Efficient Sampling Startup for Sampled Processor Simulation

Michael Van Biesbrouck† Lieven Eeckhout‡ Brad Calder†

†CSE Department, University of California, San Diego
‡ELIS Department, Ghent University, Belgium
{mvanbies,calder}@cs.ucsd.edu, {leeckhou}@elis.UGent.be

Abstract

Modern architecture research relies heavily on detailed pipeline simulation. Simulating the full execution of an industry standard benchmark can take weeks to months. Statistical sampling and sample techniques like SimPoint that pick small sets of execution samples have been shown to provide accurate results while significantly reducing simulation time. The inefficiencies in sampling are (a) needing the correct memory image to execute the sample, and (b) needing a warm architecture state when simulating the sample.

In this paper we examine efficient Sampling Startup techniques addressing two issues: how to represent the correct memory image during simulation, and how to deal with warmup. Representing the correct memory image ensures the memory values consumed during the sample's simulation are correct. Warmup techniques focus on reducing error due to the architecture state not being fully representative of the complete execution that proceeds the sample to be simulated. This paper presents several Sampling Startup techniques and compares them against previously proposed techniques. The end result is a practical sampled simulation methodology that provides accurate performance estimates of complete benchmark executions in the order of minutes.

1 Introduction

Modern computer architecture research relies heavily on cycle-accurate simulation to help evaluate new architectural features. In order to measure cycle-level events and to examine the effect that hardware optimizations would have on the whole program, architects are forced to execute only a small subset 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 this approximation, and picking these points so that they are as *representative* as possible of the full program is a topic of several research studies [1–4]. The two bottlenecks in using these sampling techniques are the efficiency of having (a) the correct memory image to execute the sample, and (b) warm architecture state when simulating the sample. We collectively refer to both issues as *Sampling Startup*.

1.1 Sample Starting Image

The first issue to deal with is how to accurately provide a sample's starting image. The *Sample Starting Image* (SSI) is the state needed to accurately emulate

and simulate the sample's execution to achieve the correct output for that sample[1]. The two traditional approaches for providing the SSI are fast-forwarding and using checkpoints. Fast-forwarding quickly emulates the program's execution from the start of execution or from the last sample to the current sample. The advantage of this approach is that this is trivial for all simulators to implement. The disadvantage is that it serializes the simulation of all of the samples for a program, and it is non-trivial to have a low-overhead fast-forwarding implementation—most fast-forwarding implementations in current simulators are fairly slow.

Checkpointing is the process of storing the program's image right before the sample of execution is to start. This is similar to storing a core dump of the program so that it can be replayed at that point in execution. A checkpoint stores the register contents and the memory state prior to a sample. The advantage of checkpointing is that it allows for efficient parallel simulation. The disadvantage is that if a full checkpoint is taken it can be huge and consume too much disk space and take too long to load.

In this paper we examine two efficient ways of storing the SSI. One is a reduced checkpoint where we only store in the checkpoint the words of memory that are to be accessed in the sample we are going to simulate. The second approach is very similar, but is represented differently. For this approach we store a sequence of executed load values for the complete sample. Both of these approaches take about the same disk space, which is significantly smaller than a full checkpoint. Since they are small they also load instantaneously and are significantly faster than using fast-forwarding and full checkpoints.

1.2 Sample Architecture Warmup

Once we have an efficient approach for dealing with the sample's starting image we also need to reduce as much error in simulation due to the architecture components not being in the same state as if we simulated the full detailed execution from the start of the program up to that simulation point. To address this we examine a variety of previously proposed techniques and compare them to storing the detailed state of the memory hierarchy as a form of architecture checkpoint.

We first examine a technique called "Hit on Cold" which assumes that all architecture components are cold and the first access to it during the sample's simulation is a hit. A second technique we study uses a fixed warmup period before the execution of each sample. Recently, more sophisticated warmup techniques [5–7] have focused on finding for each sample how far back in the instruction stream to go to start warming up the architecture structures. We examine the performance of MRRL [6, 7] in this paper. An important advantage of this type of technique is its accuracy. The disadvantage is that it requires architecture component simulation for N million instructions before detailed simulation of the sample, which adds additional overhead to simulation.

[1] For convenience of exposition, we use 'sample' as a noun to refer to a sampling unit and 'sample' as a verb to refer to collecting a sample unit.

The final technique we examine is storing an architecture checkpoint of the major architecture components at the start of the sample. This *Architecture Checkpoint* is used to faithfully recreate the state of the major architecture components, such as caches, TLBs and branch predictors at the start of the sample. It is important that this approach works across different architecture designs for it to be used for architecture design space explorations. To that end, we examine a form of architecture checkpointing that allows us to create the smaller size instances of that architecture component. For example, you would create an architecture checkpoint of the largest cache you would look at in your design space exploration study, and the way we store the architecture checkpoint will allow smaller sizes and associativities to be faithfully recreated.

2 Sampled Simulation Background

Detailed cycle-by-cycle simulation of complete benchmarks is practically impossible due to the huge dynamic instruction counts of today's benchmarks (often several hundred billions of instructions), especially when multiple processor configurations need to be simulated during design space explorations. Sampling is an efficient way for reducing the total simulation time. There exist two ways of sampling, statistical sampling and phase-based sampling.

2.1 Statistical Sampling

Statistical sampling takes a number of execution samples across the whole execution of the program, which are referred to as clusters in [1] because they are groupings of contiguous instructions. These clusters are spread out throughout the execution of the program in an attempt to provide a representative cross-cut of the application being simulated. Conte *et al.* [1] formed multiple simulation points by randomly picking intervals of execution, and then examining how these fit to the overall execution of the program for several architecture metrics (IPC, branch and data cache statistics).

SMARTS [4] provides a version of SimpleScalar [8] using statistical simulation, which uses statistics to tell users how many samples need to be taken in order to reach a certain level of confidence. One consequence of statistical sampling is that tiny samples are gathered over the complete benchmark execution. This means that in the end the complete benchmark needs to be functionally simulated, and for SMARTS, the caches and branch predictors are warmed through the complete benchmark execution. This ultimately impacts the overall simulation time.

2.2 SimPoint

The SimPoint [3] sampling approach picks a small number of samples, that when simulated, accurately create a representation of the complete execution of the program. To do this they break a program's execution into intervals, and for each interval they create a code signature. They then perform 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, and this has been shown to be the case in [3, 9–11]. Therefore, only one interval from each phase needs to be simulated in order to recreate

a complete picture of the program's execution. They then choose a representative from each phase and perform detailed simulation on that interval. Taken together, these samples can represent the complete execution of a program. The set of chosen samples are called *simulation points*, and each simulation point is an interval on the order of millions of instructions. The simulation points were found by examining only a profile of the basic blocks executed for a program.

