EFFICIENT SAMPLING STARTUP FOR SIMPOINT

SAMPLING TECHNIQUES DRAMATICALLY SHORTEN SIMULATION TIMES FOR INDUSTRY-STANDARD BENCHMARKS, BUT ESTABLISHING THE CORRECT ARCHITECTURE AND MICROARCHITECTURE STATES AT THE BEGINNING OF EACH SAMPLE CAN BE TIME-CONSUMING. THIS ARTICLE COMPARES THE ACCURACY AND SPEED OF VARIOUS SAMPLING STARTUP TECHNIQUES, INTRODUCING *TOUCHED MEMORY IMAGE* AND *MEMORY HIERARCHY STATE*. TOGETHER, THESE TWO TECHNIQUES REDUCE SAMPLED BENCHMARK SIMULATION TIMES FROM HOURS TO MINUTES.

Michael Van Biesbrouck
Brad Calder
University of California, San Diego

Lieven Eeckhout
Ghent University, Belgium

Modern computer architecture research relies heavily on cycle-accurate simulation to evaluate new architectural features, but for industry-standard benchmarks, full simulation can take weeks to months. Thus, to measure cycle-level events and examine the effect that hardware optimizations would have on a whole program, architects perform *sampled simulation*. They execute only small portions of the program at cycle-level detail and then use that information to approximate the full program behavior. The subset chosen for detailed study has a profound impact on the accuracy of the performance approximation, and picking the samples to be as representative as possible of the full program is a topic of several research studies.[1–3] The SimPoint tool,[2] developed at the University of California, San Diego, uses a phase classification algorithm to choose representative simulation samples. (For a brief summary, see the sidebar "How SimPoint works.")

Although sampling techniques such as SimPoint offer substantial time savings, their bottleneck has been *sampling startup,* the process of reconstructing the state that would be created by simulating the full benchmark up to the point when the sample starts. (We use the noun "sample" to mean a sampling unit, and the verb "sample" to mean collecting a sample unit.) Sampling startup has two elements:

- the *sample starting image*—determining the correct program memory contents (architecture state) at the beginning of the sample; and
- the *sample warm-up*—preparing the processor's internal data structures (the microarchitecture state) for the sample.

This article proposes efficient and accurate sampling startup approaches. To guarantee a correct architecture state (the program's memory contents), we present the *touched memory image* (TMI), which stores only the words of memory that are accessed in a sample. Our TMI files are two orders of magnitude smaller than normal checkpoints. Because they are small, they also load instantaneously and are

Published by the IEEE Computer Society

significantly faster than using either fast-forwarding or full checkpoints.

To guarantee a warm microarchitecture state, we propose the *memory hierarchy state* (MHS), which we can use to faithfully recreate the state of the major microarchitecture components, such as caches and translation look-aside buffers (TLBs), at the start of a sample. The MHS is a microarchitecture checkpoint of a cache that is at least as large as any to be examined during a design space exploration study; the way we store the microarchitecture checkpoint allows for the recreation of smaller cache sizes and associativities.

By combining TMI with MHS, we can accurately and efficiently collect samples of simulated processor execution. The end result is a sampled simulation method that is accurate, efficient in terms of disk storage, and fast enough for simulating industry-standard benchmarks in minutes. In this article, we focus on the applicability of our sampling startup techniques to SimPoint,[2] but they apply identically to statistical sampling and stratified sampling, as we have discussed in a previous publication.[4]

Sample starting image

The sample starting image (SSI) is the state of the architecture (the programmer-visible registers and memory) needed to enable the correct functional simulation of a sample. Computer architects usually obtain the SSI by fast-forwarding from the start of execution or by loading a checkpoint. Let's look at how these conventional approaches work before we introduce the TMI, which eliminates their drawbacks.

Fast-forwarding

Fast-forwarding quickly emulates the program's execution from the start of execution to reach the sample of execution to be simulated. The advantage of this approach is that it is simple to implement in simulators. The disadvantage is that it serializes the simulation of all of the samples in a program, and so can require fast-forwarding through the same instructions many times. In addition, most fast-forwarding implementations in current simulators are fairly slow. Proposals for accelerating fast-forwarding—through native execution, JIT compilation, and binary modification techniques[5,6]—are complex and difficult to port across simulated instruction set architectures (ISAs) and host platforms, yet they still don't completely remove fast-forwarding time. The technique that we propose is not dependent upon the host platform and requires the change of no more than a few lines of simulator code.

Full checkpoint

A checkpoint, which consists of program register state and an image of the program memory, is similar to a core dump of the program. Using checkpoints avoids time-consuming fast-forwarding and enables efficient parallel simulation. The disadvantage is that full checkpoints can be very large. Using many samples could be prohibitively costly in terms of disk space. In addition, loading the large checkpoint file from disk (or transferring it over a network) at the beginning of a sample adds to the total simulation time.

Touched memory image

To reduce the size of the checkpoint file, we use the TMI, which stores only the blocks of memory to be accessed during sample simulation. TurboSMARTSim, created in parallel and independently of this work, uses a similar

How SimPoint works

The SimPoint sampling approach picks a few samples that, when simulated, accurately create a representation of the program's complete execution. To do this, SimPoint breaks a program's execution into intervals, and for each interval creates a code signature, which is a profile of the basic blocks executed during the interval. Next, it performs clustering on the code signatures, grouping intervals with similar code signatures into phases. The notion is that intervals of execution with similar code signatures have similar architecture behavior, which research has confirmed.[1,2] Therefore, we need simulate only one interval from each phase to recreate a complete picture of the program's execution. SimPoint then chooses a representative from each phase and performs detailed simulation on that interval. The chosen samples, each an interval on the order of millions of instructions, are called *simulation points*. Taken together, these simulation points can represent the complete execution of a program.

