Uppsala University, Department of Information Technology, Technical Report 2016-004

Implementing Fast, Virtualized Profiling to Eliminate Cache Warming

Nikos Nikoleris, Andreas Sandberg, Erik Hagersten and Trevor E. Carlson
Department of Information Technology
Uppsala University
P.O. Box 337, SE-751 05, Uppsala, Sweden
Email: nikos.nikoleris,andreas.sandberg,erik.hagersten,trevor.carlson@it.uu.se

Abstract

Simulation is an important part of the evaluation of next-generation computing systems. Detailed, cycle-level simulation, however, can be very slow when evaluating realistic workloads on modern microarchitectures. Sampled simulation (e.g., SMARTS and SimPoint) improves simulation performance by an order of magnitude or more through the reduction of large workloads into a small but representative sample. Additionally, the execution state just prior to a simulation sample can be stored into checkpoints, allowing for fast restoration and evaluation. Unfortunately, changes in software, architecture or fundamental pieces of the microarchitecture (e.g., hardware-software co-design) require checkpoint regeneration. The end result for co-design degenerates to creating checkpoints for each modification, a task checkpointing was designed to eliminate. Therefore, a solution is needed that allows for fast and accurate simulation, without the need for checkpoints.

Virtualized fast-forwarding proposals, like FSA, are an alternative to checkpoints that speed up sampled simulation by advancing the execution at near-native speed between simulation points. They rely, however, on functional simulation to warm the architectural state prior to each simulation point, a costly operation for moderately-sized last-level caches (e.g., above 8MB). Simulating future systems with DRAM caches of many GBs can require warming of billions of instructions, dominating the time for simulation and negating the benefit of virtualized fast-forwarding.

This paper proposes CoolSim, an efficient simulation framework that eliminates cache warming. CoolSim advances between simulation points using virtualized fast-forwarding, while collecting sparse memory reuse information (MRI). The MRI is collected more than an order of magnitude faster than functional warming. At the simulation point, detailed simulation is used to evaluate the design while a statistical cache model uses the previously acquired MRI to estimate whether each memory request hits in the cache. The MRI is an architecturally independent metric and therefore a single profile can be used in simulations of any size cache. We describe a prototype implementation of CoolSim based on KVM and gem5 running 19x faster than the state-of-the-art sampled simulation, while it estimates the CPI of the SPEC CPU2006 benchmarks with 3.62% error on average, across a wide range of cache sizes.

I. Introduction

Detailed simulation can deliver high modeling accuracy. This accuracy, though, comes at a cost. Detailed simulation models, e.g., gem5 [1], run at ~10 KIPS, 5 orders of magnitude slower than native execution. The simulation overhead can be improved through various sampling techniques [2, 3]. With sampled simulation, a very small percentage of the application is simulated in detail – the simulation points. As a result, faithful simulation of multi-billion instruction workloads can be substituted by the simulation of the representative simulation points.

Prior work proposes methods for selecting representative simulation points of an application. Sherwood et al. [3] exploit the phase behavior of an application, while Wunderlich et al. [2] use uniform sampling in a statistically robust way.

Using checkpoints to store and restore the architectural state of a simulated system prior to the simulation points, subsequent simulations can readily start without the need to warm or fast-forward the entire application again. Therefore, the selected simulation points can be simulated immediately, which typically results in 1000-fold reduction in simulation time [2]. These techniques make slow full-system, detailed simulators more practical.

Modern checkpointing techniques allow for subsequent simulations with small microarchitectural changes, but even the slightest software change prevents the use of the checkpoint as a valid starting point. Modern software systems have dynamic properties, e.g., dynamic scheduling, finalizers (translating virtualized instruction sets into machine code) and jit compilation. Each configuration combination would therefore require an additional set of checkpoints to be created, potentially outweighing the cost of using checkpoints.

Therefore, in cases where the flexibility of software changes is needed, the only option is to re-run the entire application and appropriately warm the hardware state using functional simulation. This is a major setback since it increases simulation time. Worse, future systems are likely to employ mechanisms with larger state (e.g, DRAM caches) which make the exploration of such systems even harder.

The accuracy vs. efficiency trade-off of simulation sampling is largely determined by the speed of fast-forwarding, the warming strategy, and the overhead of detailed simulation.

a) Detailed simulation overhead: Some methods raise the level of abstraction of the detailed simulator to reduce its overhead. A different approach [2, 3] achieves high accuracy with the required confidence intervals simulating only 1% of the total instructions in detail. While making detailed simulation more efficient is an interesting problem, typically, its performance impact on sampled simulation frameworks is small.

b) Efficient fast-forwarding: While only 1% of the total instructions are simulated in detail, the execution still needs to move quickly, or fast-forward, to the relevant points in the execution of a workload for warming and simulation. Sandberg et al. [4] propose a simulation methodology, *Full Speed Ahead, (FSA)* [4] which aims to minimize the overhead of fast-forwarding compared to software-based functional fast-forwarding. FSA uses *virtualized fast-forwarding* to allow the hardware to execute the application up to each point of interest at near-native speed.

c) Efficient Warming: While fast-forwarding to a region of interest can now be done at near-native speeds, warming up microarchitectural structures efficiently can still be a challenge given recent technology and microarchitectural developments. 2.5D, 3D stacking and other integration technologies enable large on-chip caches (e.g., DRAM), and therefore studies on systems with caches of hundreds of MB are becoming very relevant. These systems require functional warming of ~1 B instructions, well beyond the practical range for methods that rely on functional cache warming. Figure 1 shows simulations with caches up to 512 MB using different warming lengths. Even relatively simple applications, e.g., 436.cactusADM-ref from the SPEC CPU2006 benchmarks, can require more than $400\,M$ instructions to sufficiently populate the cache prior to a simulation point. Such long warming would typically dominate the overhead of simulation sampling methods.