In this paper we focus on studying the applicability of the Sample Startup techniques presented for SimPoint. In addition, we also provide summary results for applying these Sample Startup techniques to SMARTS.

3 Sampling Startup Related Work

This section discusses prior work on Sample Startup techniques. We discuss checkpointing and fast-forwarding for obtaining a correct SSI, and warmup techniques for obtaining an architecture checkpoint as accurately as possible.

3.1 Starting Sample Image

As stated in the introduction, starting the simulation of a sample is much faster under checkpointing than under fast-forwarding (especially when the sample is located deep in the program's execution trace—fast-forwarding in such a case can take several days). The major disadvantage of checkpoints however is their size; they need to be saved on disk and loaded at simulation time. The checkpoint reduction techniques presented in this paper make checkpointing a much better alternative to fast-forwarding as will be shown in the evaluation section of this paper.

Szwed *et al.* [12] propose to fast-forward between samples through native hardware execution, called direct execution, and to use checkpointing to communicate the application state to the simulator. The simulator then runs the detailed processor simulation of the sample using this checkpoint. When the end of the sample is reached, native hardware execution comes into play again to fast-forward to the next simulation point, *etc.* Many ways to incorporate direct hardware execution into simulators for speeding up the simulation and emulation systems have been proposed, see for example [13–16].

One requirement for fast-forwarding through direct execution is that the simulation needs to be run on a machine with the same ISA as the program that is to be simulated. One possibility to overcome this limitation for cross-platform simulation would be to employ techniques from dynamic binary translation methods such as just-in-time (JIT) compilation and caching of translated code, as is done in Embra [17], or through compiled instruction-set simulation [18, 19]. Adding a dynamic binary compiler to a simulator is a viable solution, but doing this is quite an endeavor, which is why most contemporary out-of-order simulators do not include such functionality. In addition, introducing JITing into a simulator also makes the simulator less portable to host machines with different ISAs. Checkpoints, however, are easily portable.

Related to this is the approach presented by Ringenberg *et al.* [20]. They present intrinsic checkpointing, which takes the SSI image from the previous simulation interval and uses binary modification to bring the image up to state for the current simulation interval. Bringing the image up to state for the current

simulation interval is done by comparing the current SSI against the previous SSI, and by providing fix-up checkpointing code for the loads in the simulation interval that see different values in the current SSI versus the previous SSI. The fix-up code for the current SSI then executes stores to put the correct data values in memory. Our approach is easier to implement as it does not require binary modification. In addition, when implementing intrinsic checkpointing one needs to be careful to make sure the fix-up code is not simulated so that it does not affect the cache contents and branch predictor state for warmup.

3.2 Warmup

There has been a lot of work done on warmup techniques, or approximating the hardware state at the beginning of a sample. This work can be divided roughly in three categories: (*i*) simulating additional instructions prior to the sample, (*ii*) estimating the cache miss rate in the sample, and (*iii*) storing the cache content or taking an architecture checkpoint. In the evaluation section of this paper, we evaluate four warmup techniques. These four warmup techniques were chosen in such a way that all three warmup categories are covered in our analysis.

Warmup N Instructions Before Sample - The first set of warmup approaches simulates additional instructions prior to the sample to warmup large hardware structures [1, 4, 6, 7, 21–25]. A simple warmup technique is to provide a fixed-length warmup prior to each sample. This means that prior to each sample, caches and branch predictors are warmed by, for example, 1 million of instructions. MRRL [6, 7] on the other hand, analyzes memory references and branches to determine where to start warming up caches and branch predictors prior to the current sample. Their goal is to automatically calculate how for back in execution to go before a sample in order to capture the data and branch working set needed for the cache and branch predictor to be simulated. MRRL examines both the instructions between the previous sample and the current sample and the instructions in the sample to determine the correct warmup period. BLRL [21], which is an improvement upon MRRL, examines only references that are used in the sample to see how far one needs to go back before the sample for accurate warmup.

SMARTS [4] uses continuous warmup of the caches and branch predictors between two samples, *i.e.*, the caches and branch predictor are kept warm by simulating the caches and branch predictor continuously between two samples. This is called functional warming in the SMARTS work. The reason for supporting continuous warming is their small sample sizes of 1000 instructions. Note that continuously warming the cache and branch predictor slows down fast-forwarding.

The warmup approaches from this category that are evaluated in this paper are fixed-length warmup and MRRL.

Estimating the Cache Miss Rate - The second set of techniques does not warm the hardware state prior to the sample but estimates which references in the sample are cold misses due to an incorrect sample warmup [26, 27]. These misses are then excluded from the miss rate statistics when simulating the sample. Note that this technique in fact does no warmup, but rather estimates what

the cache miss rate would be for a perfectly warmup hardware state. Although these techniques are useful for estimating cache miss rate under sampled simulation, extending these techniques to processor simulation is not straight-forward. The hit-on-cold approach evaluated in this paper is another example of cache miss rate estimation; the benefit of hit-on-cold over the other estimation techniques is its applicability to detailed processor simulation.

Checkpointing the Cache Content - Lauterbach [28] proposes storing the cache tag content at the beginning of each sample. This is done by storing tags for a range of caches as they are obtained from stack simulation. This approach is similar to the Memory Hierarchy State (MHS) approach presented in this paper (see section 5 for more details on MHS). However, there is one significant difference. We compute the cache content for one single large cache and derive the cache content for smaller cache sizes. Although this can be done through stack simulation, it is still significantly slower and more disk space consuming than simulating only one single cache configuration as we do.

The Memory Timestamp Record (MTR) presented by Barr *et al.* [29] is also similar to the MHS proposed here. The MTR allows for the reconstruction of the cache and directory state for multiprocessor simulation by storing data about every cache block. The MTR is largely independent of cache size, organization and coherence protocol. Unlike MHS, its size is proportional to program memory. This prior work did not provide a detailed comparison between their architectural checkpointing approach and other warmup strategies; in this paper, we present a detailed comparison between different warmup strategies.

3.3 SMARTS and TurboSMARTS

Wunderlich *et al.* [4] provide SMARTS, an accurate simulation infrastructure using statistical sampling. SMARTS continuously updates caches and branch predictors while fast-forwarding between samples of size 1000 instructions. In addition, it also warms up the processor core before taking the sample through the detailed cycle-by-cycle simulation of 2000 to 4000 instructions.