References

1. T. Sherwood et al., "Automatically Characterizing Large Scale Program Behavior," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, 2002, pp. 45-57.
2. J.J. Yi et al., "Characterizing and Comparing Prevailing Simulation Techniques," *Proc. 11th Int'l Symp. High-Performance Computer Architecture* (HPCA 05), IEEE CS Press, 2005, pp. 266-277.

Automatically logging operating-system effects

The checkpointing techniques in this article focus on providing the starting image and the warmed microarchitecture state to accurately start simulation. Similar checkpointing techniques can also capture system interactions to provide application simulation without having to provide any emulation support for the system calls in the simulator.

Narayanasamy et al. present an approach that creates a *system effect log* to automatically capture all system effects in a simulation sample.[1] This approach automatically determines when a system effect has modified an application's memory location. To create the system effect log, the approach uses a tool such as Pin[2] to profile the sample of execution. During this profiling, the application keeps an up-to-date shadow copy of all of the memory values the application code has read and written. For each load the application executes, if the value in the real memory differs from the shadow copy, some external event—such as a system call, an interrupt, or a DMA transfer—has modified the memory value. The approach automatically identifies when this has occurred for a load, and then logs the execution instance of the load along with the value. During simulation of a sample, when the application executes that load instance, the simulator consumes the load's value from the system effect log and then stores that value in the simulator's memory. This allows the reproducible simulation of application-level behavior without system call emulation support in the simulator. For each system call executed, the system effect log also contains any changes to the register state affected by the system call. The instrumentation added to the application automatically discovers these changes by examining the differences between the register states before and after the system call. It loads these logged register values into registers when the system call is simulated.

This approach has several benefits:

- The log can be used during simulation to deterministically reexecute the application across system calls and interrupts, providing reproducible simulation results.
- The simulation environment does not have to support and maintain system call emulation, and it makes it significantly easier to simulate other operating systems as well as port the simulation environment to other operating systems.
- The approach easily allows simulation of samples of real-world applications on today's popular architecture simulators (such as SimpleScalar).

Because it identifies the registers and memory locations modified by system calls completely independently of the semantics of the system call, this approach is easy to implement and is portable across operating systems. Computer architects use this approach to simulate Linux, Mac OS X, and Windows applications, with Pin generating the system effect logs, which are then consumed by application-level simulators.

References

1. S. Narayanasamy et al., "Automatic Logging of Operating System Effects to Guide Application Level Architecture Simulation," *Proc. Joint Int'l Conf. Measurement and Modeling of Computer Systems* (Sigmetrics 06), ACM Press, 2006, pp. 216-227.
2. C.K. Luk et al., "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Proc. Conf. on Programming Language Design and Implementation* (PLDI 05), ACM Press, 2005, pp. 190-200.

technique.[7] The TMI is a collection of chunks of memory (touched during the sample) with their corresponding memory addresses and data. These are the blocks read during the simulation sample. For each sample, we store a TMI on disk. At simulation time, before simulating the given sample, we load the TMI from disk. The chunks of memory in the TMI are then written to their corresponding memory addresses. This guarantees a correct SSI when starting the sample simulation.

Several optimizations reduce the TMI's required storage. We use a sparse image representation so that the TMI does not store regions of memory that consist of consecutive 0s. In addition, the TMI combines large regions of non-0 sections of memory and stores them as a single chunk. This saves storage space in terms of memory addresses in the TMI, because it needs to store only one memory address for a large consecutive data region. Finally, we store only memory regions that are read during the sample; the TMI does not need to store regions that are written before being read.

The TMI contains only the data needed to execute the correct path of execution. Therefore, there may be some memory locations accessed during speculative execution while simulating the sample that will not be in the sample's memory image. We compared the performance between having the full memory image versus TMI and found that not having the exact memory data used by some of these wrong-path effects resulted on average in only a 1 percent difference in estimated performance.[4] An alternative to the TMI is the load value sequence (LVS), a sequence of load values read in the sample.[4]

Remainder of the sample startup image

The rest of the SSI includes the code used for the simulation sample and a trace of all of the system call interactions to guide reproducible architecture simulations. We could either represent the code in the checkpoint as the original binary along with the dynamically loaded libraries and where these libraries were loaded, or as a set of code pages used during execution. We can represent the system call interactions in a checkpoint by a trace of all inputs and outputs of all system calls executed for the sample to be simulated, as done in SimpleScalar external I/O (EIO) files. The "Automatically logging operating-system effects" sidebar describes an improved approach for providing this trace for an SSI.

Sample warm-up

Sample warm-up techniques prepare the microarchitecture state for a sample. Memory hierarchy structures such as caches and TLBs represent a very large fraction of the state in a microprocessor. Structures such as branch predictors and processor core structures (reorder buffer, issue buffers, and so on) have significantly less state. Because our main focus in this article is the SimPoint sampling methodology, which uses relatively large samples (on the order of millions of instructions), we concentrate here on warming the memory hierarchy state.

Memory hierarchy warm-up techniques fall into three main categories:

- estimating the cache miss rate in the sample,
- simulating additional instructions before the sample, and
- taking a microarchitecture checkpoint.

