Thomas F. Wenisch

Roland E. Wunderlich

Michael Ferdman

Anastassia Ailamaki

Babak Falsafi

James C. Hoe

Carnegie Mellon University

•••••• Simulating a single CPU is typically thousands of times slower than the actual CPU. Full-system multiprocessor simulation, which involves simulating many CPUs, peripherals, and other system components on a single host, compounds the slowdown by another factor of 10 to 100. In other words, multiprocessor simulation is up to a million times slower than real hardware. This speed difference leads to prohibitively long turn-around times for simulating complete computer benchmarks—in particular, multiprocessor server benchmarks. (These benchmarks are often longer than their uniprocessor counterparts to compensate for nondeterministic thread scheduling and for perturbation effects from the operating system and I/O that can lead to significant short-term performance variations.)

Statistical sampling makes full-system multiprocessor simulation feasible by reducing simulation times by roughly a factor of 10,000. Sampling provides such drastic reductions by exploiting the homogeneity of application performance—application behaviors that repeat millions of times. By applying rigorous statistical methods, we can identify the minimal sample that assesses application performance with a desired confidence level. Our first work on this topic, Smarts, investigated statistical sampling of the System Performance Evaluation Cooperative (SPEC) CPU2000 benchmarks on uniprocessor simulators.[1] That work demonstrated that the nature of performance variability across measurement granularities favors a large sample of thousands of brief execution windows to minimize total simulation.

The primary challenge in realizing sampling's drastic acceleration lies in rapidly constructing the correct initial state for the large number of fine-grained performance measurements; we call this the *warming problem*. Our first solution to this problem uses a simplified simulation model to maintain architectural and selected microarchitectural state while fast-forwarding between measurements. Although this approach provides accurate results,[1,2] it does not realize the full potential of statistical sampling: The time spent fast-forwarding (making up 99 percent of experiment turnaround time) grows with

Published by the IEEE Computer Society

0272-1732/06/$20.00 © 2006 IEEE

benchmark length. Moreover, this warming approach precludes using the parallelism of computation clusters or multiprocessor simulation hosts to reduce turnaround time. Finally, the Smarts sample design and warming approach do not address multiprocessor simulation, where performance can depend heavily on which program phases execute concurrently on different CPUs.

Through our SimFlex research project, we have developed a new solution to the warming problem that decouples experiment turnaround time from benchmark length. In our enhanced warming solution, we store reusable, warm architectural and microarchitectural state in checkpoints. *Live points,* our storage-efficient implementation of checkpoint-based sampling for uniprocessor applications, allow highly accurate simulation of SPEC CPU2000 benchmarks in an average of only 91 seconds.[3] Moreover, checkpoint-based sampling lets individual performance measurements be independent. Thus, a 1,000-checkpoint sample allows 1,000-way simulation parallelism. Checkpoint independence also enables further sampling optimizations to reduce turnaround time, such as online results reporting and matched-pair sample comparison.

With our latest work, we extend the Smarts sample design to a critical class of multiprocessor server workload. We provide a new sampling population definition for *throughput applications*—the server side of client-server applications, such as the Transaction Processing Performance Council (TPC) database and SPECweb workloads. We leverage the random nature of transaction arrivals in these applications to construct a meaningful random sample despite deterministic simulation models. Furthermore, to obtain tractable samples for these applications, we measure and validate fine-grain progress metrics that are proportional to transaction completion rates. We describe our experiences with the SimFlex methodology using our full-system multiprocessor simulator, *Flexus,* and our multiprocessor checkpoint implementation, *flex points.* Flex points enable a multiprocessor simulation turnaround of only 10 to 100 CPU hours rather than the 10 to 20 CPU years required without sampling.

Smarts: Statistical sampling of SPEC CPU2000

The SPEC CPU2000 benchmark suite consists of 26 computation-intensive desktop and engineering applications, used primarily for uniprocessor performance comparisons. Our goal was to apply statistical sampling to accelerate the simulation of this benchmark suite while still producing accurate and reliable performance estimates.

Statistical sampling of simulation estimates the performance of benchmark applications (in cycles per instruction, energy per instruction, transaction throughput, and so on) on a simulated microarchitecture from measurements of a sample of the application's dynamic instruction stream. By choosing the measured sample according to established statistical sampling methods, simulation sampling can rely on statistical measures of confidence to validate that estimated results represent the full application's behavior.

Today's applications, as exemplified by SPEC CPU2000, exhibit homogeneous execution phases that can last for millions of instructions. Consequently, statistical sampling can reduce the total simulation effort required to estimate the performance of such applications. However, two broad challenges prevent the easy application of sampling to software-based microarchitecture simulators:

- using sampling theory to measure a minimal, but representative and unbiased sample that produces accurate estimates; and
- overcoming the practical constraints imposed by software-based simulators and processor architectures, such as fast-forwarding between measurements and warming to eliminate cold-start bias.

Sample design

To achieve the fastest measurement of a sample for a given accuracy target, we must optimize several sampling parameters. The sample size, n, is the number of measurements that will be taken. The required sample size is proportional to the square of the target metric's (x) coefficient of variation V_x. For example, for 95 percent confidence (2.0 standard deviations) of ±5 percent error,

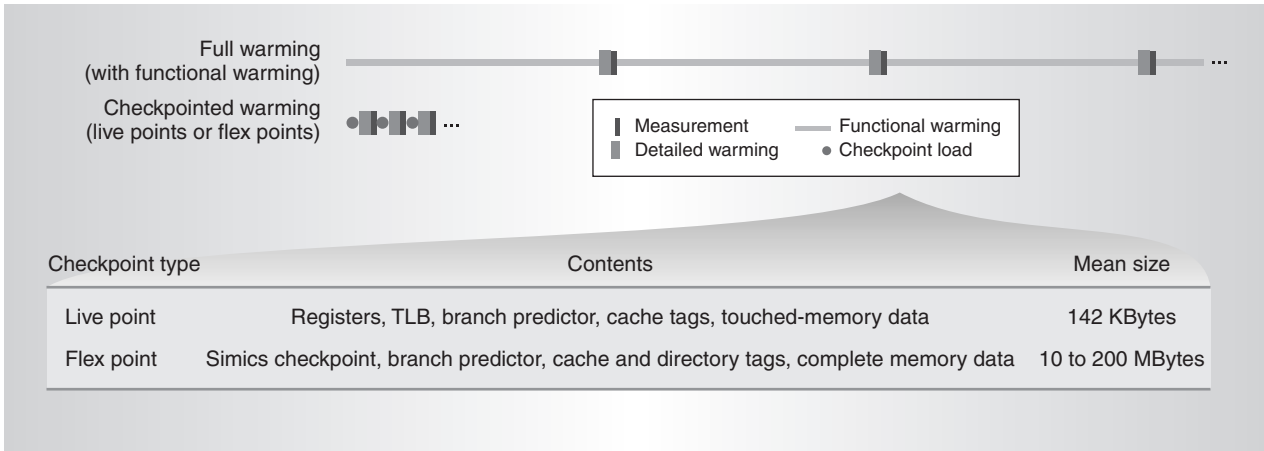| Checkpoint type	Contents	Mean size
Live point	Registers, TLB, branch predictor, cache tags, touched-memory data	142 KBytes
Flex point	Simics checkpoint, branch predictor, cache and directory tags, complete memory data	10 to 200 MBytes