At the same time we completed the research for our paper, TurboSMARTS [30] presented similar techniques that replace functional warming with a checkpointed SSI and checkpointed architectural state similar to what we discuss in our paper. In addition to what was studied in [30], we compare a number of reduced checkpointed SSI techniques, we study the impact of wrong-path load instructions for our techniques, and we examine the applicability of checkpointed sampling startup techniques over different sample sizes.

4 Sample Starting Image

The first issue to deal with to enable efficient sampled simulation is to load a memory image that will be used to execute the sample. The *Sample Starting Image* (SSI) is the program memory state needed to enable the correct functional simulation of the given sample.

4.1 Full Checkpoint

There is one major disadvantage to checkpointing compared to fast-forwarding and direct execution for providing the correct SSI. This is the large checkpoint

files that need to be stored on disk. Using many samples could be prohibitively costly in terms of disk space. In addition, the large checkpoint file size also affects total simulation time due to loading the checkpoint file from disk when starting the simulation of a sample and transferring over a network during parallel simulation.

4.2 EIO Files and Checkpointing System Calls

Before presenting our two approaches to reduce the checkpoint file size, we first detail our general framework in which the reduced checkpoint methods are integrated. We assume that the program binary and its input are available through an EIO file during simulation. We use compressed SimpleScalar EIO files; this does not affect the generality of the results presented in this paper however. An EIO file contains a checkpoint of the initial program state after the program has been loaded into memory. Most of the data in this initial program image will never be modified during execution. The rest of the EIO file contains information about every system call, including all input and output parameters and memory updates associated with the calls. This keeps the system calls exactly the same during different simulation runs of the same benchmarks.

In summary, for all of our results, the instructions of the simulated program are loaded from the program image in the EIO file, and the program is not stored in our checkpoints. Our reduced checkpoints focus only on the data stream.

4.3 Touched Memory Image

Our first reduced checkpoint approach is the *Touched Memory Image (TMI)* which only stores the blocks of memory that are to be accessed in the sample that is to be simulated. The TMI is a collection of chunks of memory (touched during the sample) with their corresponding memory addresses. The TMI contains only the chunks of memory that are read during the sample. Note that a TMI is stored on disk for each sample. At simulation time, prior to simulating the given sample, the TMI is loaded from disk and the chunks of memory in the TMI are then written to their corresponding memory addresses. This guarantees a correct SSI when starting the simulation of the sample. A small file size is achieved by using a sparse image representation, so regions of memory that consist of consecutive zeros are not stored in the TMI. In addition, large regions of non-zero sections of memory are combined and stored as one chunk. This saves storage space in terms of memory addresses in the TMI, since only one memory address needs to be stored for a large consecutive data region.

An optimization to the TMI approach, called the *Reduced Touched Memory Image* (*RTMI*), only contains chunks of memory for addresses that are read before they are written. There is no need to store a chunk of memory in the reduced checkpoint in case that chunk of memory is written prior to being read. A TMI, on the other hand, contains chunks of memory for all reads in the sample.

4.4 Load Value Sequence

Our second approach, called the *Load Value Sequence (LVS)*, involves creating a log of load values that are loaded into memory during the execution of the sample. Collecting an LVS can be done with a functional simulator or binary

instrumentation tool, which simply collects all data values loaded from memory during program execution (excluding those from instruction memory and speculative memory accesses). When simulating the sample, the load log sequence is read concurrently with the simulation to provide correct data values for non-speculative loads. The result of each load is written to memory so that, potentially, speculative loads accessing that memory location will find the correct value. The LVS is stored in a compressed format to minimize required disk space. Unlike TMI, LVS does not need to store the addresses of load values. However, programs often contain many loads from the same memory addresses and loads with value 0, both of which increase the size of LVS without affecting TMI.

In order to further reduce the size of the LVS, we also propose the *Reduced Load Value Sequence (RLVS)*. For each load from data memory the RLVS contains one bit, indicating whether or not the data needs to be read from the RLVS. If necessary, the bit is followed by the data value, and the data value is written to the simulator's memory image at the load address so that it can be found by subsequent loads; otherwise, the value is read from the memory image and not included in the RLVS. Thus the RLVS does not contain load values when a load is preceded by a load or store for the same address or when the value would be zero (the initial value for memory in the simulator). This yields a significant additional reduction in checkpoint file sizes. An alternate structure that accomplishes the same task is the first load log presented in [31].

5 Sample Warmup

In this paper we compare five warmup strategies, not performing any warmup, hit on cold, 1M-instructions of detailed execution fixed warmup, MRRL and stored architecture state. The descriptions in this section summarize the warmup techniques in terms of how they are used for uniprocessor architecture simulation.

5.1 No Warmup

The no-warmup strategy assumes an empty cache at the beginning of each sample, *i.e.* assumes no warmup. Obviously, this will result in an overestimation of the number of cache misses, and by consequence an underestimation of overall performance. However, the bias can be small for large sample sizes. This strategy is very simple to implement and incurs no runtime overhead.

5.2 Hit on Cold

The *hit on cold* strategy also assumes 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. The no warmup strategy, on the other hand, assumes a miss for the first use of a cache block in the sample. Hit on cold works well for programs that have a high hit rate, but it requires modifying the simulator to check a bit on every cache miss. If the bit indicates that the cache block has yet to be used the sample then the address tag is added to the cache but the access is considered to be a hit.

An extension to this technique is to try to determine the overall program's average hit rate or the approximate hit rate for each sample, then use this probability to label the first access to a cache block as a miss or a hit. We did not evaluate this approach for this paper.

5.3 Memory Reference Reuse Latency

The *Memory Reference Reuse Latency (MRRL)* [6, 7] approach proposed by Haskins and Skadron builds on the notion of memory reference reuse latency. The memory reference reuse latency is defined as the number of dynamic instructions between two consecutive memory references to the same memory location. To compute the warmup starting point for a given sample, MRRL first computes the reuse latency distribution over all the instructions from the end of the previous sample until the end of the current sample. This distribution gives an indication about the temporal locality behavior of the references.

MRRL subsequently determines w_N which corresponds to the $N\%$ percentile over the reuse latency distribution. This is the point at which $N\%$ of memory references, between the end of the last sample to the end of the current sample, will be made to addresses last accessed within w_N instructions. Warming then gets started w_N instructions prior to the beginning of the sample. The larger $N\%$, the larger w_N, and thus the larger the warmup. In this paper we use $N\% = 99.9\%$ as proposed in [6].