While the efficiency of fast-forwarding in FSA has improved the overall performance of workload simulation, the efficiency of simulation using FSA does not scale with the size of the simulated cache. Sandberg et al. [4] report good simulation speed for caches up to 8 MB, using $25\,M$ instructions for cache warming. The simulation speed they achieve is about 35 MIPS, which is mostly limited by this cache warming. Longer warming periods further decrease the efficiency of the method which approaches the speed of traditional software-based microarchitectural warming for large caches (~1 MIPS for gem5).

As an alternative to traditional cache warming, Nikoleris et al. [5] have proposed *WarmSim* [5] to predict the outcome of a cache lookup of any given memory request using memory reuse information (MRI). The MRI is a microarchitecturally independent property that uses the same input to perform detailed simulation with any size cache. However, their proposal is based on a functional-simulator-based MRI sampler, and the speed at which they can collect MRI is as slow as warming with functional simulation.

This paper proposes *CoolSim* to address the problem of efficient simulation of systems with large caches. CoolSim combines and enhances ideas from WarmSim and FSA, and provides an efficient integrated online statistical simulation framework for large caches. It substitutes functional simulation, which is typically used to warm caches, with virtualized fast-forwarding to advance to the next simulation point. At the same time, CoolSim allows for fast and efficient collection of an application's MRI. The MRI is used as an input to a statistical cache model at the simulation point and drives the memory behavior of the detailed simulator, i.e., predicts the outcome of a cache look-up of any memory request. In order to be accurate at high simulation speeds, the techniques proposed in FSA and WarmSim needed to be enhanced in several dimensions. The implementation of CoolSim was evaluated using a wide range of cache sizes from 1 MB to 512 MB with the SPEC CPU 2006 benchmarks. The average simulation speed is 17 MIPS, achieving a 19× speedup over traditional SMARTS simulation, while estimating CPI with an average error of 3.62%.

This paper makes contributions in the following areas:

- The simulation framework proposes a new MRI collection mechanism that runs on top of the virtualized fast-forwarding mode of gem5. We use *progressive sampling rate*, and *batch sampling* to allow for fast but still representative MRI sampling.
- We improve on WarmSim's cache modeling technology. WarmSim requires MRI that is too dense to be efficiently acquired in VFF. CoolSim, on the other hand, is accurate with sparse MRI by: 1) carefully selecting the vicinity in which any reuse distance is processed to determine the corresponding stack distance, 2) handling instructions with insufficient MRI, due to the low sampling rate, 3) leveraging the lukewarm cache and the architecture of non-blocking caches to accurately determine the cache outcome for requests that expose temporal locality. Additionally, we demonstrate the flexibility of CoolSim as it can operate in the presence of other microarchitecture features, such as prefetchers.
- We leverage technologies from FSA and WarmSim and propose an integrated online statistical simulation framework.

II. Background

This section introduces a number of key concepts used as a basis for this work. CoolSim enhances insights from prior work, WarmSim and FSA, to build an integrated online framework that allows for very efficient simulation sampling. To

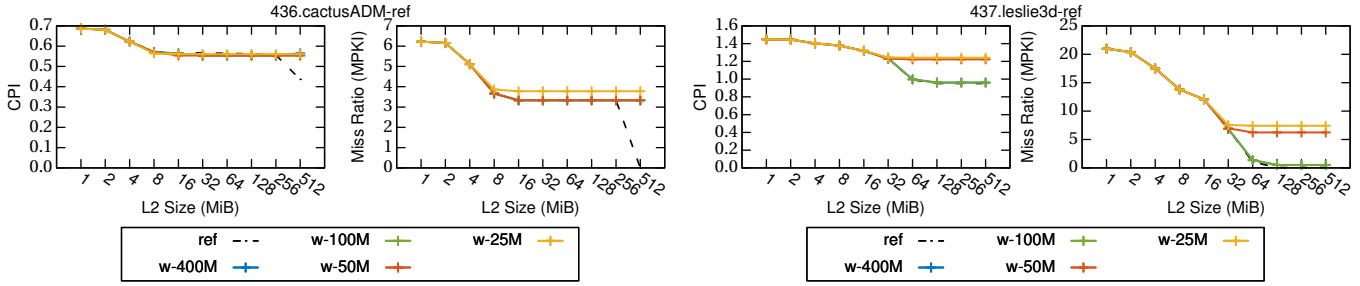


Fig. 1: Warming requirements (in instruction count) for 436.cactusADM and 437.leslie3d with the reference input set. The black dashed line shows the reference measurements, while the colors represent different simulation with different warming amounts. 437.leslie3d (right) requires at least a 100 M instructions warming (green line) to match the reference. Worse, 436.cactusADM requires more than 400 M instructions of warming to accurately determine the performance for a 512 MB cache.

achieve its efficiency CoolSim aims at: 1) fast-forwarding at near-native speeds using virtualized fast-forwarding (VFF), 2) using an enhanced statistical model that eliminates warming without the need for dense statistical profiles. CoolSim uses a novel profiler based on virtualized fast-forwarding which obtains the MRI information and a cache model that uses the sparse MRI information to estimate the timing behavior of the cache.

A. Virtualized Simulation Fast-Forwarding

Traditionally, researchers' flexibility has been limited, as they typically have to use a fixed set of benchmarks to perform experiments, and they lack the ability to re-compile or change any software parameters. The appeal of hardware-software co-design to improve performance and efficiency requires changes in the current methodologies. Previously, the cost to simulate functionally between simulation samples could be amortized through the use of checkpoints. This is not the case, however, when software changes depend on the hardware configuration. Functional simulation incurs a huge penalty as it covers most of the workload's instructions. *Full Speed Ahead (FSA)*, eliminates slow functional fast-forwarding (FF) as it leverages hardware virtualization to fast-forward simulation at near-native speeds. FSA enables hardware-software co-design as it can advance the execution to the next simulation point at near-native speed.