Figure 1. Warming approaches for simulation sampling. Checkpointed warming greatly accelerates simulation sampling while maintaining the same accuracy as full warming. We developed *live points* for uniprocessor sampling and *flex points* for multiprocessor sampling.

$$n \geq \left(\frac{2.0}{0.05} V_x \right)^2 \qquad (1)$$

The size of each measurement, called a *sampling unit*, greatly affects the selection of n because larger sampling units average out short-term variation in performance, reducing V_x and requiring fewer measurements for an accurate estimate. To determine the best combination of n and sampling unit size, we need to know

- V_x, which we can obtain with preliminary samples, and
- the warming strategy to be used.

Warming

Although statistics provide us with probabilistic guarantees that estimated results are representative, these guarantees do not ensure that estimated results are error-free. Errors introduced into the individual measurements of a sample (for example, by the measurement methodology) are called *bias*, and are not accounted for by statistical confidence calculations. In simulation sampling, the most common cause of bias is the cold-start effect of unwarmed microarchitectural structures. Examples include assuming empty caches that result in incorrectly low performance estimates, and empty interconnect networks that produce overly optimistic performance.

The relationship between performance variability and measurement granularity leads to a sampling framework that minimizes instructions simulated by measuring a large number of brief simulation windows—for example, 10,000 windows of 1,000 instructions each. Thus, the primary design challenge lies in devising a strategy for constructing accurate initial state rapidly. For each measurement, the simulator must construct both architectural state (such as register and memory values) and microarchitectural state (such as pipeline components and the cache hierarchy) to avoid the bias of a cold start.

Functional warming

Figure 1 shows Smarts' two-tiered strategy for constructing every measurement's initial state, a combination of *detailed warming* and *functional warming*. Before each measurement, Smarts warms microarchitectural structures for which current state reflects the history of a small, bounded set of recent instructions—such as the reorder buffer or issue queue. This takes place through detailed warming: brief simulation (for example, a few thousand instructions) with the complete detailed performance model sufficient to warm such small structures.

The second component of the Smarts warming strategy, *functional warming*, addresses state updates between two measurements. Smarts functionally simulates each instruction to update architectural state. In addition, Smarts continuously updates structures with microarchitectural state that have

long or unpredictable warming requirements—caches, translation look-aside buffers (TLBs), and branch predictors. These structures sometimes require millions of instructions to warm, and cannot be warmed sufficiently by a brief, detailed warming period. Caches in particular have unpredictable requirements; functional warming eliminates the need to determine these requirements.

Smarts results

We evaluated the Smarts framework in the context of a wide-issue, out-of-order superscalar simulator running the SPEC CPU2000 benchmark suite. We created SmartSim, an implementation of Smarts, by modifying SimpleScalar 3.0's sim-outorder and Wattch 1.02 to support systematic sampling. The results of our evaluations demonstrated the following:

- SmartSim achieves an actual average error of only 0.64 percent on cycles per instruction (CPI), and 0.59 percent on energy per instruction, by simulating fewer than 50 million instructions in detail per benchmark. A recent survey of simulation sampling approaches corroborates that the Smarts simulation sampling approach provides the highest estimation accuracy.[2]
- By simulating exceedingly small fractions of complete benchmarks, SmartSim with functional warming reduces simulation time by 35 to 60 times relative to full-stream simulation with sim-outorder.

Live points: Checkpoint-based sampling

Although functional warming enables accurate performance estimation, it limits Smarts' speed, occupying more than 99 percent of simulation runtime. Functional warming dominates simulation time because Smarts must functionally simulate the entire benchmark's execution, even though it will simulate only a tiny fraction of the execution using detailed microarchitecture timing models.

The second shortcoming of the original Smarts framework is that functional warming requires simulation time proportional to benchmark length rather than sample size. As a result, the overall runtime of a Smarts experiment remains constant even when we reduce the measured sample size—for example, by

relaxing an experiment's statistical confidence requirements. Moreover, functional warming time will increase with the advent of new benchmark suites, such as SPEC CPU2006, that lengthen benchmarks to scale with hardware performance improvement.

Live points provide an alternative to functional warming that reduces simulation turnaround time without sacrificing accuracy. A live point stores the necessary data to reconstruct warm state for a simulation sampling execution window. Although modern computer architecture simulators frequently provide checkpoint creation and loading capabilities,[4] current checkpoint implementations have two limitations:

- They don't provide complete microarchitectural model state.
- They cannot scale to the required checkpoint library size (about 10,000 checkpoints per benchmark), which would require multiple terabytes of storage.

We address the first limitation by storing only selected microarchitectural state in live points, an approach we call *checkpointed warming*. The key challenge of checkpointed warming lies in storing microarchitectural state such that live points can still simulate the range of microarchitectural configurations of interest. Fortunately, with the exception of the branch predictor and memory hierarchy, most microarchitectural state can be reconstructed dynamically with minimal simulation (a few thousand instructions of detailed warming), and thus need not be stored. For the exceptional structures, researchers can often place limits on the configurations of interest (for example, through trace-based studies). We've designed checkpointed warming to reproduce these structures under user-specified limits.

We reduce the size of conventional checkpoints by three orders of magnitude through storing in live points only the subset of state necessary for limited execution windows, an approach we call *live state*. Live state exploits the brevity of simulation sampling execution windows (thousands of instructions) to omit most state. Figure 1 illustrates how live points replace functional warming and details live-point contents.