Sampled simulation under MRRL then proceeds as follows. The first step is to fast-forward to or load the checkpoint at the starting point of the warmup simulation phase. From that point on until the starting point of the sample, functional simulation is performed in conjunction with cache and branch predictor warmup, *i.e.* all memory references warm the caches and all branch addresses warm the branch predictors. When the sample is reached, detailed processor simulation is started for obtaining performance results. The cost of the MRRL approach is the w_N instructions that need to be simulated under warmup.

5.4 Memory Hierarchy State

The fourth warmup strategy is the *Memory Hierarchy State (MHS)* approach, which stores cache state so that caches do not need to be warmed at the start of simulation. The MHS is collected through cache simulation, *i.e.* functional simulation of the memory hierarchy. Design-space exploration may require many different cache configurations to be simulated. Note that the MHS needs to be collected only once for each block size and replacement policy, but can be reused extensively during design space exploration with smaller-sized memory hierarchies. Our technique is similar to trace-based construction of caches, except that storing information in a cache-like structure decreases both storage space and time to create the cache required for simulation. In addition to storing cache tags, we store status information for each cache line so that dirty cache lines are correctly marked.

Depending on the cache hierarchies to be simulated, constructing hierarchies of *target caches* for simulation from a single *source cache* created by functional simulation can be complicated, so we explain the techniques used and the necessary properties of the source cache. If a target cache i has s_i sets with w_i ways, then every cache line in it would be contained in a source cache with $s' = c_i s_i$ sets and $w' \geq w_i$ ways, where c_i is a positive integer. We now describe how to initialize the content of each set in the target cache from the source cache. To initialize the contents of the first set in the target cache we need to look at

the cache blocks in c_i sets of the source cache that map to the first set in the target cache. This gives us $c_i w'$ blocks to choose from. For a cache with LRU replacement we need to store a sequence number for the most recent use of each cache block. We select the most recently used w_i cache blocks, as indicated by the data in our source cache, to put in the target set. The next c_i sets of the source cache initialize the second set of the target cache, and so forth. In general, $s' = \texttt{LCM}_i(s_i)$ and $w' = \texttt{max}_i(w_i)$ ensure that the large cache contains enough information to initialize all of the simulated cache configurations. In the common case where s_i is always a power of two, the least common multiple (LCM) is just the largest such value.

Inclusive cache hierarchies can be initialized easily as just described, but exclusive cache hierarchies need something more. For example, assume that the L2 cache is 4-way associative and has the same number of cache sets as a 2-way associative L1 cache. Then the 6 most recently accessed blocks mapped to a single cache set will be stored in the cache hierarchy, 2 in the L1 cache and 4 in the L2 cache. Thus the source cache must be at least 6-way associative. If, instead, the L2 cache has twice as many sets as the L1 cache then the cache hierarchy will contain 10 blocks that are mapped to the same L1 cache set, but at most 6 of these will be mapped to either of the L2 cache sets associated with the L1 cache set. The source cache still only needs to be 6-way associative.

We handle exclusive cache hierarchies by creating the smallest (presumably L1) target cache first and locking the blocks in the smaller cache out of the larger (L2, L3, *etc.*) caches. Then the sets in the larger cache can be initialized. Also, the associativity of the source cache used to create our MHS must be at least the sum of the associativities of the caches within a target cache hierarchy.

Unified target caches can be handled by collecting source cache data as if the target caches were not unified. For example, if there are target IL1, DL1 and UL2 caches then data can be collected using the same source caches as if there were IL2 and DL2 caches with the same configuration parameters as the UL2 cache. Merging the contents of two caches into a unified target cache is straight-forward. The source caches must have a number of cache sets equal to the LCM of all the possible numbers of cache sets (both instruction and data) and an associativity at least as large as that of any of the caches. Alternately, if all of the target cache hierarchy configurations are unified in the same way then a single source cache with the properties just described can collect all of the necessary data.

Comparing MHS versus MRRL, we can say that they are equally architecture-independent. MHS traces store all addresses needed to create the largest and most associative cache size of interest. Similarly, MRRL goes back in execution history far enough to also capture the working set for the largest cache of interest. The techniques have different tradeoffs, however: MHS requires additional memory space compared to MRRL, and MRRL just needs to store where to start the warming whereas MHS stores a source cache. In terms of simulation speed, MHS substantially outperforms MRRL as MHS does not need to simulate instructions to warm the cache as done in MRRL—loading the MHS trace is done

I Cache	8k 2-way set-assoc., 32 byte blocks, 1 cycle latency
D Cache	16k 4-way set-assoc., 32 byte blocks, 2 cycle latency
L2 Cache	1Meg 4-way set-assoc., 32 byte blocks, 20 cycle latency
Memory	150 cycle round trip access
Branch Pred	hybrid
O-O-O Issue	up to 8 inst. per cycle, 128 entry re-order buffer
Func Units	8-integer ALU, 4-load/store units, 2-FP adders, 2-integer MULT/DIV, 2-FP MULT/DIV

Table 1. Processor simulation model.

very quickly. As simulated cache sizes increase, MHS disk space requirements increase and MRRL warming times increase.

The techniques discussed in this paper can also be extended to warmup for TLBs and branch predictors. For 1M-instruction simulation points, we only consider sample architecture warmup for caches. We found that the branch predictor did not have a significant effect until we used very small 1000-instruction intervals with SMARTS. When simulating the tiny SMARTS simulation intervals we checkpointed the state of branch predictor prior to each sample. While we can modify TLB and cache checkpoints to work with smaller TLBs and caches, we cannot yet do this in general for branch predictors. For experiments requiring branch predictor checkpoints we simulate all of the branch predictors to be used in the design space exploration concurrently and store their state in the sample checkpoints.

6 Evaluation

We now evaluate Sample Startup for sampled simulation. After discussing our methodology, we then present a detailed error analysis of the warmup and reduced checkpointing techniques. We subsequently evaluate the applicability of the reduced checkpointing and warmup techniques for both targeted sampling as done in SimPoint and statistical sampling as done in SMARTS.

6.1 Methodology

We use the MRRL-modified SimpleScalar simulator [6], which supports taking multiple samples interleaved with fast-forwarding and functional warming. Minor modifications were made to support (reduced) checkpoints. We simulated SPEC 2000 benchmarks compiled for the Alpha ISA and we used reference inputs for all of these. The binaries we used in this study and how they were compiled can be found at `http://www.simplescalar.com/`. The processor model assumed in our experiments is summarized in Table 1.