Virtual fast-forwarding (VFF) is implemented in gem5 and allows transparent switching to functional and detailed simulation. VFF offers a very efficient alternative to functional simulation in simulation sampling, and makes hardware-software co-design studies possible.

B. Statistical Cache Modeling Instead of Warming

Prior work, WarmSim, fast-forwards between the simulation points using functional simulation. While fast-forwarding, it also collects MRI information. As soon as it reaches a simulation point, it switches to detailed simulation to measure performance, without any prior cache warming. A statistical model determines the performance of the cache by estimating the outcome of the cache look-up of any given memory request. In this subsection, we provide details about WarmSim, focusing on the statistical model that is used to determine the cache behavior as it forms the basis of the model used in this work.

1) The Lukewarm Cache: A cache has state that we, typically, need to populate with valid data prior to a simulation point. As Figure 1 shows, cold caches can introduce large errors in performance estimations and cannot provide reliable measurements of a workload's memory behavior.

Suppose we started simulating with a cold (empty) cache; then apart from the requests that miss due to the limited capacity of the cache, requests also miss due to its insufficiently populated state. The problem that arises is that there are memory requests which miss in the cold cache when in the complete (non sampled) simulation would have hit in the warm cache. At the same time, a number of requests happen to hit in the cache. Typically spatial and short temporal reuses that fit within the short simulation window will hit. Contrary to misses, a cold cache cannot overestimate hits, i.e., there can be no extra hits in the cache due to insufficient warming. A hit happens only when the cache block is resident in the cache and that can only be if the block was referenced before within the simulation point. In other words, *the history of simulation with a cold cache is a subset of the history of simulation with a sufficiently warmed cache, and therefore the requests that hit in the cache are also a subset of the those that are estimated by the warm cache.*

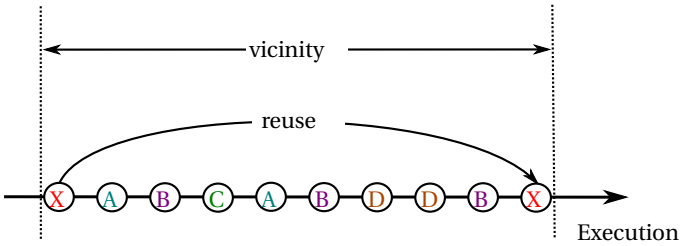


Fig. 2: A reuse in a memory access stream. The second access to X has a reuse distance of 8 as there are 8 memory accesses since the last use of X. The second access to X has a stack distance of 4 as there are 4 distinct memory blocks accessed since the last use of X. The vicinity of the reuse to X includes all the reuses of all blocks accessed between the two accessed of X.

WarmSim leverages this property. During a simulation point, memory requests that miss fetch memory blocks into the cache, and therefore its state is slowly populated. The, now lukewarm cache, i.e., a cache with partly populated state, will correctly determine short, temporal and spatial reuses as hits. A significant number of requests, e.g., reuses of the same cache block, hit in the lukewarm cache and WarmSim does not have to handle them further. On the other hand, requests that miss are further handled to determine whether they miss due to the limited capacity or due to the incomplete state of the cache.

2) Capacity misses: WarmSim lets the lukewarm cache handle requests that hit. For requests that miss, WarmSim uses statistical cache modeling to determine whether the referenced block would be resident in the warm cache. If it determines that the block would have been in the cache, it fetches it from the memory functionally with zero latency and restarts the cache look-up; otherwise it lets the cache satisfy the request that misses. WarmSim's cache modeling is based on stack distance analysis and StatStack [6] to compute the stack distances.

The *stack distance* [7] of a memory access to cache block X, as shown in Figure 2, is the number of distinct cache blocks accessed since the last access to block X. In a fully associative LRU cache, a memory request with stack distance larger than the size of the cache (in blocks) misses, otherwise it hits. The stack distance is a hardware independent metric, it can model any size cache. Knowledge of the stack distances of all the memory requests of a simulation point would eliminate the need for warming. However obtaining stack distances is an expensive process [8].

Eklov and Hagersten [6] propose StatStack, an efficient statistical method to compute stack distances using a sample of reuse distances. The reuse distance of a memory access to cache block X, is the number of memory accesses since the last use to X. Berg and Hagersten [9] show that a sample of a workload's memory reuse distances can be obtain using software only techniques with an overhead of just 40% on native execution. They also show that their sampled information allows for accurate modeling of random replacement, fully associative caches.

3) Estimating stack distances per PC: WarmSim leverages StatStack's modeling technique to estimate stack distances using reuse distances as an input. The sample of the MRI information of the workload is used to compute a histogram of the workloads stack distances. Sampling RDI means that exact information about and every memory request is not known. To overcome this limitation, WarmSim computes stack distance histograms on a per-instruction basis that can be used to compute an instruction's miss ratio. At the same time, WarmSim keeps track of the per-instruction number of hits and misses as determined by the lukewarm cache and previous requests and determines future requests such that the cache model's per-instruction miss ratio matches the currently measured miss ratio.

4) Modeling Set-Associative Caches: Stack distances allow accurate modeling of fully associative LRU caches. In set-associative caches, WarmSim uses the lukewarm cache and a stride model to determine conflict misses. When the set that a memory requests maps to is fully populated, we can safely determine that the request is a conflict miss. As the lukewarm cache is not fully populated and some sets might not contain the full state, WarmSim uses a stride model to determine unexpected worst-case access patterns that lead to conflict misses.

C. Reuse Distance Profiling

WarmSim collects dense (1 sample per 1000 memory accesses) information of the workload's memory reuse information. The authors implement a sampler on top of functional simulation, which typically runs as slow as functional simulation.

Several works propose efficient techniques to sample memory reuse distance information. Berg and Hagersten [9] demonstrate a software profiler that obtains reuse distances using performance counters on commodity hardware. They are able to sample every 10,000 to 100,000 memory instructions with an overhead of 40% compared to native execution. Sembrant et al. [10] leverage phase information to reduce the overhead to 21%.