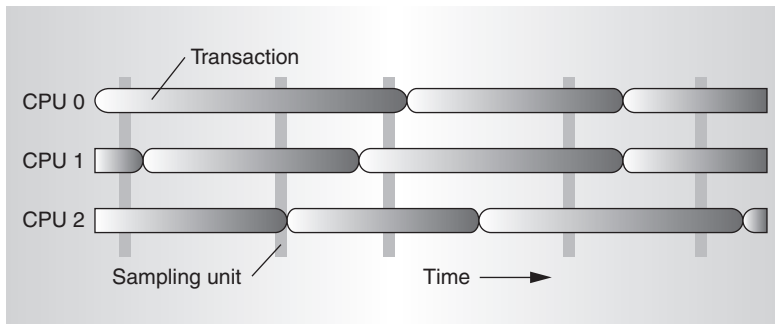For an eight-way out-of-order superscalar

Figure 2. Sampling a throughput application. Each sampling unit measures the performance of a random interleaving of transactions. The sample's population is then defined as the runtime that effectively covers all reachable transaction interleavings.

processor, live-point simulation sampling is more than 250 times faster than the original Smarts framework (on average, 91 seconds per benchmark) while maintaining the same estimated CPI error (±3 percent with 99.7 percent confidence). Although functional warming produces an aggregate of 36 Tbytes of state while sampling the SPEC CPU2000 suite, a gzip-compressed SPEC CPU2000 live-point library supporting 1-Mbyte caches requires only 12 Gbytes of storage.

Sampling multiprocessor throughput applications

Uniprocessor benchmarks that have a finite length, such as SPEC CPU, make selecting a uniform sample easy. In a uniform sample, each portion of the dynamic instruction stream must have an equal probability of being measured. When the benchmark's length is known, we can collect such a sample using systematic or random sampling. The benchmark's complete dynamic instruction stream is the population from which we take samples.

In multiprocessor benchmarks, we cannot define the population in terms of the dynamic instruction stream. Instruction interleaving across processors varies over multiple benchmark runs and can cause changes in the dynamic instruction stream as races (for locks, for example) resolve differently on different runs. When measuring on real hardware, we account for variability in interleaving by running a benchmark repeatedly or for an extended time window to exercise the possible interleavings. Hence, we define the population as the set of all reachable instruction inter-

leavings and their occurrence probabilities. For general multiprocessor applications, it isn't clear how to construct this population using a simplified (often deterministic) simulation model like that used in live-point creation.

Fortunately, the largest commercial multiprocessor market is for servers that run *throughput applications*—such as online transaction processing (OLTP), decision-support queries, and Web serving—for which we can construct the population of interleavings efficiently. In throughput applications, a server process satisfies a sequence of arriving transactions, queries, or requests. (For the remainder of this article, we use the term *transaction* to include all of these.) Throughput application benchmarks consist of long (or unbounded) sequences of randomly arriving transactions. Because transactions arrive randomly, a single run will cover the range of possible transaction interleavings. We draw our sample by selecting measurement locations over a time window that has proven reliable on real hardware—for example, about 30 seconds.[5] Figure 2 depicts sampling for a throughput application.

We typically report the performance of throughput applications in terms of transactions per second. With our definition of the population in hand, we could naively sample this metric. However, transactions are too long for a simulator to execute, and their completion rate has a high coefficient of variation. Figure 3 plots the coefficient of variation for transaction throughput, $V_{transaction}$, for several applications measured on a real four-way multiprocessor system. We include IBM DB2 running the TPC-C OLTP benchmark with a 6.4-Gbyte database in two configurations (on disk and memory-resident in a RAM disk), and an Apache Web server running SPECweb99. Points on the plot correspond to sampling units of a logarithmically increasing number of transactions (labeled every factor of 10), with the mean time to complete those transactions indicated on the *x*-axis. Transactions' high variance and long runtimes imply that sampling transaction throughput would require simulating seconds to minutes of real CPU time to obtain high confidence results. Seconds of real time, however, translate to years of simulation time on a full-system multiprocessor simulator.
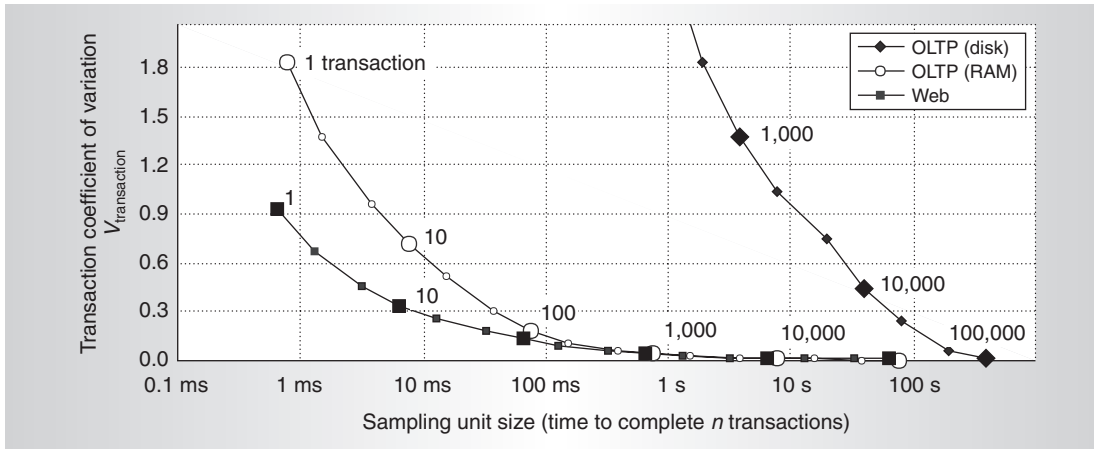
Figure 3. Variance of transaction throughput, measured on a real four-way system. Sample size is quadratically related to coefficient of variation. For example, measurements of 10 transactions each for OLTP (RAM) yield a $V_{transaction}$ of 0.7. Thus, from equation 1, to achieve a 95 percent chance of ± 5 percent error requires about 800 10-ms measurements.
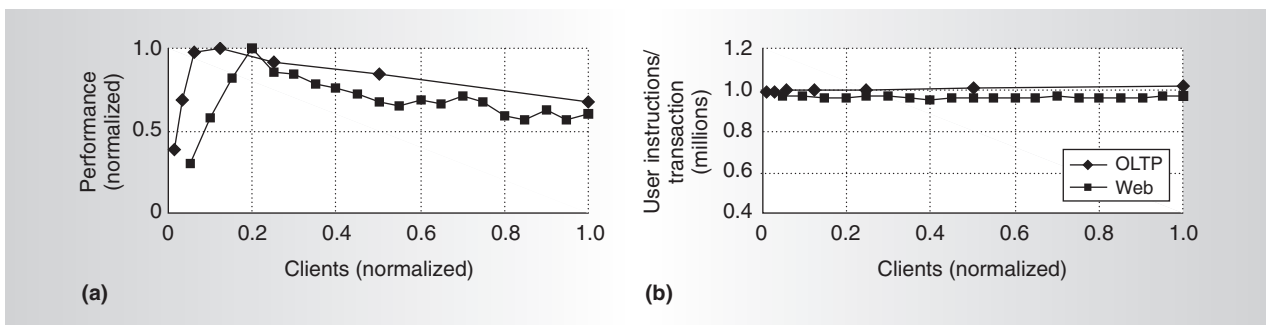