To cover all three types of warm-up, we examine five warm-up strategies: no warm-up, hit on cold, fixed-length warm-up, memory reference reuse latency, and our memory hierarchy state approach.

No warm-up

The no-warm-up strategy assumes an empty cache at the beginning of each sample—that is, that the first use of each cache block in the sample will be a miss. Obviously, this will result in an overestimation of the number of cache misses, and consequently an underestimation of overall performance. However, for large sample sizes, the bias can be small. This strategy is very simple to implement and incurs no runtime overhead.

Hit on cold

The hit-on-cold strategy also uses an empty cache at the beginning of each sample, but assumes that the first use of each cache block in the sample is always a hit. Hit on cold works well for programs that have a high hit rate, but it requires a modification to the simulator to check a bit on every cache miss. If the bit indicates that the cache block has yet to be used, the address tag is added to the cache but the access is considered to be a hit.

Fixed-length warm-up

Many warm-up approaches simulate additional instructions prior to the sample to warm large hardware structures.[1,7–9] The simplest warm-up technique merely provides a fixed-length warm-up before each sample. For example, caches and branch predictors might undergo a warm-up of one million instructions of execution before simulation of each sample.

Memory reference reuse latency

The memory reference reuse latency (MRRL) approach, proposed by Haskins and Skadron, makes a microarchitecture-independent analysis of memory references.[8] We can use this to determine how far back in execution we need to go so that warming will encounter a prior access to 99.9 percent of the memory locations that we will access during simulation of a particular sample. This point will be the start of the warm-up period for simulation of that sample. We fast-forward to or load the checkpoint for the warm-up starting point.

From that point until the sample's starting point, we run functional simulation in conjunction with cache and branch-predictor warm-up, also called functional warming. That is, all memory references warm the caches, and all branch addresses warm the branch predictors. When the simulation reaches the beginning of a sample, detailed processor simulation begins to obtain performance results.

The cost of the MRRL approach is that it requires simulation of many instructions during warm-up.

Memory hierarchy state

Our final warm-up strategy, based on the memory hierarchy state (MHS), stores cache state so that caches do not need warming at the start of simulation. We can apply the same technique to other cache-like structures, such as TLBs. By this approach, we collect the MHS through cache simulation—that is, functional simulation of the memory hierarchy. Design-space exploration can require simulation of many different cache configurations. We collect the MHS only once for each block size and replacement policy, and then reuse it extensively during design space exploration of

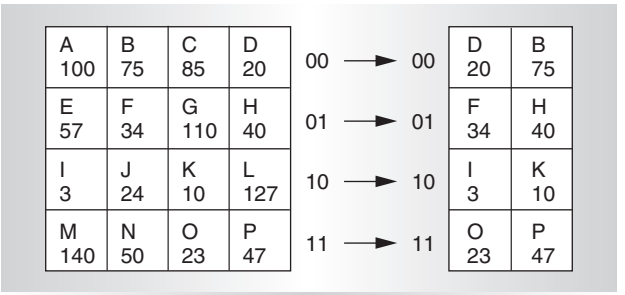

Figure 1. Using the memory hierarchy state strategy to reduce a cache from four-way to two-way associativity.

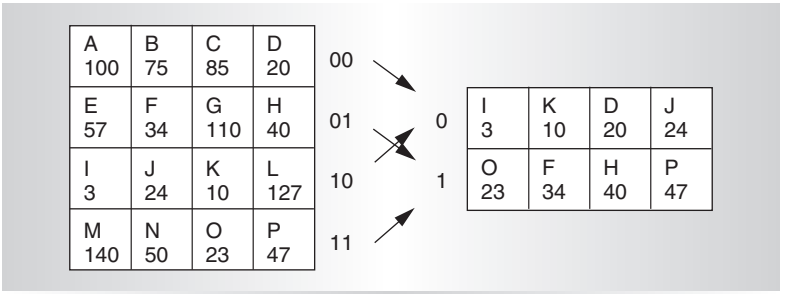

Figure 2. Using MHS to reduce the number of cache sets.

memory hierarchies, any of which may have a smaller size or a smaller associativity.

Our technique is similar to trace-based construction of caches, except that storing information in a cache-like structure decreases both storage space and the time to create the cache required for simulation. Along with the cache tags, we store status information for each cache line so that dirty cache lines are correctly marked.

Figures 1 and 2 demonstrate how MHS works when reducing the cache associativity and the number of sets. In these figures, each row is a cache set with a number of columns equal to the cache's associativity. Each cache block is labeled with a letter representing a tag and a number representing the time (in memory operations) since the cache block was last used. In the top row of the large cache (cache set 00), D and B are the most recently used blocks. When MHS reduces associativity to two (Figure 1), those are the two most recently used blocks, so MHS retains them. Figure 2 shows the reduction of the number of cache lines to two, merging cache sets 00 and 10 to a single set (0), and cache sets 01 and 11 into set 1. The new cache sets contain the most recently used entries from both of the cache sets that merged. This operation increases the length of the cache tags by one bit, indicating from which cache sets the cache lines originated.

MHS and MRRL are equally microarchitecture-independent. MHS stores all addresses needed to create the largest and most associative cache size of interest. Similarly, MRRL goes back in execution history far enough to capture the working set for the largest cache of interest. The techniques have different trade-offs, however. MHS requires more disk space than MRRL; MRRL needs only to store the point at which to begin warming, whereas MHS stores a source cache. In terms of simulation speed, MHS substantially outperforms MRRL because MHS does not need to simulate instructions to warm the cache. Loading the MHS trace takes very little time.