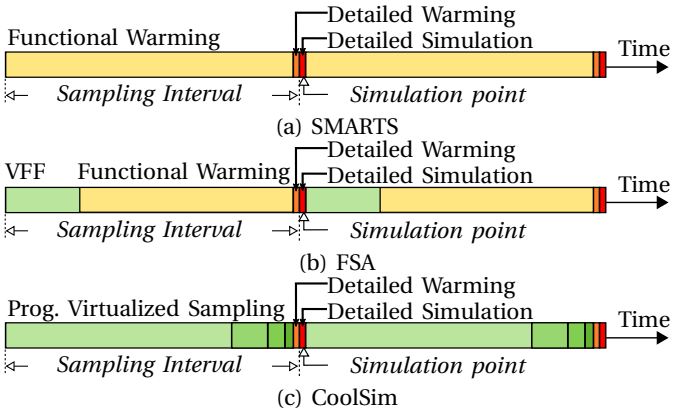


Fig. 3: A comparison of sampling methodologies

III. CoolSim

In this work, we propose CoolSim, a simulation framework that allows for fast and accurate sampled simulation. The goal of CoolSim is to eliminate cache warming, a significant source of slowdown in microarchitectural simulation. The elimination of cache warming is especially relevant for hardware-software co-design studies, as checkpoints do not always provide the flexibility needed (especially concerning software changes). CoolSim uses a statistical model based on insights from WarmSim to eliminate cache warming and determine the behavior of the cache in the simulation points. Prior to a simulation point, CoolSim runs the workload using VFF while also building upon hardware virtualization to efficiently collect the workload's MRI. The collected MRI is then used as an input to the statistical cache model.

A. Memory Reuse Information Sampling

In prior work, WarmSim, a workload's MRI is captured offline, using functional simulation. CoolSim relies on VFF to make hardware-software co-design simulation more efficient. To leverage the benefits of VFF and unleash the speed and the flexibility it offers in sampled simulation, it obtain its input, online, while the workload is fast-forwarded to the next simulation point.

CoolSim's profiler uses VFF in gem5 to collect a workload's MRI online. The MRI information required by CoolSim's statistical cache model is a sample of the workload's memory reuse distances and instruction-relative memory strides.

As the workload executes in VFF, CoolSim's profiler randomly selects memory instructions for profiling. For any selected memory instruction, the profiler records the memory reuse distance of the cache block it accesses and the memory stride of the instruction itself. Next time the same cache block is reused, the profiler records its reuse distance while next time the same instruction is executed, the profiler records the memory stride of this instruction. In this section, we describe CoolSim's profiling technology.

a) Random sampling: To obtain a small but representative sample of a workload's MRI, the profiler selects random memory instructions. The profiler uses the performance counters to stop the execution after a random number of memory instructions. To do this, it programs the hardware counters to overflow after the required number of memory instructions have been executed. The overflow triggers an interrupt that the profiler handles. To overcome, a limitation of our system's performance counters[1], the profiler uses two separate performance counters to count instructions with load or stores micro-ops separately. One counter is programmed to stop after a random number of loads and the other after a random number of stores.

As the workload is executed in a virtualized system, the performance counters are program to filter out events that come from the host system. As previous studies have shown [11], interrupts due to performance counter overflows are not precise, they suffer from skid. As the skid is likely to bias the sampling process, the profiler compensates for it by programming the performance counter overflow before the desired count of loads or stores. Once the interrupts triggers, the profiler switches the execution to functional simulation[2] to execute the remaining number of loads/stores until the desired memory instruction.

[1]Our system can reliably count instructions with a load or store micro-op but CoolSim's statistical model requires the count of load and store micro-ops

[2]KVM provides support for single-stepping on commodity hardware. However, the overhead of single-stepping using the KVM debugging API, most likely due to the high number of expensive context switches is very high.

b) Measuring Reuse Distances: Once the virtualized simulation has stopped at a randomly selected memory instruction, the profiler records the current load and store count. It then uses a watchpoint for the accessed cache block (typically a 64byte block). The watchpoint allows execution at full speed until the next access to the watched memory block. Commodity hardware does not support 64 byte watchpoints, to overcome this limitation the profiler uses the memory page protection.

The profiler's memory block watchpoint is implemented using memory page protection [12]. The memory page that contains the watched blocked is protected against reads and writes. As soon the same page is accessed again, a page fault stops the virtualized simulation. If the address is within the watched block, the profiler records the distance in memory references since the previous access to the same block. The page protection is removed and the execution is resumed. If the accessed block is not the same as the watched block the execution is resumed.

c) Measuring Instruction-Relative Strides: Once the virtualized simulation has stopped at a randomly selected memory instruction, apart from the watchpoint, the profiler also places a breakpoint on the current memory instruction. When the same instruction is executed again, a software interrupt stops the virtualized simulation and the profiler records the memory stride between the two consecutive executions of this memory instruction.

B. Cache Model

CoolSim leverages the lukewarm cache to determine the behavior of the warm cache. CoolSim takes no further actions for requests that hits in the lukewarm cache. The cache model handles memory requests that miss in the lukewarm cache to determine whether the request would hit in a sufficiently warmed cache. For such requests that would hit in the sufficiently warm cache, CoolSim fetches instantly the cache block from the memory and serves the memory request with the same latency as a cache hit.

1) Non-Blocking Caches: Modern caches can serve multiple outstanding memory requests. To do this, the *miss status holding registers (MSHRs)*, keep track of requests that are outstanding and are waiting to be satisfied. When a request misses in the cache, we search the MSHRs to find other outstanding requests for the same cache block. If such an MSHR exists, the cache block has been requested already and the new request get services as soon as the block is fetched. Such requests are typically classified as *delayed hits*.

When a request misses in the lukewarm cache, CoolSim first searches in the MSHRs. If the look-up for a matching MSHR is successful, CoolSim takes no further action, as it has determined for a previous request to the same block which initiated the MSHR that the cache block is not resident in the warm cache.