Figure 4. Throughput (a) and user-mode instructions per transaction (b) for commercial application benchmarks. Each OLTP and Web transaction completes with an essentially constant number of user-mode instructions. This constant holds despite large variations in benchmark performance with increasing numbers of clients.

To address this problem, we seek an alternative performance metric that is proportional to transaction throughput but has lower variance at smaller measurement sizes. Although the time to complete a particular number of transactions varies greatly, the amount of work the database or Web server process must perform to complete a certain transaction type does not vary. As a result, the rate at which user-mode instructions complete is linearly proportional to transaction throughput. Hankins et al. first observed this linear relationship running the TPC-C benchmark on Oracle,[6] and we have found that this result applies across throughput applications. Figure 4 shows that the number of user instructions per transaction remains constant despite a transaction throughput that varies by a factor

of 2 as we vary benchmark configuration. This relationship holds because when applications are not making forward progress, they yield to the operating system (for example, to the operating system idle loop or spin loops in operating system locking primitives).

The linear relationship between user-instruction throughput and transaction throughput lets us sample user-instructions per cycle (U-IPC) to assess transaction throughput. We define U-IPC as the number of user-mode instructions that commit divided by all measured cycles. The commonly used metric of instructions per cycle cannot be used because it is not proportional to transaction throughput, as it includes many system instructions that do not contribute to forward progress. Sampling U-IPC is advantageous because the variance of
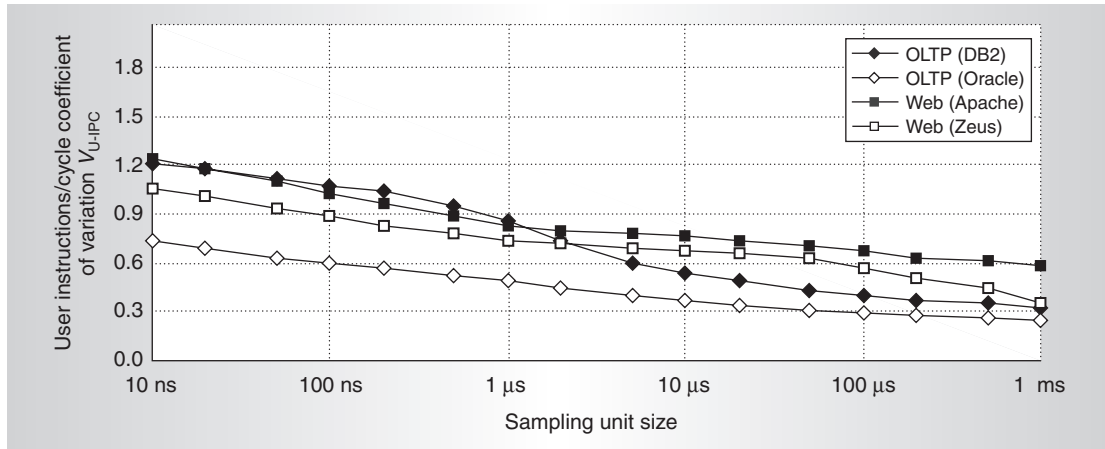
Figure 5. Variance of user-instruction throughput, measured on a simulated 16-way system. The sampling unit size for a particular $V_{\text{U-IPC}}$ is three orders of magnitude smaller than for the same $V_{\text{transaction}}$ (compare Figure 3). Thus, sampling U-IPC requires 1,000 times less total simulation.

U-IPC is lower than that of transaction throughput at far smaller measurement sizes. Thus, we can simulate shorter sampling units while achieving the same confidence. Figure 5 plots the coefficient of variation for U-IPC across a range of measurement sizes. Using U-IPC as the target metric saves three orders of magnitude in simulation time over using transaction throughput.

Flex points

Our full-system simulation infrastructure, Flexus, builds on top of Virtutech Simics, a commercially available computer architecture simulation tool.[4] Simics emulates a complete multiprocessor computer system, including all peripheral devices, and is capable of booting and executing unmodified commercial software. However, Simics provides only functional simulation; it does not attempt to model the passage of time accurately. Flexus, which furnishes Simics with microarchitectural hardware and timing models, can model uniprocessor, chip multiprocessor, and distributed-shared-memory multiprocessor systems at various levels of timing fidelity, ranging from functional warming (cache and branch predictor state only) to detailed timing of superscalar out-of-order processor cores.[7]

In its most detailed mode, each CPU that Flexus simulates is as much as 100,000 times slower than actual hardware. However, Flexus' functional warming mode is only 100 to 1,000 times slower than hardware, within a

factor of 10 of Simics' top speed. Because of this enormous performance gap, simulation sampling is essential to meaningful performance evaluation of commercial applications.

To support efficient sampling, we integrate our checkpointed-warming approach with Simics' native capability of storing architectural state checkpoints. We store microarchitectural structures similar to those we store in live points, such as caches and branch predictors, along with the coherence directory state. We call these augmented Simics checkpoints *flex points*.

As with live points, we generate flex points rapidly by using a simplified simulation model. To produce a valid sample, we require the simplified model to produce the same population (probability distribution of interleavings) as detailed timing simulation. However, some multiprocessor applications adapt program behavior to their performance, which can lead to different populations. We have not observed differing populations due to adaptation in the workloads we study when the CPUs are fully saturated under both timing models.