6.2 Detailed error analysis

We first provide a detailed error analysis of our warmup techniques as well as the reduced checkpointing techniques. For the error analysis, we consider 50 1M-instruction sampling units randomly chosen from the entire benchmark execution. We also experimented with a larger number of sampling units, however, the results were quite similar.

The *average CPI error* is the average over all samples of the relative difference between the CPI through sampled simulation with full warmup, versus the CPI

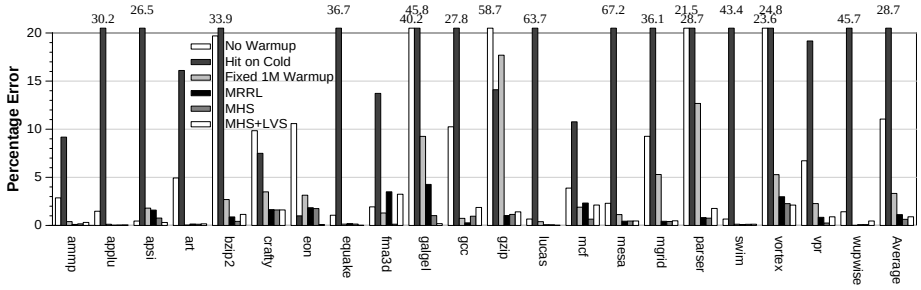

Fig. 1. Average CPI error: average CPI sample error as a percentage of CPI.

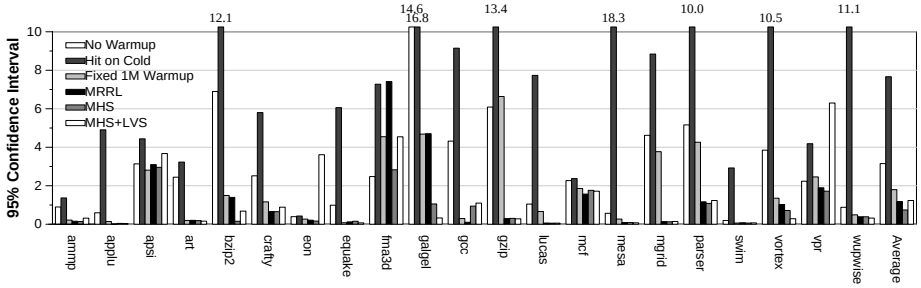

Fig. 2. The 95% confidence interval as a percentage of CPI.

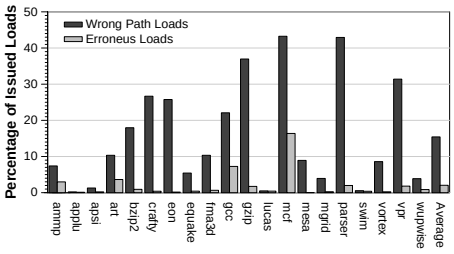

Fig. 3. Analysis of wrong-path loads while using LVS.

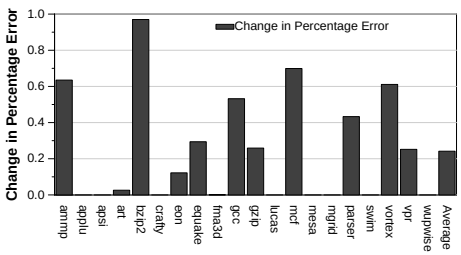

Fig. 4. Change in percentage error due to LVS.

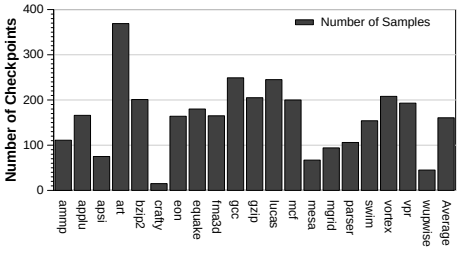

Fig. 5. Number of Simulation Point samples used.

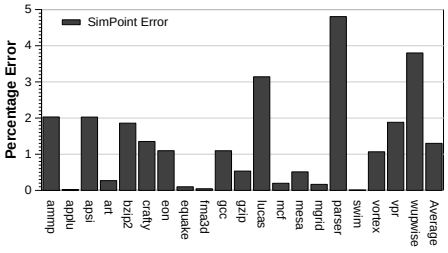

Fig. 6. Accuracy of SimPoint assuming perfect sampling.

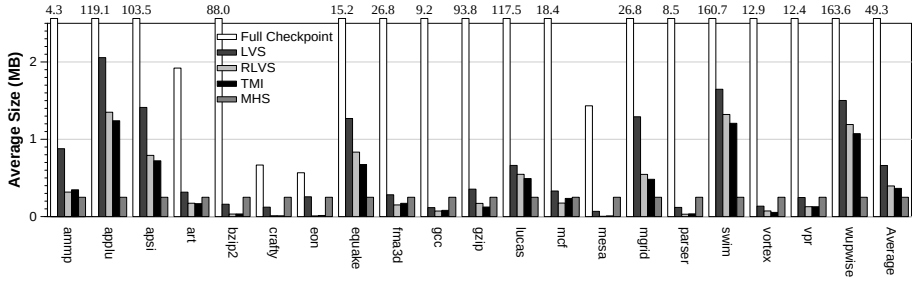

Fig. 7. Average storage requirements per sample for full checkpointing, reduced checkpointing (LVS, RLVS and TMI) and sample warmup through MHS.

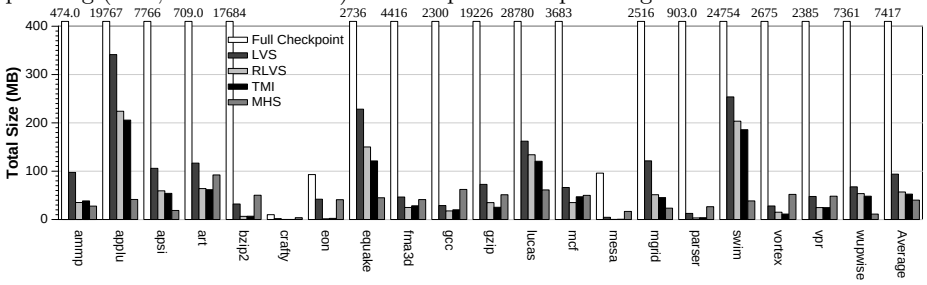

Fig. 8. Total storage requirements per benchmark for full checkpointing, reduced checkpointing (LVS, RLVS and TMI) and sample warmup through MHS.