2) Statistical Cache Model: A large number of requests hit either in the lukewarm cache or the MSHRs. A request look-up, however, that misses both in the lukewarm cache and in the MSHRs needs to be handled by the statistical cache model. StatStack is used to compute the miss ratio of the instruction that sent the request. This miss ratio is compared with the current miss ratio as computed by the per-instruction counters that CoolSim maintains. *a*) If the miss ratio of the counters is lower than the one computed by StatStack then the request is accounted for as a miss. The miss counter for this instruction is incremented and the request is handled as a normal cache miss; *b*) Otherwise the request is accounted for as a hit. The hit counter for this instruction is incremented and a functional request to the next level in the memory hierarchy instantly fetches the memory block. The request is handled as a normal cache hit.

Prefetching: Modern systems employ prefetching as one of several latency hiding techniques. For most prefetchers, the training period is rather small. Some prefetchers do not cross page boundaries, while others trade performance at the cost of bandwidth, i.e., prefetching accuracy. CoolSim does not warm the prefetcher, however in the presence of prefetching, it counts the first access to a block that has been prefetched as a miss.

CoolSim uses StatStack to compute the per-instruction miss ratio. While WarmSim captures a fairly dense sample of a workload's MRI (1 in 1000 memory instructions), CoolSim needs to be accurate with sparser MRI, to keep performance high. The overhead of switching between simulation modes can be rather high and there would be no performance benefit from VFF.

StatStack computes the stack distance of a memory access to X using its reuse distance and the reuse distance histogram of the vicinity of the reuse to X. Given CoolSim's sparsely-sampled MRI information, the vicinity histograms need to be carefully selected to be representative. CoolSim forms a set of backwards growing vicinities which all end at the current simulation point. Any given vicinity needs to be large enough to avoid statistical errors, while not being too large as to blend phases, changing the average characteristics of this region. In practice, the smallest vicinity will have at least 50 samples, and its size will not be more than 20% larger than the reuse distance.

C. Progressive Sampling and Batch Sampling

Small vicinities need to be statistically stable, i.e., have a large enough number of samples, while bigger vicinities due their size will always contain enough samples. At the same time, the sparse sampling rates result in higher

Pipeline	gem5's default OoO CPU	
	Store Queue	64 entries
	Load Queue	64 entries
Branch Predictors	Tournament Predictor	2-bit choice counters, 8 k entries
	Local Predictor	2-bit counters, 2 k entries
	Global Predictor	2-bit counters, 8 k entries
	Branch Target Buffer	4 k entries
Caches	L1I	64 kB, 2-way LRU
	L1D	64 kB, 2-way LRU
	L2	1 MB to 512 MB, 8-way LRU

TABLE I: Summary of simulation parameters.

performance. CoolSim's profiler uses a progressively increasing sampling rate to benefit from the higher performance of sparse sampling far from the simulation point and the dense sampling rate close to the simulation point that allows statistically stable small vicinities. To ensure that all samples are represented according to their sampling probability they are weighted accordingly. The VFF profiling is split in 3 parts Figure 3c): *a*) the first part, 75% of virtualized fast-forwarding phase is sampled with a low rate; *b*) the second part, 75% of the remaining, with an average sampling rate; *c*) and the last part with a high rate.

Further, to keep the number of expensive simulation switches low compared to the sampling rate, the profiler uses batch sampling, i.e., selects a number of consecutive memory accesses with the same random mechanism described earlier.

IV. Evaluation

In this section we look into the accuracy and the performance of CoolSim. Our experiments simulate a 64 bit x86 system (Ubuntu 12.04.5 LTS with Linux 3.2.44) with split 2-way 64 kB L1 instruction and data caches and a unified 8-way L2 cache with sizes ranging from 1 MB to 512 MB. We use gem5's OoO CPU model. A summary of the important simulation parameters can be found in Table I. All benchmarks are compiled with GCC 4.6.3 in 64 bit mode. We use the SPEC CPU2006 benchmark[3] suite with the reference data set. All simulation runs were started from the same checkpoint of a booted system that has already executed 100 B instructions. Simulation execution rates are shown running on an 8-core 2.3 GHz Intel Xeon E5520.

We run 10 B instructions in functional warming to ensure that the caches are warm. We evaluate the accuracy of our simulation framework in the next 10 B instructions, using 10 uniformly distributed simulation samples. The slow reference experiments in combination with the wide range of L2 sizes we evaluate does not allow us to run the benchmarks to completion, but allows us to build a sample with a region suitable for comparison. Before each simulation point 30 k instructions are executed in detail to warm the microarchitectural state (ROB, Load/Store Queues, etc.) and 20 k instructions in detail where we perform the desired measurements. [2] report that 30 k instruction are enough to warm the microarchitectural state. Between the simulation points we uses functional simulation and which continuously populates the caches.