Unfortunately, we cannot employ live state to optimize the storage requirements of flex points. First, we rely on Simics to save and restore architectural state. Second, unlike uniprocessor applications, the instruction stream of multiprocessor applications depends on precise timing, such as the outcome of a data race. Therefore, it is not possible to identify the instruction stream of a timing-accurate simulation through functional warming, which is
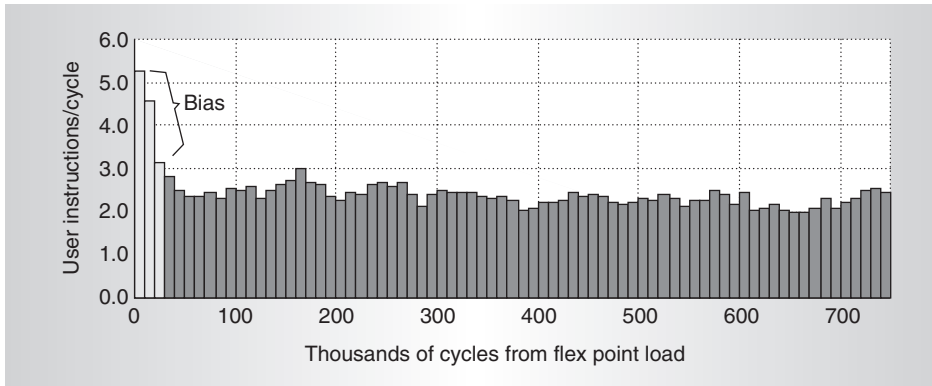
Figure 6. Empirical warming determination for the OLTP (DB2) benchmark. Each bar in this graph represents the mean U-IPC from 50 sampling units. The sampling units are 10,000 cycles long, and their offset from a flex-point load is plotted on the *x*-axis. This benchmark requires at least 30,000 cycles of detailed warming.

necessary for live state. However, we do exploit Simics' ability to store only the change in memory and disk state between consecutive flex points to minimize storage requirements.

As with our uniprocessor approaches, we use a brief detailed simulation period to refresh microarchitectural structures whose state the flex points do not store. However, because of the larger and more complex queues in multiprocessor systems (for example, queues in the interconnection network), analysis of worst-case detailed warming requirements is more challenging than in uniprocessors. Nevertheless, these queues' warming requirement differs fundamentally in time scale from caches because the history reflected in a queue's state cannot persist indefinitely as it can in a cache. The short time scale over which queues warm induces a steep slope in cold-start bias that we can detect empirically.

Cold-start bias manifests as a correlation between performance metrics such as U-IPC and how far simulation has progressed from the initial state captured in a flex point. Figure 6 plots U-IPC against elapsed cycles for an example application. Each bar on this graph shows the U-IPC for contiguous 10,000-cycle sampling units averaged over a sample of 50 flex points. The *x*-axis indicates the offset of the sampling units from a flex-point load. If there were no cold-start bias, all bars would be approximately the same height. However, the first three bars (from the left), corresponding to the first 30,000 cycles of simulation from each flex point, show a clear positive perfor-

mance bias. The caches become warm immediately upon the load of a flex point, but queues in the interconnection network and within the processor cores are initially empty, which, in this case, leads to higher than average performance at first. For each new combination of workload and system configuration, we examine plots like Figure 6 to determine an appropriate detailed warming interval. The OLTP benchmark in this example requires at least 30,000 cycles of detailed warming after the load of each flex point.

The SimFlex experimental procedure

Our experimental procedure is effective for uniprocessor and multiprocessor benchmarks, and several optimizations reduce experiment turnaround time.

Preparing a new workload

To prepare a workload for study, we must investigate its performance variability to design an optimal sample—one that minimizes total simulation for a desired confidence level. Then we construct a live-point or flex-point library for the optimal sample. The following steps detail how we construct these libraries.

1. *Create preliminary sample of live points or flex points.* First, we construct a 30-point preliminary sample, which we use to characterize the application's variability and warming requirements. A 30-point sample is insufficient to provide high-confidence simulation results, but
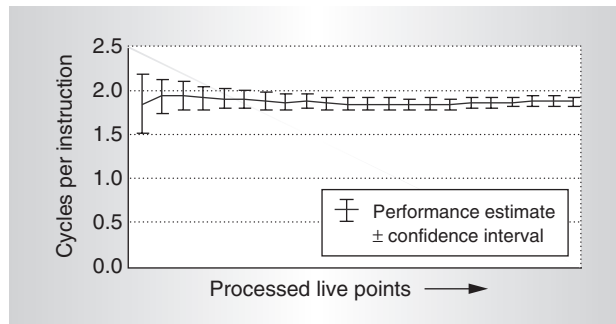
Figure 7. Online results example. Results converge toward their final values, and confidence improves as more live points are processed. Flex points can also produce online results when simulated in random order.

fixed-length benchmarks, such as SPEC CPU2000.

The live-point or flex-point library is now ready for experimentation. If we drastically alter microarchitecture configuration, we repeat steps 2 and 3 with a subset of the library to ensure that warming requirements and the optimal sampling unit size have not changed.

Optimizations

When using a live-point or flex-point library, we can apply several optimizations to further reduce the turnaround time of a particular simulation experiment.

Parallel simulation. Each live point or flex point can be simulated independently. Live-point and flex-point independence allow massive simulation parallelism over many host machines (up to the sample size, typically hundreds or thousands of points), reducing the overall time to obtain results.

Online results. Live-point and flex-point independence affords us a second opportunity to improve our experimental methodology. Each complete library forms an unbiased uniform sample of a workload. A randomly selected subset of a library also forms a uniform sample. Thus, if we process points in a random order, after each point is simulated, the points processed thus far form an unbiased sample of the full workload. We can use this property to provide a continual update of estimated results as points are processed.[3,8] Figure 7 illustrates how a mean estimate converges and confidence intervals tighten as we process additional live points.

Matched-pair sample comparison. Many computer architecture studies compare an experimental design against a base case. In such studies, relative rather than absolute performance is the key evaluation metric. Frequently, the change in performance from design A to design B varies less than the absolute performance of either design. Figure 8 illustrates this concept with live points. Each vertically aligned pair of points in Figure 8a represents performance data for the same sampling unit measured under two microarchitecture designs. Figure 8b plots the performance deltas for each

typically provides a good estimate of target metric variance.

2. *Determine detailed warming requirement.* We measure the preliminary sample for intervals several times longer than our expected detailed warming at a measurement granularity several times finer than our expectations for sampling unit size. Using these extended, fine-grained measurements, we perform the empirical warming analysis illustrated in Figure 6. Although this analysis is specific to both microarchitecture and workload, we find that detailed warming requirements tend to vary little and depend on only a few key microarchitecture parameters (such as store buffer size and memory controller queue depths).