For many microarchitectural configurations, caches and TLBs are the only important structures to simulate and store to disk for use during simulation. Most other microarchitectural structures are small enough that they warm up in the first few thousand instructions of execution, causing negligible error in the context of 1-million-instruction samples.

We can achieve low error rates by storing only caches to disk, but for completeness, let's look at appropriate mechanisms for storing other large microarchitectural structures as well. If a microarchitectural feature remains constant over all experiments, it is usually sufficient to simulate its behavior once and store it to disk, and then use this to restore its state for all experiments. Only microarchitectural features whose configurations change during the experiments need special treatment. Furthermore, as long as stored microarchitectural structures are independent in their contents, a single run of a functional simulator can save all possible versions of the various structures at the same time that the simulator creates the MHS. MRRL can simulate any combination of microarchitectural features, but it executes the same presample instructions once for every microarchitectural configuration.

It might be necessary to store large, complex conditional branch predictors to disk for each of their possible configurations. If the components of a branch predictor can be considered separately, it might be possible to look at many configurations, with only a few variations saved to disk. For example, branch target buffers and return-address stacks can be resized, just like

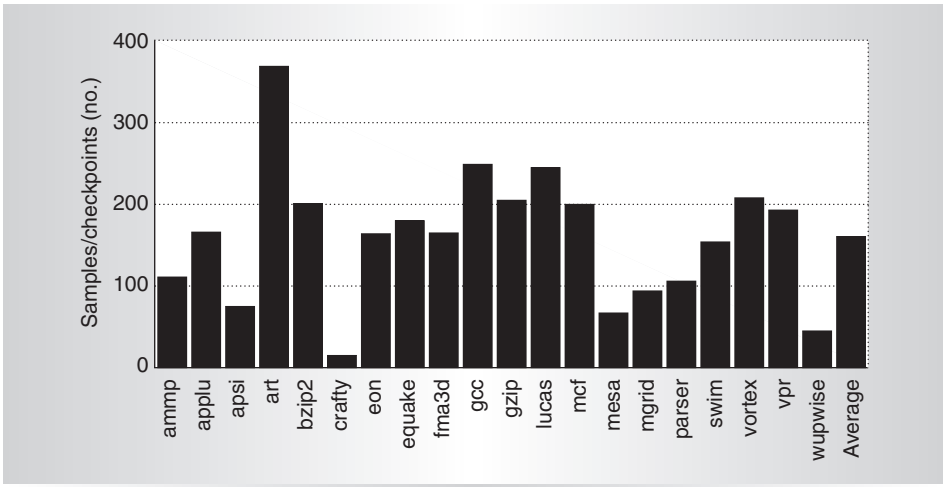

Figure 3. Number of simulation point samples used per benchmark, with the maximum number of phases set to 400.

caches, so only one large instance of each must be stored. Other conditional branch predictor components might need to be simulated for each size and algorithm used, and these can usually be combined with branch target buffers and address stacks of any size.

Situations in which separate microarchitectural components affect each other are more complex to handle. For example, consider an experiment that examines various cache sizes and several different prefetchers with internal state. The prefetchers affect the cache's contents, so an ideal simulation would simulate the two structures together, storing correlated cache and prefetcher contents to disk. In the worst case, every combination of two components that interfere with each other would need to be stored to disk for warm-up. In this case, our technique to resize the caches is not affected by the existence of prefetched data from a single prefetcher. Thus, we can store one copy of the MHS for each prefetcher design even though we examine many cache configurations. Any microarchitectural structure that does not interfere with prefetching and cache contents can have its configurations simulated separately, minimizing the number of microarchitectural combinations that must be simulated.

Evaluation

To evaluate our sampling startup techniques—TMI and MHS—we first performed experiments to compare their accuracy with the other startup techniques. Next, we evaluated the applicability of our sampling startup techniques for targeted sampling as done in SimPoint.

Methodology

Our experiments used the MRRL-modified SimpleScalar simulator, which supports taking multiple samples, interleaved with fast-forwarding and functional warming (http://www.cs.virginia.edu/~jwh6q/mrrl-web). With minor modifications, this simulator also supports checkpoints, TMI, hit on cold, and MHS. We simulated SPEC 2000 benchmarks compiled for the Alpha ISA and using reference inputs.

The simulation we performed was of an aggressive, eight-wide, superscalar, out-of-order processor. The branch predictor combines a bimodal predictor with a two-level predictor, using a 8,192-entry table of 2-bit counters in each case. Using a conservative branch predictor with an aggressive execution engine emphasizes the negative side effects of TMI by increasing the number of misspeculated instructions that execute. The processor's memory hierarchy consists of an 8-Kbyte L1 instruction cache and a 16-Kbyte L1 data cache along with a unified 1-Mbyte, four-way, set-associative, unified L2 cache.