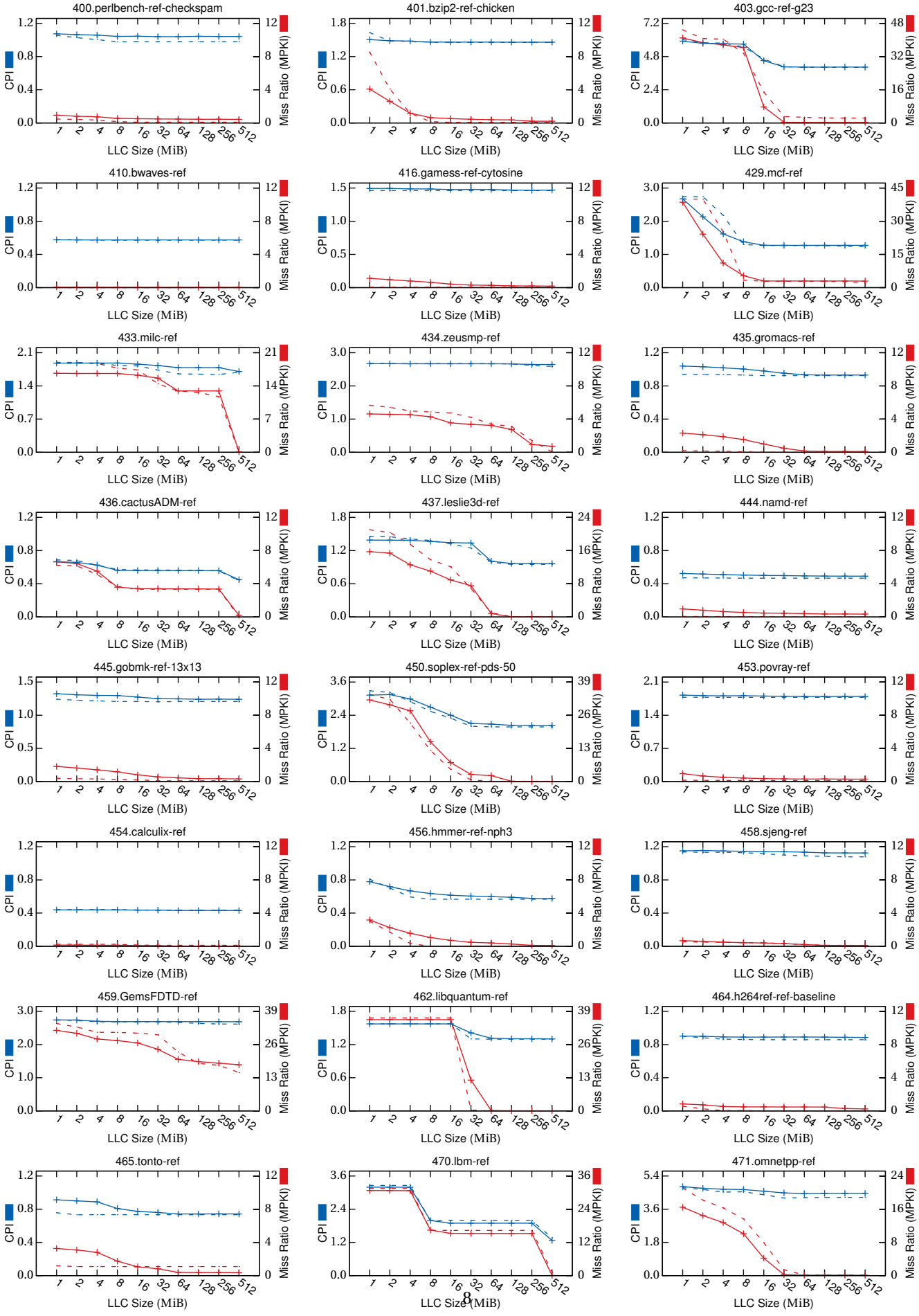
Experiments using CoolSim use a similar setup, but instead of functional warming, we execute in VFF with MRI sampling. The profiler uses progressive sampling rate. In the first 10 B instructions, it randomly selects 1 in $1e7$ memory instructions and it samples 50 memory instructions in a batch, these samples are assigned a weight of 10. Then before each simulation point, it executes 750 M instructions where it selects 1 in 1×10^6 instructions, and it samples 25 memory instructions in a batch, weighing them by 4; in the next 200 M instructions it selects 1 in 1×10^6, taking a batch of 50 samples, weighing them by 2; in the last 50 M instructions it selects 1 in 1×10^{-5}, taking a batch of 10 samples, weighting them by 1.

A. Accuracy

To evaluate the accuracy of CoolSim we compare the results from simulations with a variety of L2 cache sizes. We start with experiments without any prefetching. Prefetchers are typically trained within short periods of execution and are likely to hide CoolSim's errors in modeling the cache performance.

Figure 3 shows how the CPI and the miss ratio estimations compare with the reference for the 27 benchmarks. 473.astar-ref-BigLakes2048 shows the largest average error of 9.7%. Looking closer at its simulations we see that the largest part of its error comes from the L1D miss ratio, where CoolSim for many instructions fails to track it accurately. The L1D is only 64 kB where statistical modeling is more challenging due to the small size. 429.mcf-ref shows the

[3]We were unable to run 447.dealII and 481.wrf; both benchmarks produced outputs that could not be verified with the reference outputs provided by the benchmark suite and therefore we did not use them in our evaluation.


400.perlbench-ref-checkspam
401.bzip2-ref-chicken
403.gcc-ref-g23
410.bwaves-ref
416.gamess-ref-cytosine
429.mcf-ref
433.milc-ref
434.zeusmp-ref
435.gromacs-ref
436.cactusADM-ref
437.leslie3d-ref
444.namd-ref
445.gobmk-ref-13x13
450.soplex-ref-pds-50
453.povray-ref
454.calculix-ref
456.hmmer-ref-nph3
458.sjeng-ref
459.GemsFDTD-ref
462.libquantum-ref
464.h264ref-ref-baseline
465.tonto-ref
470.lbm-ref
471.omnetpp-ref
CPI
Miss Ratio (MPKI)
LLC Size (MiB)
1 2 4 8 16 32 64 128 256 512

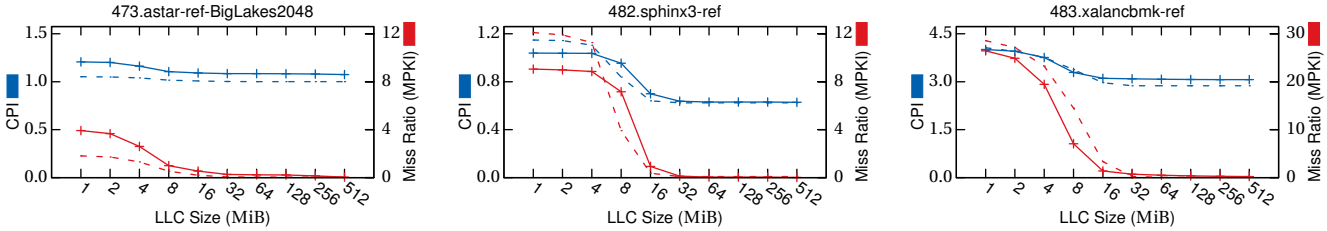


Fig. 3: CPI and LLC miss ratio measured for the SPEC CPU2006 benchmarks for LLC sizes from 1MB to 512MB. The reference is shown with dashed lines, while CoolSim simulations are shown with solid lines.