3. *Determine optimal sample design.* Once we have identified a detailed warming interval, we determine the best sample design. We estimate sample size for various sampling unit sizes by summing the data from step 2 to construct various unit sizes, and then calculate coefficients of variation across the preliminary sample. We can then use equation 1 to compute sample size for any confidence. The optimal sample design minimizes the product of sample size and the sampling unit size plus warming interval.

4. *Create live-point or flex-point library.* With a desired sample size in hand, we can now launch live-point or flex-point creation to spread the final sample over a known-representative execution interval (such as 30 seconds) or the complete execution of
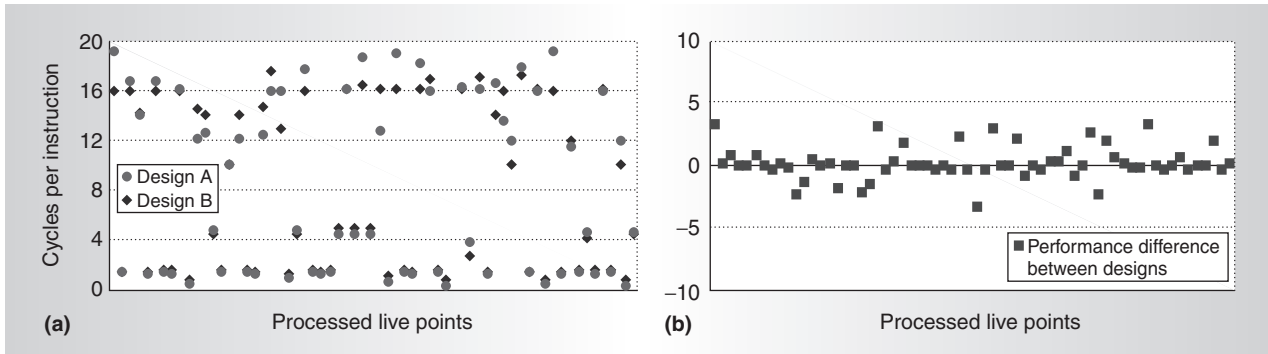
Figure 8. Matched-pair comparison example: performance data for the same sampling unit measured under two microarchitecture designs (a) and performance deltas for each sampling unit (b). The lower variability of performance deltas reduces sample size by 3.5 to 150 times.

Table 1. Sampling parameters for simulation sampling experiments.

Application	Detailed warming	Sampling unit size	Target confidence interval (%)
SPEC CPU	2,000 instructions	1,000 instructions	99.7 ± 3
Multiprocessor applications*	100,000 cycles	50,000 cycles	95 ± 5

    * Multiprocessor applications are OLTP (DB2), OLTP (Oracle), Web (Apache), and Web (Zeus).

Table 2. Simulation sampling with SimFlex.

Application	Runtime on real hardware	Simulation time without sampling (CPU yrs.)	Absolute performance estimate			Typical relative performance estimate	
			Typical sample size	Simulation time (CPU hrs.)	Live-point library size (Gbytes)	Typical sample size	Simulation time (CPU hrs.)
SPEC CPU	1.5 hrs.	0.6	8,816*	0.025*	12	3,511*	0.01*
OLTP (DB2)	30 s.	10-20	301	39	28	193	25
OLTP (Oracle)	30 s.	10-20	100	25	4	30	7
Web (Apache)	30 s.	10-20	774	140	74	91	17
Web (Zeus)	30 s.	10-20	289	58	19	183	37

    * Per reference input (mean). There are 45 reference inputs in the SPEC CPU2000 suite.

sampling unit, using the same vertical scaling.

We can exploit the reduced variability of relative performance through a sampling procedure called *matched-pair sample comparison*,[3,9] by which we measure the same points for both designs and then build a confidence interval directly on the delta performance observed at each point. The reduced variability lets us achieve the same confidence on relative performance with a smaller sample than required for absolute performance estimates.

Evaluation

Tables 1 and 2 summarize our experiences using the SimFlex methodology to estimate performance of both SPEC CPU2000 and commercial multiprocessor applications. (We describe our experiments with SPEC CPU2000 fully elsewhere.[3]) The multiprocessor simulation results come from our investigations of spatial memory streaming (SMS), a hardware mechanism designed to stream spatially correlated data from main memory and secondary caches to a processor's primary cache

ahead of explicit processor requests.[10] These experiments model a 16-way, distributed-shared-memory multiprocessor loosely based on the HP GS1280 and Compaq Piranha designs. The relative comparison contrasts a system with SMS hardware to a base system. Our commercial applications are scaled to use 10-Gbyte data sets. (For complete details of the experiments, system models, and workload configurations, see our latest work.[3,10])

Simulation without sampling is simply infeasible, requiring CPU years to provide a high probability of accurate results. In contrast, the SimFlex methodology enables high-confidence estimates of application performance in minutes for SPEC CPU2000 and with only a few CPU days for commercial applications. Matched-pair comparison further reduces turnaround time by up to a factor of 7 for design comparison experiments. By parallelizing simulations on a compute cluster, we can complete even multiprocessor experiments in only a few hours.

Unfortunately, because flex points are built atop Simics' checkpointing mechanism, each flex point must include either a complete snapshot of system state, or delta images of memory and disk state from another flex point. These Simics checkpoints require from 10 Mbytes to 200 Mbytes. As a result, a flex-point library's disk space requirements are high—as much as 74 Gbytes. However, even on real systems, commercial applications have high disk space requirements (more than 25 Gbytes for a 100-warehouse TPC-C installation). Flex-point library storage requirements are well within the capabilities of modern high-capacity disks.

Further sampling optimizations

Are there alternative sampling methodologies or optimizations that we could apply to further reduce SimFlex's simulation time or storage requirements? Using representative sampling and considering multiple execution paths from one flex point offer two interesting possibilities.

Representative sampling

Our proposal uses a uniform sampling approach that takes measurements randomly or systematically from the instruction stream. However, we might reduce the required amount of measurement if we can identify low-variance program phases.[11] Sampling approaches that choose measurement locations according to some selection criteria are called representative sampling.

Hamerly et al.[12] study program phase identification for representative sampling with Sim-Point, which relies on a clustering algorithm to identify instruction stream regions that have similar basic-block occurrence frequencies. SimPoint selects measurement locations from each program phase identified by clustering basic-block, relative-frequency vectors. This approach cannot achieve the high level of accuracy and reliability of statistical sampling. However, SimPoint does not require a warming strategy as carefully calibrated as SimFlex because it uses fewer, but larger, measurements, which amortize bias. Sim-Point is most effective on uniprocessor simulations, for which its typical IPC error, between 1 percent and 10 percent, is acceptable.