To study the applicability of the reduced checkpointing and warm-up techniques for targeted sampling using SimPoint, we used SimPoint with an interval size of 1 million, with the maximum number of phases set to 400. Figure 3 shows the number of 1-million-

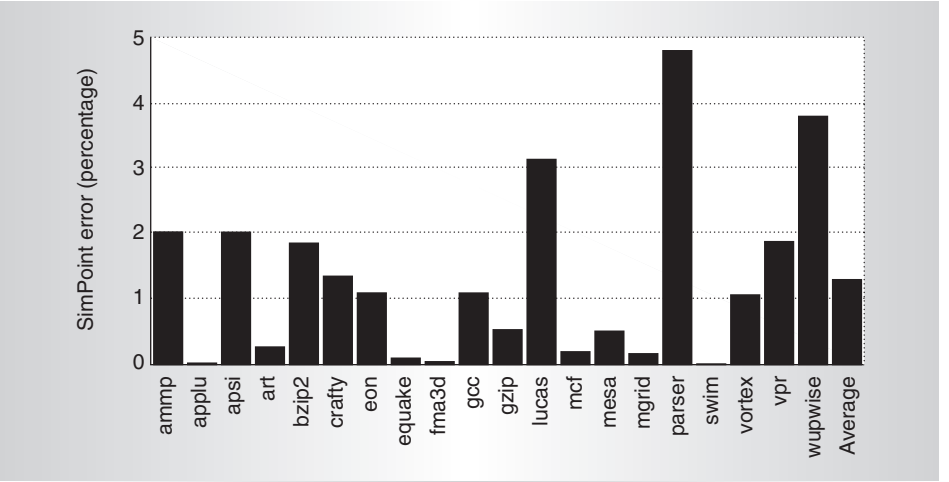

Figure 4. SimPoint's accuracy with perfect sampling, compared to the complete execution of the program's input.

instruction simulation points (samples) per benchmark. This is also the number of checkpoints per benchmark, because we need one checkpoint per simulation point. The number of checkpoints per benchmark varies from 15 (for crafty) to 369 (for art). We focus on small, 1-million-instruction intervals because SimPoint is most accurate when many (at least 50 to 100) small intervals (1 million instructions or less) are accurately simulated. However, we found that using the TMI checkpoints offers an important savings in disk space even for 10-million and 100-million instruction interval sizes.

Figure 4 shows SimPoint's accuracy with perfect sampling startup compared to the complete execution of the program; perfect sampling startup assumes the state prior to the sample to be the same as under complete benchmark execution. The average error is 1.3 percent; the maximum error is 4.8 percent (for the parser benchmark). We use this configuration as the baseline for our experiments, which concentrate on reducing error, disk usage, and simulation time.

Error analysis

Figure 5 evaluates error rate in cycles per instruction (CPI) for various sample warm-up techniques as compared to the SimPoint approach using perfect warm-up—this excludes any error introduced by SimPoint. To calculate this error rate, we used a baseline for which we took CPI samples from a complete detailed execution of the entire benchmark. The no-warm-up and hit-on-cold strategies result in high error rates—17 percent and 25 percent on average. For many benchmarks, one of these two strategies is dramatically better than the other, suggesting that an algorithm that intelligently chooses between them might offer a significantly lower error rate.

The fixed 1-million-instruction warm-up achieves better accuracy, with an average error of 4 percent. However, its maximum error can be fairly large—16 percent, for the parser benchmark. MRRL and MHS obtained significantly better error rates, with the average error for each approaching 1 percent. Thus, we conclude that MRRL and MHS are equally accurate when used in conjunction with SimPoint.

So far, however, we've only discussed error rates assuming full checkpoints. Using TMI in conjunction with MHS increases the error rates only slightly, from 1 to 1.2 percent. This increase stems from the fact that TMI does not always include load values for loads being executed along mispredicted paths. TMI includes these load values in the memory only if they are read by correct-path instructions during the sample or previously written during the sample. We found that, on average, only about 2 percent of the issued loads read incorrect data and then used it in a way that affected wrong-path execution. This is what led to the average difference in CPI error rates of 0.2 percent; the maximum difference in CPI error we saw was 1 percent. (We show detailed results elsewhere.[4])