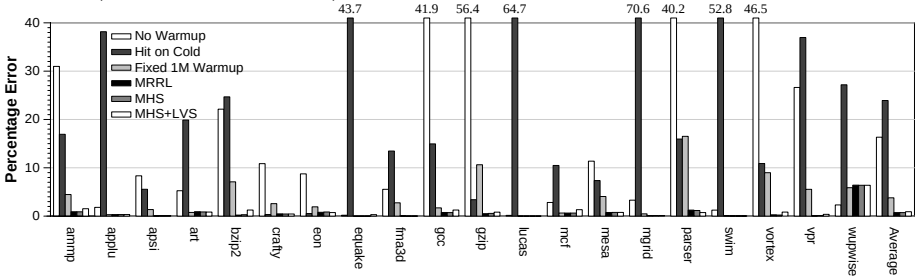

Fig. 9. Percentage error in estimating overall CPI as compared to CPI estimated by SimPoint with no sampling error.

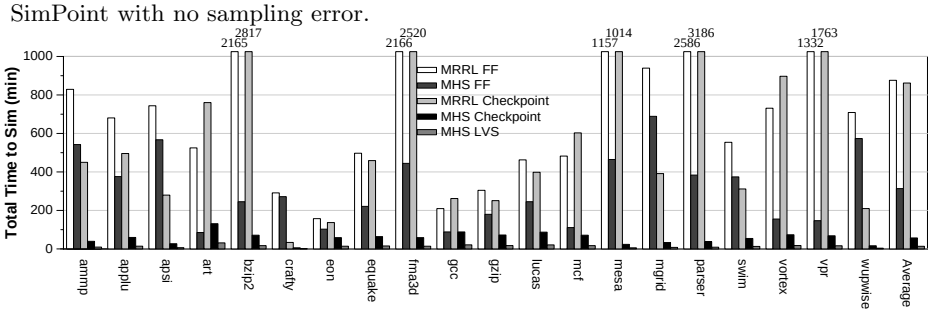

Fig. 10. Total time to simulate all samples including fast-forwarding, loading checkpoints, warming and doing detailed simulation.

through sampled simulation with the warmup and reduced checkpoint techniques proposed in this paper. Our second metric is the *95% confidence interval* for average CPI error. Techniques that are both accurate and precise will have low average CPI error and small confidence interval widths.

Figures 1 and 2 show the average CPI error and the 95% confidence interval, respectively. In both cases they are expressed as a percentage of the correct CPI. The various bars in these graphs show full SSI checkpointing along with a number of architectural warmup strategies (no warmup, hit on cold, fixed 1M warmup, MRRL and MHS), as well as a reduced SSI checkpointing technique, namely LVS, in conjunction with MHS. We only present data for the LVS reduced SSI for readability reasons; we obtained similar results for the other reduced SSI techniques. In terms of error due to warmup, we find that the no-warmup, hit-on-cold and fixed 1M warmup strategies perform poorly. MRRL and MHS on the other hand, are shown to perform equally well. The average error is less than a few percent across the benchmarks.

In terms of error due to the starting image, we see the error added due to the reduced SSI checkpointing is very small. Comparing the MHS bar (MHS with full checkpointing) versus the MHS+LVS bar, we observe that the error added is very small, typically less than 1%. The reason for this additional error is that under reduced SSI checkpointing, load instructions along mispredicted paths might potentially fetch wrong data from memory since the reduced checkpointing techniques only consider on-path memory references. In order to quantify this we refer to Figures 3 and 4. Figure 3 shows the percentage of wrong-path load instructions being issued relative to the total number of issued loads; this figure also shows the percentage of issued wrong-path loads that fetched incorrect data (compared to a fully checkpointed simulation) relative to the total number of issued loads. This graph shows that the fraction of wrong-path loads that are fetching uncheckpointed data is very small, 2.05% on average. Figure 4 then quantifies the difference in percentage CPI error due to these wrong-path loads fetching uncheckpointed data. We compare the CPI under full checkpoint versus the CPI under reduced checkpoints. The difference between the error rates is very small, under 1% of the CPI.

6.3 Targeted Sampling Using SimPoint

We now study the applicability of the reduced warmup and checkpointing techniques for two practical sampling methodologies, namely targeted sampling using SimPoint, and statistical sampling using SMARTS. For these results we used SimPoint with an interval size of 1 million with Max K set to 400. Figure 5 shows the number of 1M-instruction simulation points per benchmark. This is also the number of checkpoints per benchmark since there is one checkpoint needed per simulation point. The number of checkpoints per benchmark varies from 15 (`crafty`) up to 369 (`art`). In this paper, we focus on small, 1M-instruction intervals because SimPoint is most accurate when many (at least 50 to 100) small (1M instructions or less) intervals are accurately simulated. However, we found that the reduction in disk space is an important savings even for 10M and 100M interval sizes when using a reduced load value trace. Figure 6 shows the accuracy

of SimPoint while assuming perfect sampling startup. The average error is 1.3%; the maximum error is 4.8% (`parser`).

Storage Requirements - Figures 7 and 8 show the average and total sizes of the files (in MB) that need to be stored on disk per benchmark for various Sample Startup approaches: the Full Checkpoint, the Load Value Sequence (LVS), the Reduced Load Value Sequence (RLVS), the Touched Memory Image (TMI) and the Memory Hierarchy State (MHS). Clearly, the file sizes for Full Checkpoint are huge. The average file size per checkpoint is 49.3MB (see Figure 7). The average total file size per benchmark is 7.4GB (see Figure 8). Storing all full checkpoints for a complete benchmark can take up to 28.8GB (`lucas`). The maximum average storage requirements per checkpoint can be large as well, for example 163.6MB for `wupwise`. Loading and transferring over a network such large checkpoints can be costly in terms of simulation time as well.

The SSI techniques, namely LVS, RLVS, TMI and RTMI, result in a checkpoint reduction of more than two orders of magnitude, see Figures 7 and 8. The results for RTMI are similar to those for TMI, so they are not shown in the Figure. Since TMI contains at most one value per address and no zeros, size improvements can only come from situations where the first access to an address is a write and there is a later read from that address. This is fairly rare for the benchmarks examined, so the improvements are small.