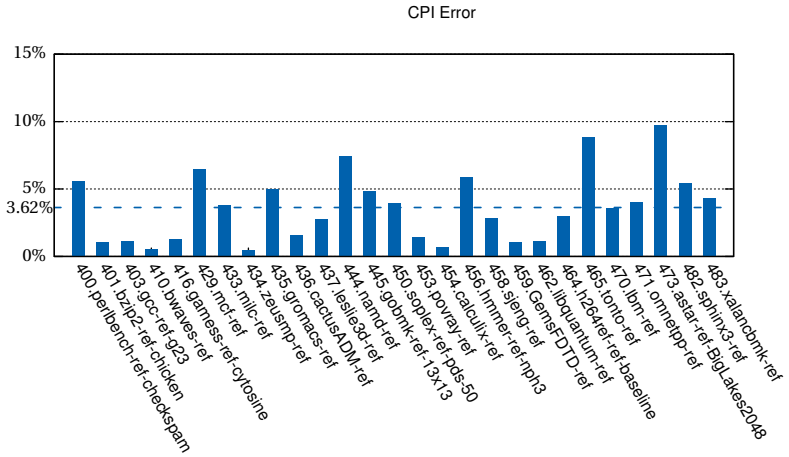


Fig. 4: SPEC CPU2006: Average CPI error across the 27 SPEC CPU2006 benchmarks, and across cache sizes from 1 MB to 512 MB.

maximum error. For an L2 cache of 4 MB shows an error of 25.7%, however for the most sizes and especially for caches larger than 8 MB both the miss ratio and the CPI estimations are accurate.

1) Prefetching: We also evaluate CoolSim in the presence of a stride prefetcher. The stride prefetcher fetches cache blocks in the L2 cache and trains on every memory request to the L2 cache. Figure 5 shows 3 benchmarks where prefetching improves significantly the performance and the miss ratio. 403.gcc-ref-g32 improves by almost a constant factor through the range of the evaluated cache sizes, while the performance of 462.libquantum-ref improves only for small cache sizes as beyond 32MB its working set size fits in the cache.

CoolSim predicts accurately both the miss ratio and the CPI for these 3 benchmarks in the presence of stride prefetchers. It models the performance of the SPEC CPU2006 benchmarks with an average error of 4.77%. It's worth also mentioning that CoolSim uses the same MRI input for the experiments with and without the L2 prefetchers.

B. Performance

In our framework, CoolSim's average performance is 17.23 MIPS, a 19x speedup over the reference which simulates at 0.89 MIPS. While detailed warming and simulation perform the same both in CoolSim and the reference (around 0.1 MIPS), the contribution is small as the total number of instructions simulated is negligible (less than 0.001% of the dynamic instruction count).

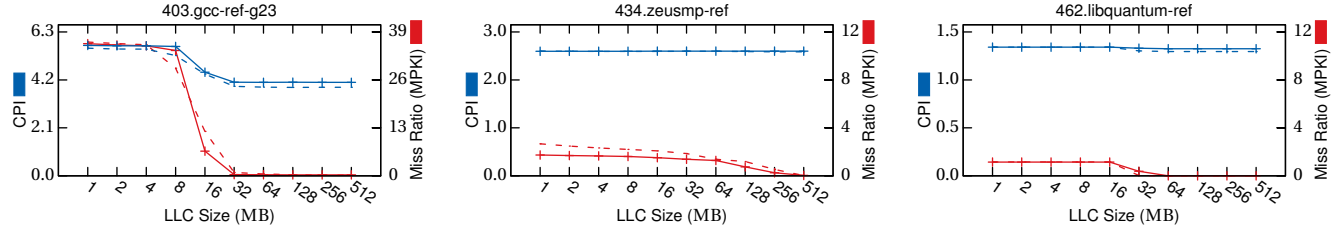


Fig. 5: CPI and LLC miss ratio measured for the SPEC CPU2006 benchmarks for LLC sizes from 1MB to 512MB with prefetching. The reference is shown with dashed lines, while CoolSim simulations are shown with solid lines.

V. Related Work

The problem of warming has been the focus of much work since the introduction of simulation sampling. Wenisch et al. [13] store the state of the caches and other microarchitectural structures in customized checkpoints, the *Flex points*. This way, they reduce the overall simulation overhead by eliminating warming. In a follow-up work [14] generate *Live points* which reduces the space requirement for each checkpoint from 20−100MB required for a Flex point down to 142KB. Barr et al. [15] propose the Memory Timestamp Record (MTR), a method that records memory patterns and then compresses and stores them in checkpoints. CoolSim, in contrast, requires the collection of a workload's MRI. The MRI can be obtained faster using VFF. Additionally, the MRI allows for more flexible simulations as it can be saved and restored to simulate systems with any size caches without any warming.

Other work focuses on minimizing the amount of warming needed to produce accurate results. Haskins and Skadron [16] and Luo et al. [17] use heuristics to find the minimum number of instructions needed to warm a cache. Burugula and Skadron [18] introduce the concept of *Memory Reference Reuse Latencies (MRRLs)* which is the number of completed instructions between consecutive references to the same memory location. The number of instructions with a large enough cumulative distribution of MRRLs is used as a sufficient warming interval. Eeckhout et al. [19] introduce the concept of *Boundary Line Reuse Latency (BLRL)* which extends the concept of MRRLs. They apply similar heuristics to find the optimal warming period. Van Ertvelde et al. [20] extend on the concept of BLRL using a form of hardware state check points. Sandberg et al. [4] use limited warming of 5M and 25M instructions to warm a 2 MB and 8 MB cache. They propose a method that uses 2 parallel simulations, a pessimistic and an optimistic one, to bound the maximum error due to warming. While minimizing warming improves simulation efficiency, this minimum amount of warming needed for very large caches still dominates the simulation time for traditional methodologies.