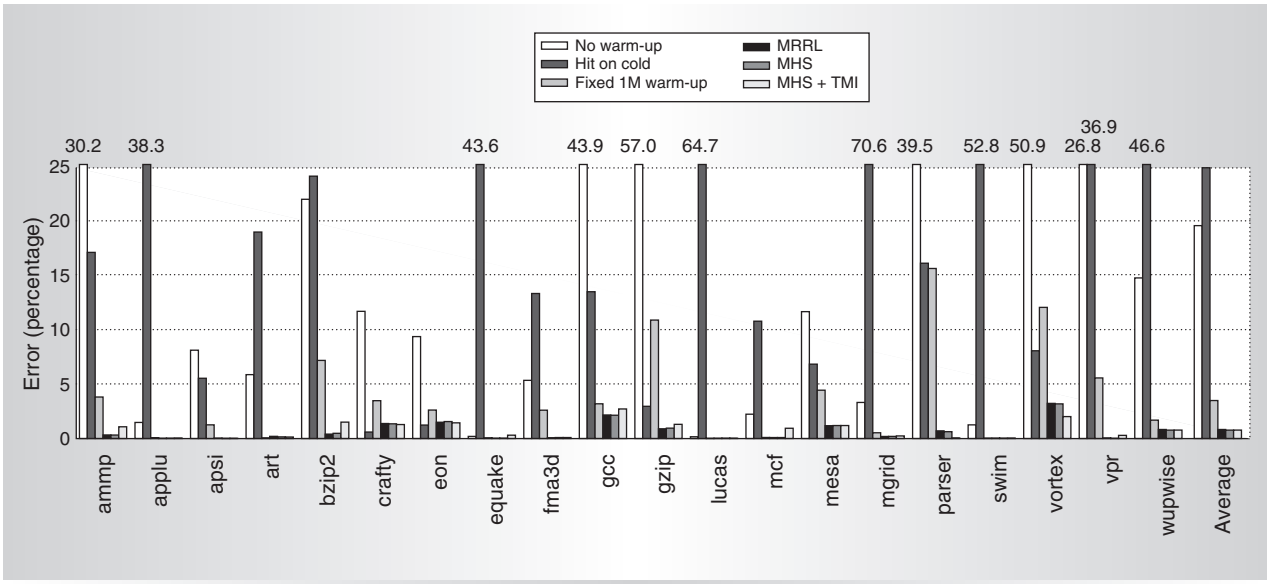

Figure 5. Percentage error in estimating overall CPI for SimPoint using various warm-up techniques, as compared to SimPoint with no sampling error (perfect warm-up). Labels indicate error rates exceeding 25 percent.

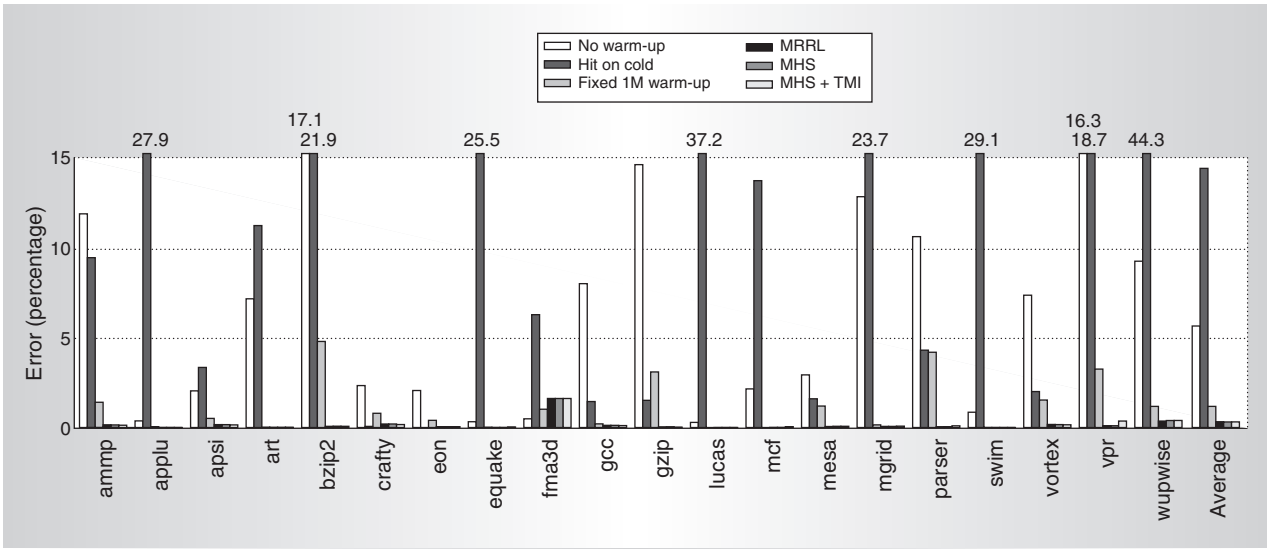

Figure 6. Percentage error in estimating overall L2 cache hit rates for SimPoint using various warm-up techniques, as compared to hit rates estimated by SimPoint with no sampling error (perfect warm-up). Labels indicate error rates exceeding 15 percent.

We also found that all techniques accurately predicted the branch predictor and L1 cache hit rates. The branch prediction error is about 0.35 percent for all techniques, even though only MRRL warms the branch predictor. Since the L1 caches are small, they warm quickly. For the data cache, every technique other than hit on cold (which shows a 0.15 percent average error) has less than 0.1 percent average error. The instruction cache error rates are lower with MRRL or MHS. But with the average error below 0.02 percent for all the techniques, this improvement might not be significant. Only the L2 cache is large enough to require warming for 1-million-instruction samples.

The unified L2 cache is much larger than the L1 caches, so warm-up issues can be significant. Figure 6 clearly shows the advantages of cache warm-up. MHS and MRRL have