Uniform sampling is simpler in theory than representative sampling, which becomes critical when sampling multiprocessor benchmarks. Researchers have not yet determined how to profile multiprocessor applications to effectively identify program phases that can represent the complete execution. Our SimFlex methodology does not require program phases to be identified, and therefore can be applied to both uniprocessor and multiprocessor applications.

Multiple execution paths from one flex point

Past multiprocessor methodology research has noted that small timing variations can cause two executions that start from identical initial conditions (for example, the same flex point) to follow drastically different execution paths.[5] Such variation is possible because even a single cycle's difference in memory latency might cause a race for a lock to resolve differently, which can induce changes in thread scheduling or cause other large-scale differences in execution paths.

The possibility of exploring multiple execution paths from a single flex point is of interest in simulation sampling because it might let us increase sample size without having to create and store additional flex points. Alameldeen and Wood suggest launching multiple simulations from a single checkpoint and randomly perturbing main memory latency, adding or subtracting several nanoseconds per request.[5] Unfortunately, our investigation indicates that varying memory latency typically does not lead
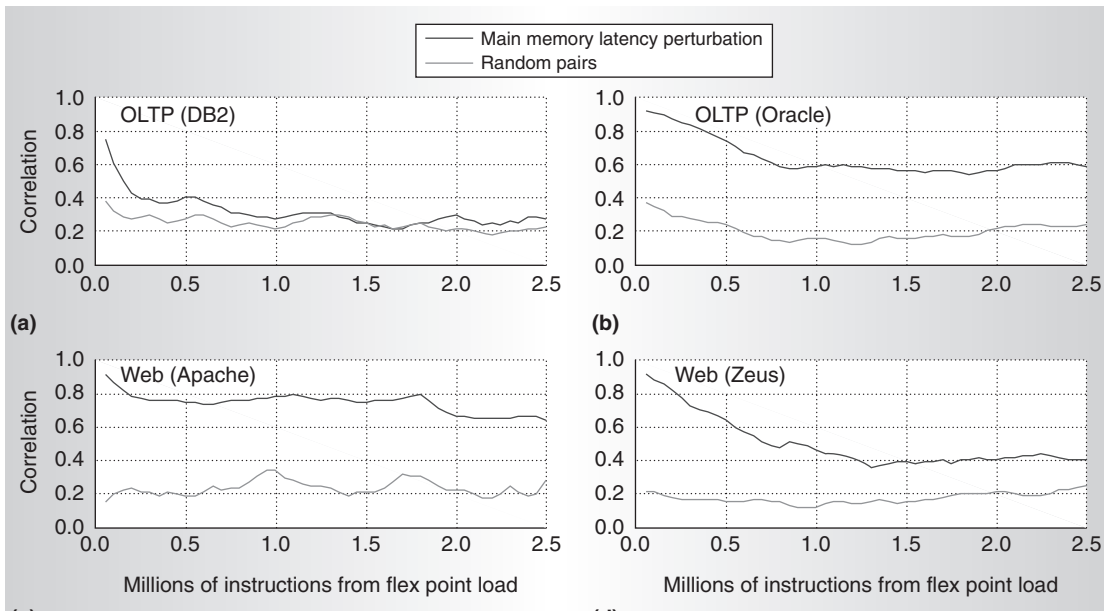
Figure 9. Code signature correlation from random main-memory latency perturbation for OLTP (DB2) (a), OLTP (Oracle) (b), Web (Apache) (c), and Web (Zeus) (d). To avoid biasing a sample, the correlation across execution paths reached through random memory latency perturbation must match that of randomly chosen execution path pairs. Even after the execution of millions of instructions, there is still a strong correlation across perturbed simulations launched from the same flex point.

to execution paths that diverge rapidly enough to be cost-effective for simulation sampling.

The challenge facing this approach is to determine how to construct an unbiased uniform sample of transaction interleavings from a single starting state. For example, if we were to start measurement a mere 10 cycles after a flex-point load, perturbing main memory latency would have no effect on execution path, and confidence calculations over such a "sample" would be meaningless. At the other extreme, if we introduced random perturbations for an extremely long simulation interval (for example, 30 seconds of simulated time) before measuring, the transaction interleavings we measured would likely be quite random with respect to the initial state, and therefore, form a uniform sample.

As these examples show, the key issue in producing multiple measurements from a single flex point is the question of how long random perturbation must be performed before the two measurements are independent—that is, before they differ from one another as much as randomly chosen start locations. If two measurements are not independent, they introduce bias into the sample results.

We investigated this question empirically by measuring the similarity, or correlation, between the execution paths a CPU follows in two simulations starting from the same flex point with random main-memory latency perturbations. We compared this correlation to that observed between randomly chosen execution path pairs. A stronger correlation between the two simulations starting from a single flex point would imply that the measurements are not independent and the sample is biased. (The converse is not true—measurement independence is a necessary but not sufficient condition for a uniform sample.) Over time, we expect the two simulations' execution paths to drift apart with main-memory latency perturbation, and the correlation to approach that of random pairs.

To make our experiment independent of microarchitecture and timing, we characterized the execution path of each CPU using a sequence of basic-block vectors.[12] Basic-block vectors summarize the relative frequency of static basic blocks within an execution window, and are thus affected by timing perturbations only if the execution path changes.

Figure 9 shows our correlation measurements out to 2.5 million instructions from

flex-point load, more than 15 times further than a typical SimFlex detailed simulation. For most workloads, even 2.5 million instructions' execution introduces insufficient randomness in the execution path to achieve independent measurements. For OLTP on DB2, 500,000 to 1 million instructions of execution after each flex point might provide an unbiased sample.

These results show that main-memory latency perturbation might be used to achieve the same sample size with fewer flex points, but would require millions of instructions of detailed warming with main-memory latency perturbation. Creating a sample in this fashion requires an order of magnitude increase in detailed warming over the SimFlex approach, and thus a corresponding increase in simulation turnaround time.