Beyls and D'Hollander [8] report that the collection of stack distances has a runtime overhead of 1000x. Liu and Mellor-Crummey [21] propose a technique based on shadow profiling that forks a redundant copy of an application, instrumented by Pin, to measure the stack distances for a selected set of references [21]. The run time overhead is reported to be as low as 13%. To reduce their overhead they only measure stack distances for memory references that are likely to expose locality problems. CoolSim relies of VFF to be able to switch to detailed simulation; the collection of the MRI information with its profiler has a high penalty due to the slow context switches for the virtualized simulated machine to the gem5 application on the host. Adding support, however, for parallel profiling of a workload's different regions is an interesting extension and could improve CoolSim's performance.

Other approaches raise the level of abstraction of the simulation. Carlson et al. [22] propose an efficient simulation framework leveraging the insights of interval simulation [23]. Such approaches are orthogonal to CoolSim and they can improve the performance of detailed simulation.

VI. Conclusion

Unbiased measurements, even of moderately sized caches, require long functional warming which can dominate the overhead of sampled simulation methodologies. CoolSim addresses this cache warming problem with fast, virtualized fast-forwarding and statistical cache modeling to eliminate traditional warming completely. The result is a flexible and efficient solution in the presence of cache hierarchy changes, large LRU caches, or software changes commonly seen in hardware-software co-design. Simple extensions allow for modeling of different replacement policies such as random [24] or bit-pLRU [25].

CoolSim obtains its input data using a fast profiling framework that is built on top of gem5's hardware virtualization fast-forwarding mode. This data is used by a statistical model that allows one to accurately determine cache miss rates and, using detailed simulation, workload CPI.

We have evaluated CoolSim with the SPEC CPU2006 benchmark suite, and have demonstrated high accuracy across a wide range of cache sizes (1−512MB). When bringing together the fast, hardware-based sampler with the detailed enhancements for improved accuracy, CoolSim estimates an application's CPI with an average error of 3.62% while being able to simulate 19x faster than traditional SMARTS simulations that make use of functional warming.

References

[1] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, and T. Krishna, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011.

[2] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2003.

[3] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[4] A. Sandberg, N. Nikoleris, T. E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer, "Full speed ahead: Detailed architectural simulation at near-native speed," in *Proc. International Symposium on Workload Characterization (IISWC)*, 2015.

[5] N. Nikoleris, D. Eklov, and E. Hagersten, "Extending statistical cache models to support detailed pipeline simulators," in *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2014.

[6] D. Eklov and E. Hagersten, "Statstack: Efficient modeling of lru caches," in *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2010.

[7] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, 1970.

[8] K. Beyls and E. D'Hollander, "Discovery of locality-lmproving refactorings by reuse path analysis," in *Proc. International Conference on High Performance Computing and Communications (HPCC)*, 2006.

[9] E. Berg and E. Hagersten, "Statcache: A probabilistic approach to efficient and accurate data locality analysis," in *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2004.

[10] A. Sembrant, D. Black-Schaffer, and E. Hagersten, "Phase guided profiling for fast cache modeling," in *Proc. International Symposium on Code Generation and Optimization (CGO)*, 2012.

[11] D. Chen, N. Vachharajani, R. Hundt, X. Li, S. Eranian, W. Chen, and W. Zheng, "Taming hardware event samples for precise and versatile feedback directed optimizations," *IEEE Transactions on Computers*, vol. 62, no. 2, 2013.

[12] B. Beander, "Vax debug: An interactive, symbolic, multilingual debugger," in *Proc. Symposium on High-level Debugging*, 1983.

[13] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe, "Turbosmarts: Accurate microarchitecture simulation sampling in minutes," *ACM SIGMETRICS Performance Evaluation Review*, vol. 33, no. 1, 2005.

[14] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "Simflex: Statistical sampling of computer system simulation," *IEEE Micro special issue on Computer Architecture Simulation*, vol. 26, no. 4, 2006.

[15] K. C. Barr, H. Pan, M. Zhang, and K. Asanovic, "Accelerating multiprocessor simulation with a memory timestamp record," in *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2005.

[16] J. Haskins, J.W. and K. Skadron, "Minimal subset evaluation: Rapid warm-up for simulated hardware state," in *Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*, 2001.

[17] Y. Luo, L. K. John, and L. Eeckhout, "Self-monitored adaptive cache warm-up for microprocessor simulation," in *Proc. Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2004.

[18] R. S. Burugula and K. Skadron, "Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation," in *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2003.

[19] L. Eeckhout, Y. Luo, K. De Bosschere, and L. K. John, "Pblrl: Accurate and efficient warmup for sampled processor simulation," *Comput. J.*, vol. 48, no. 4, 2005.

[20] L. Van Ertvelde, F. Hellebaut, L. Eeckhout, and K. De Bosschere, "Nsl-blrl: Efficient cachewarmup for sampled processor simulation," in *Proceedings of the 39th annual Symposium on Simulation*, 2006.

[21] X. Liu and J. Mellor-Crummey, "Pinpointing data locality bottlenecks with low overhead," in *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2013.

[22] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *Proc. High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[23] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2004.

[24] E. Berg and E. Hagersten, "Fast data-locality profiling of native execution," in *Proc. Internationcal Conference on Measuring and Modeling of Computer Systems (SIGMETRICS)*, 2005.

[25] X. Pan and B. Jonsson, "A modeling framework for reuse distance-based estimation of cache performance," in *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2015.