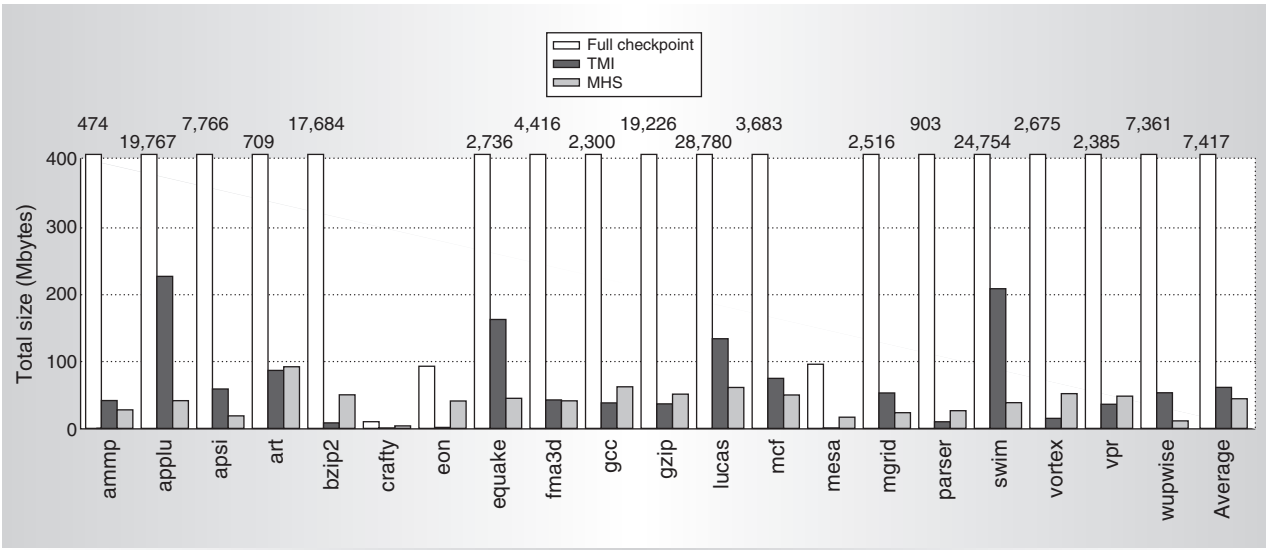

Figure 7. Total storage requirements per benchmark for full checkpointing, TMI, and sample warm-up through MHS.

average error rates below 0.18 percent; the only benchmark with error rates higher than average is fma3d, with error rates of 1.6 percent. Fixed warm-up gives an average error rate of 1.2 percent, but for parser and bzip2 the error rate is nearly 5 percent. For some benchmarks, hit on cold is better than no warm-up, but overall their error rates are 14 percent and 6.5 percent, respectively.

More significant than the error rates is the precise correlation between L2 cache rate error (Figure 6) and CPI error (Figure 5). The relative magnitudes of the bars match nearly exactly for every benchmark. Clearly, accurate performance prediction is sensitive to the warm-up of the L2 cache.

Storage requirements

Figure 7 shows the total sizes (in Mbytes) of the files that need to be stored on disk per benchmark for three sample startup approaches: full checkpoint, TMI, and MHS. The file sizes for the full-checkpoint approach are huge. The average file size per compressed checkpoint is 49.3 Mbytes, and the average total file size per benchmark is 7.4 Gbytes. Storing all full checkpoints for a complete benchmark can, however, take up to 28.8 Gbytes (lucas). The maximum average storage requirement per checkpoint can be large as well; for wupwise, it is 163.6 Mbytes. Loading and transferring these large checkpoints over a network can be costly in terms of simulation time.

TMI reduces checkpoint size by more than two orders of magnitude. The average total TMI checkpoint file size per benchmark is 52.6 Mbytes, and the maximum total file size is 206 Mbytes, for applu. These huge checkpoint file size reductions make checkpointing feasible in terms of storage cost for sampled simulation. Also, the typical single checkpoint size is significantly reduced to 365 Kbytes, which makes loading the checkpoints highly efficient.

MHS is the only warm-up approach we discussed that requires additional storage. Figure 7 quantifies the storage this strategy needs to store cache contents. The total average storage MHS requires per benchmark is 40 Mbytes, with an average of 256 Kbytes per checkpoint (8 bytes per cache block).

Of course, MHS requires this additional storage on top of the storage needed for the checkpoints. The small file sizes, however, allow efficient loading.

Total simulation time

Figure 8 shows the total simulation time (in minutes) for the various sample startup techniques when simulating all simulation points on a single machine. These simulation times include the time required for fast-forwarding, loading checkpoints or TMI, loading MHS, or warming structures by functional warming or detailed execution.

We considered the SSI techniques of fast-forwarding, checkpointing, and reduced check-

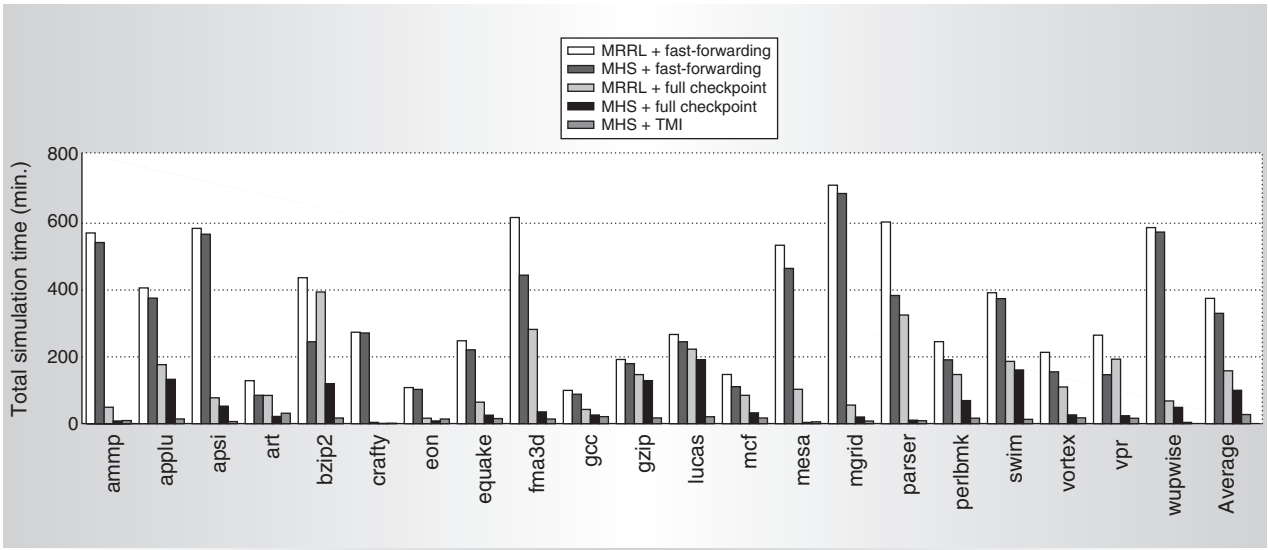

Figure 8. Total time to simulate all samples for SimPoint using various sample startup techniques. Total time includes time for fast-forwarding, loading checkpoints, and warming.