The large speedup and parallelism enabled through statistical sampling and checkpointing have important implications for the design of future computer system simulators. First, the large speedups from sampling let simulator authors focus on designing flexible, modular simulators rather than optimizing for speed at all costs. Second, although hardware prototypes have many other advantages, it is not clear that they can reduce experiment turnaround time relative to sampled simulations. Finally, the 100- to 1,000-way parallelism available across measurements mitigates the need to multithread detailed simulation models. MICRO

Acknowledgments

References

1. R.E. Wunderlich et al., "Statistical Sampling of Microarchitecture Simulation," *ACM Trans. Modeling and Computer Simulation*, vol. 16, no. 3, July 2006, pp. 197-224.
2. J.J. Yi et al., "Characterizing and Comparing Prevailing Simulation Methodologies," *Proc. Int'l Symp. High-Performance Computer Architecture* (HPCA 05), IEEE CS Press, 2005, pp. 266-277.
3. T.F. Wenisch et al., "Simulation Sampling with Live-Points," *Proc. Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 06), IEEE Press, 2006, pp. 2-12.
4. P.S. Magnusson et al., "Simics: A Full-System Simulation Platform," *Computer*, vol. 35, no. 2, Feb. 2002, pp. 50-58.
5. A.R. Alameldeen and D.A. Wood, "Variability in Architectural Simulations of Multithreaded Workloads," *Proc. 9th Int'l Symp. High-Performance Computer Architecture* (HPCA 03), IEEE Press, 2003, pp. 7-18.
6. R. Hankins et al., "Scaling and Characterizing Database Workloads: Bridging the Gap between Research and Practice," *Proc. 36th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 36), IEEE CS Press, 2003, pp. 151-163.
7. N. Hardavellas et al., "SimFlex: A Fast, Accurate, Flexible Full-System Simulation Framework for Performance Evaluation of Server Architecture," *Sigmetrics Performance Evaluation Rev.*, vol. 31, no. 4, Apr. 2004, pp. 31-35.
8. J.M. Hellerstein, P.J. Haas, and H.J. Wang, "Online Aggregation," *Proc. Int'l Conf. Management of Data*, ACM Press, 1997, pp. 171-182.
9. M. Ekman and P. Stenstrom, "Enhancing Multiprocessor Architecture Simulation Speed Using Matched-Pair Comparison," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 05), IEEE CS Press, 2005, pp. 89-99.
10. S. Somogyi et al., "Spatial Memory Streaming," *Proc. Int'l Symp. Computer Architecture* (ISCA 06), IEEE CS Press, 2006, pp. 252-263.
11. R.E. Wunderlich et al., "An Evaluation of Stratified Sampling of Microarchitecture Simulations," *3rd Ann. Workshop Duplicating, Deconstructing, and Debunking* (WDDD 04), 2004, pp. 13-18; http://www.ece.wisc.edu/~wddd/2004/WDDD2004_proceedings.pdf.
12. G. Hamerly et al., "Simpoint 3.0: Faster and More Flexible Program Analysis," *J. Instruction-Level Parallelism*, vol. 7, Sept. 2005; http://www.jilp.org/vol7.

Thomas F. Wenisch is a PhD candidate in electrical and computer engineering at Carnegie Mellon University. His research interests include multiprocessor computer architecture, memory system design, and computer system performance evaluation methodology. Wenisch has an MS from Carnegie Mellon and a BS from the University of Rhode Island, both in computer engineering. He is a student member of the IEEE and ACM.

Roland E. Wunderlich is a PhD candidate in electrical and computer engineering at Carnegie Mellon University. His research interests focus on the application of statistical modeling techniques to computer architecture and automated software tuning. Wunderlich has a BS in computer engineering from Rutgers University and an MS from Carnegie Mellon. He is a student member of the IEEE and ACM.

Michael Ferdman is a PhD candidate in electrical and computer engineering at Carnegie Mellon University. His research interests include computer architecture with an emphasis on proactive memory system design. Ferdman has a BS in computer science, and a BS and an MS in electrical and computer engineering from Carnegie Mellon. He is a student member of the ACM.

Anastassia Ailamaki is an assistant professor of computer science and a Sloan Research Fellow at Carnegie Mellon University. Her research interests focus on database systems and applications, with an emphasis on database system behavior on modern processor hardware and disks. Ailamaki has a BSc in computer engineering from the Polytechnic School of the University of Patra, Greece; an MSc in computer engineering from the Technical University of Crete, Greece; an MSc in computer science from the University of Rochester, New York; and a PhD in computer science from the University of Wisconsin-Madison. She is a member of the IEEE and the ACM.
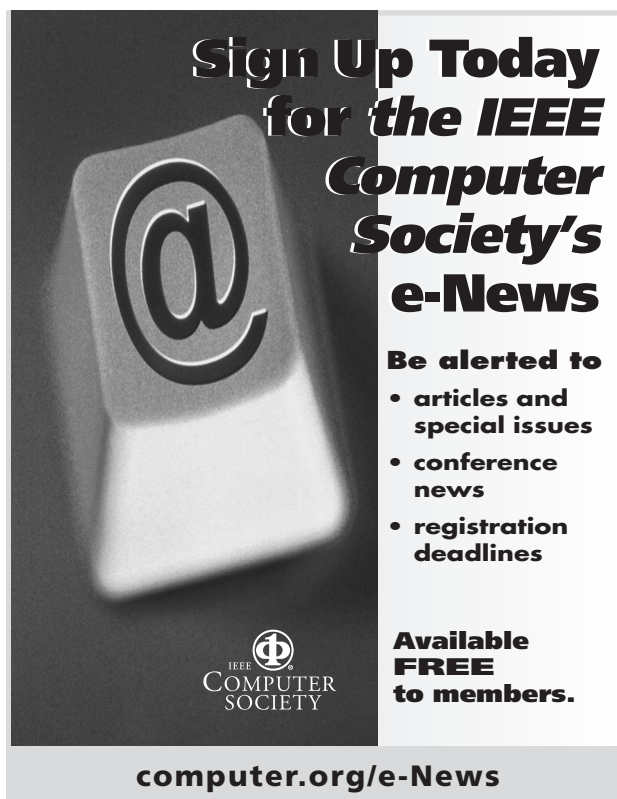
Babak Falsafi is an associate professor of electrical and computer engineering and a Sloan Research Fellow at Carnegie Mellon University. His research interests include computer architecture with an emphasis on high-performance memory systems, architectural support for gigascale integration, and computer system performance evaluation tools. Falsafi has a PhD in computer science from the University of Wisconsin. He is a member of the IEEE and the ACM.

James C. Hoe is an associate professor of electrical and computer engineering at Carnegie Mellon University. His research interests include computer architecture and high-level hardware description and synthesis. Hoe has a PhD in electrical engineering and computer science from MIT. He is a member of the IEEE and ACM.

Direct questions and comments about this article to Thomas Wenisch, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, 5000 Forbes Ave. Pittsburgh, PA 15213; twenisch@ece.cmu.edu.

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