The average total checkpoint file sizes per benchmark for LVS, RLVS and TMI are 93.9MB, 57MB and 52.6MB, respectively; the maximum total file sizes for are 341MB, 224MB and 206MB, respectively, for `applu`. These huge checkpoint file reductions compared to full checkpoints make checkpointing feasible in terms of storage cost for sampled simulation. Also, the typical single checkpoint size is significantly reduced to 661KB, 396KB and 365KB for LVS, RLVS and TMI, respectively. This makes loading the checkpoints highly efficient.

Memory Hierarchy State (MHS) was the only warmup approach discussed that requires additional storage. Figures 7 and 8 quantify the additional storage needed for MHS. The total average storage needed per benchmark is 40MB. The average storage needed for MHS per checkpoint is 256kB (8 bytes per cache block). Note that this is additional storage that is needed on top of the storage needed for the checkpoints. However, it can be loaded efficiently due to its small size.

Error analysis - Figure 9 evaluates the CPI error rates for various Sample Startup techniques as compared to the SimPoint method's estimate using perfect warmup — this excludes any error introduced by SimPoint. This graph compares four sample warmup approaches: no warmup, hit-on-cold, fixed 1M warmup, MRRL and MHS. The no warmup and hit-on-cold strategies result in high error rates, 16% and 24% on average. For many benchmarks one is dramatically better than the other, suggesting that an algorithm that intelligently chooses one or the other might do significantly better. The fixed 1M warmup achieves better accuracy with an average error of 4%; however the maximum error can be fairly large, see for example for `parser` (17%). The error rates obtained from MRRL and MHS are significantly better. The average error for both approaches is 1%.

As such, we conclude that in terms of accuracy, MRRL and MHS are equally accurate when used in conjunction with SimPoint.

The error results discussed so far assumed full checkpoints. Considering a reduced checkpoint technique, namely LVS, in conjunction with MHS increases the error rates only slightly, from 1% to 1.2%. This is due to the fact that LVS does not include load values for loads being executed along mispredicted paths.

Total simulation time - Figure 10 shows the total simulation time (in minutes) for the various Sample Startup techniques when simulating all simulation points on a single machine. This includes fast-forwarding, loading (reduced) checkpoints, loading the Memory Hierarchy State and warming structures by functional warming or detailed execution, if appropriate.

The SSI techniques considered here are fast-forwarding, checkpointing, and reduced checkpointing using the LVS—we obtained similar simulation time results for the other reduced checkpoint techniques RLVS, TMI and RTMI. These three SSI techniques are considered in combination with the two most accurate sample warmup techniques, namely MRRL and MHS. These results show that MRRL in combination with fast-forwarding and full checkpointing are equally slow. The average total simulation time is more than 14 hours per benchmark. If we combine MHS with fast-forwarding, the average total simulation time per benchmark cuts down to 5 hours. This savings over MRRL is achieved by replacing warming with fast-forwarding and loading the MHS. Combining MHS with full checkpointing cuts down the total simulation time even further to slightly less than one hour. Combining the reduced checkpoint LVS approach with MHS reduces the average total simulation time per benchmark to 13 minutes. We obtained similar simulation time results for the other reduced checkpoint techniques.

As such, we conclude that the Sample Startup techniques proposed in this paper achieve full detailed per-benchmark performance estimates with the same accuracy as MRRL. This is achieved in the order of minutes per benchmark which is a 63X simulation time speedup compared to MRRL in conjunction with fast-forwarding and checkpointing.

6.4 Using MHS and LVS with SMARTS

The SMARTS infrastructure [4] accurately estimates CPI by taking large numbers of very small samples and using optimized functional warming while fast-forwarding between samples. Typical parameters use approximately 10000 samples, each of which is 1000 instructions long and preceded by 2000 instructions of detailed processor warmup. Only 30M instructions are executed in detail, so simulation time is dominated by the cost of functional warming for tens or hundreds of billions of instructions.

We improved SMARTS' performance by replacing functional warming with our MHS and LVS techniques. Due to the very small sample length there was insufficient time for the TLB and branch predictor to warm before the end of detailed simulation warmup. Therefore, for SMARTS we enhanced MHS to include the contents of the TLBs and branch predictor. TLB structures can be

treated just like caches when considering various TLB sizes, but branch predictors need to be generated for every desired branch predictor configuration. With these changes we were able to achieve sampling errors comparable to the error rates presented in section 6.2 for the 1M-instruction samples. In addition, the estimated CPI confidence intervals are similar to those obtained through SMARTS.

Storing the entire memory image in checkpoints for 10000 samples is infeasible, so we used LVS. Due to the small number of loads in 3000 instructions, a compressed LVS only required a single 4 kB disk block per sample. The total disk space per benchmark for the LVS checkpoint is 40 MB. Disk usage however is dominated by MHS, with total storage requirements of approximately 730 MB for each benchmark. By comparison, our SimPoint experiments used under 100 MB on average for full LVS, 50 MB for RLVS and 40 MB for MHS. In terms of disk space, SimPoint thus performs better than SMARTS. On average, the total simulation time per benchmark for SMARTS with LVS and MHS is 130 seconds on average. About two-thirds of this time is due to decompressing the MHS information. Our results assumed 10000 samples, but more samples may be needed in order to reach a desired level of confidence, which would require more simulation time.

In contrast to the fact that the amount of disk space required is approximately 8 times larger with SMARTS, SMARTS is faster than SimPoint: 130 seconds for SMARTS versus 13 minutes for SimPoint. The reason for this is the larger of number of simulated instructions for SimPoint than for SMARTS.

Concurrently with our work, the creators of SMARTS have released TurboSMARTS [30], which takes a similar approach to the one that we have outlined here. Their documentation for the new version recommends estimating the number of samples that should be taken when collecting their version of MHS and TMI data. The number of samples is dependent upon the program's variability, so for floating-point benchmarks this can greatly reduce the number of samples, but in other cases more samples will be required. As a result, the average disk usage is 290 MB per benchmark, but varies from 7 MB (`swim`) to 1021 MB (`vpr`). This is still over twice as large than the disk space required for SimPoint using 1 million interval sizes.

7 Summary

Today's computer architecture research relies heavily on detailed cycle-by-cycle simulation. Since simulating the complete execution of an industry standard benchmark can take weeks to months, several researchers have proposed sampling techniques to speed up this simulation process. Although sampling yields substantial simulation time speedups, there are two remaining bottlenecks in these sampling techniques, namely efficiently providing the sample starting image and sample architecture warmup.