pointing using TMI in combination with the two most accurate sample-warm-up techniques, MRRL and MHS. MRRL and MHS with fast-forwarding are both slow. The average total simulation time is more than 6 hours per benchmark with MRRL, and more than 5 hours with MHS. If we combine MRRL with checkpointing, the average total simulation time per benchmark drops below 2.25 hours. Combining MHS with full checkpointing decreases total simulation time even further, to 55 minutes. Combining the reduced-checkpoint TMI approach with MHS reduces the average total simulation time per benchmark to less than 14 minutes on a single processor. Because most benchmarks are represented by 100 to 200 samples that can be simulated concurrently, parallel simulation of a single benchmark can reduce that time to seconds.

Using our two sampling startup techniques—TMI for the sample starting image and MHS to warm the microarchitecture—significantly improves the efficiency of sampled simulation. TMI requires two orders of magnitude less storage than full checkpointing and results in faster simulation than both fast-forwarding and full checkpointing. Our microarchitecture checkpointing method, MHS, is as accurate as MRRL and substantially faster. The end result for sampled simulation is that obtaining highly accurate per-benchmark performance estimates (only a few percent CPI prediction error) takes only minutes, whereas previously proposed techniques required multiple hours. In the future, we intend to extend the MHS technique to support shared caches in multithreaded processors and improve the MHS support for branch prediction. MICRO

Acknowledgments

We thank the anonymous reviewers for their helpful comments on this article. Our work was funded in part by NSF grant CCR-0311710, NSF grant CCF-0342522, UC MICRO grant 05-115, the European SARC project 27648, the HiPEAC Network of Excellence and a grant from Intel and Microsoft.

References

1. T.M. Conte, M.A. Hirsch, and K.N. Menezes, "Reducing State Loss for Effective Trace Sampling of Superscalar Processors," *Proc. Int'l Conf. Computer Design* (ICCD 96), IEEE CS Press, 1996, pp. 468-477.
2. T. Sherwood et al., "Automatically Characterizing Large Scale Program Behavior," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, 2002, pp. 45-57.
3. R.E. Wunderlich et al., "SMARTS:

Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," *Proc. 30th Int'l Symp. Computer Architecture* (ISCA 03), ACM Press, 2003, pp. 84-97.

4. M. Van Biesbrouck, L. Eeckhout, and B. Calder, "Efficient Sampling Startup for Sampled Processor Simulation," *Proc. Int'l Conf. High Performance Embedded Architectures and Compilation* (HiPEAC 05), Springer, 2005, pp. 47-67.
5. M. Durbhakula, V.S. Pai, and S. Adve, "Improving the Accuracy vs. Speed Tradeoff for Simulating Shared-Memory Multiprocessors with ILP Processors," *Proc. Fifth Int'l Symp. High-Performance Computer Architecture* (HPCA 99), IEEE CS Press, 1999, pp. 23-32.
6. J. Ringenberg et al., "Intrinsic Checkpointing: A Methodology for Decreasing Simulation Time through Binary Modification," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 05), IEEE, 2005, pp. 78-88.
7. T.W. Wenisch et al., "Simulation Sampling with Live-Points," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 06), IEEE, 2006, pp. 2-12.
8. J. Haskins and K. Skadron, "Accelerated Warmup for Sampled Microarchitecture Simulation," *ACM Trans. Architecture and Code Optimization,* vol. 2, no. 1, Mar. 2005, pp. 78-108.
9. L. Eeckhout et al., "BLRL: Accurate and Efficient Warmup for Sampled Processor Simulation," *The Computer J.,* vol. 48, no. 4, July 2005, pp. 451-459.

Michael Van Biesbrouck is a PhD student at the University of California, San Diego. His research interests include simulation methodology for computer architecture studies and the interaction between computer architecture and operating systems. He obtained his BMath and MMath in computer science from the University of Waterloo. He is a student member of IEEE.

Brad Calder is a professor of computer science and engineering at the University of California, San Diego, and an architect at Microsoft in the Windows Core operating system. His research interests focus on understanding program behavior toward software and hardware optimization. He cofounded *ACM Transactions on Architecture and Code Optimization,* and the International Conference on Code Generation and Optimization. He is a cofounder of three startups—TracePoint (x86 binary instrumentation tools), Entropia (desktop grid computing), and BitRaker (ARM binary instrumentation tools). He has a BS in computer science and a BS in mathematics from the University of Washington and a PhD in computer science from University of Colorado, Boulder. He is a senior member of IEEE.

Lieven Eeckhout is a postdoctoral fellow of the Fund for Scientific Research Flanders (Belgium) affiliated with Ghent University. His research interests include computer architecture and performance modeling. He has a PhD in computer science and engineering from Ghent University. He is a member of the IEEE.

Direct questions and comments about this article to Brad Calder, University of California, San Diego, Department of Computer Science and Engineering, 9500 Gilman Drive, Dept 0404, La Jolla, CA 92093-0404; calder@cs.ucsd.edu.