This paper proposed reduced checkpointing to obtain the sample starting image efficiently. This is done by only storing the words of memory that are to be accessed in the sample that is to be simulated, or by storing a sequence of load values as they are loaded from memory in the sample. These reduced

checkpoints result in two orders of magnitude less storage than full checkpointing and faster simulation than both fast-forwarding and full checkpointing. We show that our reduced checkpointing techniques are applicable on various sampled simulation methodologies as we evaluate them for SimPoint, random sampling and SMARTS.

This paper also compared four techniques for providing an accurate hardware state at the beginning of each sample. We conclude that architecture checkpointing and MRRL perform equally well in terms of accuracy. However, our architecture checkpointing implementation based on the Memory Hierarchy State is substantially faster than MRRL. The end result for sampled simulation is that we obtain highly accurate per-benchmark performance estimates (only a few percent CPI prediction error) in the order of minutes, whereas previously proposed techniques required multiple hours.

Acknowledgments

We would like to thank the anonymous reviewers for providing helpful comments on this paper. This work was funded in part by NSF grant No. CCR-0311710, NSF grant No. ACR-0342522, UC MICRO grant No. 03-010, and a grant from Intel and Microsoft. Lieven Eeckhout is a Postdoctoral Fellow of the Fund for Scientific Research—Flanders (Belgium) (F.W.O. Vlaanderen); and he is a member of the HiPEAC network of excellence.

References

1. Conte, T.M., Hirsch, M.A., Menezes, K.N.: Reducing state loss for effective trace sampling of superscalar processors. In: ICCD'96. (1996)
2. Lafage, T., Seznec, A.: Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In: WWC-3. (2000)
3. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. In: ASPLOS-X. (2002)
4. Wunderlich, R.E., Wenisch, T.F., Falsafi, B., Hoe, J.C.: SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In: ISCA-30. (2003)
5. Eeckhout, L., Eyerman, S., Callens, B., De Bosschere, K.: Accurately warmed-up trace samples for the evaluation of cache memories. In: HPC'03. (2003) 267–274
6. Haskins, J., Skadron, K.: Memory reference reuse latency: Accelerated sampled microarchitecture simulation. In: ISPASS'03. (2003)
7. Haskins, J., Skadron, K.: Accelerated warmup for sampled microarchitecture simulation. ACM Transactions on Architecture and Code Optimization (TACO) **2** (2005) 78–108
8. Burger, D.C., Austin, T.M.: The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison (1997)
9. Lau, J., Sampson, J., Perelman, E., Hamerly, G., Calder, B.: The strong correlation between code signatures and performance. In: ISPASS. (2005)
10. Patil, H., Cohn, R., Charney, M., Kapoor, R., Sun, A., Karunanidhi, A.: Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In: MICRO-37. (2004)
11. Yi, J.J., Kodakara, S.V., Sendag, R., Lilja, D.J., Hawkins, D.M.: Characterizing and comparing prevailing simulation techniques. In: HPCA-11. (2005)

12. Szwed, P.K., Marques, D., Buels, R.M., McKee, S.A., Schulz, M.: SimSnap: Fast-forwarding via native execution and application-level checkpointing. In: INTERACT-8. (2004)
13. Durbhakula, M., Pai, V.S., Adve, S.: Improving the accuracy vs. speed tradeoff for simulating shared-memory multiprocessors with ILP processors. In: HPCA-5. (1999)
14. Fujimoto, R.M., Campbell, W.B.: Direct execution models of processor behavior and performance. In: Proceedings of the 1987 Winter Simulation Conference. (1987) 751–758
15. Mukherjee, S.S., Reinhardt, S.K., B. Falsafi, M.L., Huss-Lederman, S., Hill, M.D., Larus, J.R., Wood, D.A.: Wisconsin wind tunnel II: A fast and portable parallel architecture simulator. In: PAID'97. (1997)
16. Schnarr, E., Larus, J.R.: Fast out-of-order processor simulation using memoization. In: ASPLOS-VIII. (1998)
17. Witchel, E., Rosenblum, M.: Embra: Fast and flexible machine simulation. In: SIGMETRICS'96. (1996) 68–79
18. Nohl, A., Braun, G., Schliebusch, O., Leupers, R., Meyr, H., Hoffmann, A.: A universal technique for fast and flexible instruction-set architecture simulation. In: DAC-41. (2002)
19. Reshadi, M., Mishra, P., Dutt, N.: Instruction set compiled simulation: A technique for fast and flexible instruction set simulation. In: DAC-40. (2003)
20. Ringenberg, J., Pelosi, C., Oehmke, D., Mudge, T.: Intrinsic checkpointing: A methodology for decreasing simulation time through binary modification. In: ISPASS'05. (2005)
21. Eeckhout, L., Luo, Y., De Bosschere, K., John, L.K.: Blrl: Accurate and efficient warmup for sampled processor simulation. The Computer Journal **48** (2005) 451–459
22. Conte, T.M., Hirsch, M.A., Hwu, W.W.: Combining trace sampling with single pass methods for efficient cache simulation. IEEE Transactions on Computers **47** (1998) 714–720
23. Kessler, R.E., Hill, M.D., Wood, D.A.: A comparison of trace-sampling techniques for multi-megabyte caches. IEEE Transactions on Computers **43** (1994) 664–675
24. Luo, Y., John, L.K., Eeckhout, L.: Self-monitored adaptive cache warm-up for microprocessor simulation. In: SBAC-PAD'04. (2004) 10–17
25. Nguyen, A.T., Bose, P., Ekanadham, K., Nanda, A., Michael, M.: Accuracy and speed-up of parallel trace-driven architectural simulation. In: IPPS'97. (1997) 39–44
26. Laha, S., Patel, J.H., Iyer, R.K.: Accurate low-cost methods for performance evaluation of cache memory systems. IEEE Transactions on Computers **37** (1988) 1325–1336
27. Wood, D.A., Hill, M.D., Kessler, R.E.: A model for estimating trace-sample miss ratios. In: SIGMETRICS'91. (1991) 79–89
28. Lauterbach, G.: Accelerating architectural simulation by parallel execution of trace samples. In: Hawaii International Conference on System Sciences. (1994)
29. Barr, K.C., Pan, H., Zhang, M., Asanovic, K.: Accelerating multiprocessor simulation with a memory timestamp record. In: ISPASS'05. (2005)
30. Wenisch, T.F., Wunderlich, R.E., Falsafi, B., Hoe, J.C.: TurboSMARTS: Accurate microarchitecture simulation sampling in minutes. In: SIGMETRICS. (2005)
31. Narayanasamy, S., Pokam, G., Calder, B.: Bugnet: Continuously recording program execution for deterministic replay debugging. In: ISCA. (2005)