Delorean: Virtualized Directed Profiling for Cache Modeling in Sampled Simulation

Nikos Nikoleris
Arm Research, Cambridge UK
Email: nikos.nikoleris@arm.com

Erik Hagersten
Department of Information Technology
Uppsala University, Uppsala, Sweden
Email: erik.hagersten@it.uu.se

Trevor E. Carlson
Department of Computer Science
National University of Singapore
Email: tcarlson@comp.nus.edu.sg

Abstract

Current practice for accurate and efficient simulation (e.g., SMARTS and Simpoint) makes use of sampling to significantly reduce the time needed to evaluate new research ideas. By evaluating a small but representative portion of the original application, sampling can allow for both fast and accurate performance analysis. However, as cache sizes of modern architectures grow, simulation time is dominated by warming microarchitectural state and not by detailed simulation, *reducing overall simulation efficiency*. While checkpoints can significantly reduce cache warming, improving efficiency, they limit the flexibility of the system under evaluation, requiring new checkpoints for software updates (such as changes to the compiler and compiler flags) and many types of hardware modifications. An ideal solution would allow for accurate cache modeling for each simulation run without the need to generate rigid checkpointing data a priori.

Enabling this new direction for fast and flexible simulation requires a combination of (1) a methodology that allows for hardware and software flexibility and (2) the ability to quickly and accurately model arbitrarily-sized caches. Current approaches that rely on checkpointing or statistical cache modeling require rigid, up-front state to be collected which needs to be amortized over a large number of simulation runs. These earlier methodologies are insufficient for our goals for improved flexibility. In contrast, our proposed methodology, Delorean, outlines a unique solution to this problem. The Delorean simulation methodology enables both flexibility and accuracy by quickly generating a targeted cache model for the next detailed region *on the fly* without the need for up-front simulation or modeling. More specifically, we propose a new, more accurate statistical cache modeling method that takes advantage of hardware virtualization to precisely determine the memory regions accessed and to minimize the time needed for data collection while maintaining accuracy.

Delorean uses a multi-pass approach to understand the memory regions accessed by the next, *upcoming* detailed region. Our methodology collects the entire set of *key memory accesses* and, through fast virtualization techniques, progressively scans larger, earlier regions to learn more about these key accesses in an efficient way. Using these techniques, we demonstrate that Delorean allows for the fast evaluation of systems and their software though the generation of accurate cache models on the fly. Delorean outperforms previous proposals by an order of magnitude, with a simulation speed of 150 MIPS and a similar average CPI error (below 4%)

I. Introduction

With the goal of improving system performance and efficiency, researchers use simulation to evaluate their enhancements without the need to build physical prototypes. Detailed simulation, however, tends to be many orders of magnitude slower than execution in real hardware. A popular detailed simulator, gem5 [1], executes at a rate of 100 kIPS [2], which is about 5 orders of magnitude slower than native execution. At this rate, even a relatively small application that would normally execute in just a few minutes on today's hardware would require days or months to simulate.

Researchers, typically, address this challenge using sampled simulation [3, 4], which simulates representative regions to closely approximate the performance of the original application. While the promise of fast and accurate simulation is now in hand, two major issues need to be addressed: 1) the ability to quickly locate the representative region for simulation (find the appropriate *architectural state*), and 2) the ability to warm-up large, performance-critical structures (use the appropriate *microarchitectural state*).

The cost of determining the appropriate architectural and microarchitectural state can be amortized through the use of checkpoints. By checkpointing the state in advance, the evaluation at the simulation points can readily proceed without the need to warm up the state. Unfortunately, the use of checkpoints, while efficient, limits the opportunities for software and hardware flexibility of the simulated system. Software changes (e.g., JIT compilation strategies for managed languages, updating compiler, or changing compiler options), or hardware changes (e.g., cache size) can result in non-representative, unusable checkpoints.

To efficiently determine the appropriate architectural state, Full Speed Ahead (FSA) [2] uses hardware virtualization to quickly advance to the next simulation point. FSA demonstrates low-overhead sampled simulation with an average

Fig. 1: The coverage of the working set as a function of the simulated interval. Warming the complete working set of 470.lbm (required for a large DRAM cache) requires almost 1 B instructions.

simulation speed of almost 500 MIPS. While close to native speed, FSA works best for small cache sizes, 2 MiB in this example, as larger cache sizes incur significant slowdowns due to cache warming (simulating an 8 MiB cache is more than 5× slower). We can therefore expect a significant slowdown for applications with even larger warming requirements (470.lbm-ref in Figure 1 (left) needs almost 1 B instructions, a 40× increase, to guarantee a sufficiently warm DRAM cache that fits its working set).

To overcome these limitations, we propose Delorean, a framework that provides both *ultra-fast* and *flexible* sampled simulation. To achieve this, Delorean leverages the speed of VFF and uses statistical cache modeling to eliminate the need for cache warming. A simplified explanation of the overall methodology can be described as follows:

Phase A: The simulator state is quickly checkpointed (using copy-on-write techniques) and VFF is used to advance execution to the next simulation point. Detailed simulation is then used to record the small number of cachelines accessed during that simulation point. These are the key cachelines.

Phase B: The execution is then restarted at the checkpoint to record the last use of each key cacheline just before the simulation point. Our method, Virtualized Direct Profiling (VDP), efficiently collects the exact reuse distance for each key cacheline.

Phase C: Finally, at the simulation point, we use detailed simulation and the exact reuse distance for each of its memory accesses as obtained in **Phase B** to estimate its cache behavior using a state-of-the-art statistical cache modeling technique.

This paper makes contributions in the following areas:

- We suggest using a combined virtualized execution/simulation framework to allow for efficient jumps backwards and forwards in time.
- We propose using this time traveling technique to identify the cachelines to be accessed by the next simulation point.
- We use our multi-phased profiling technique to eliminate the need for warming of large caches.
- We describe the implementation of Delorean, our multi-phase simulation tool built on KVM and gem5. Delorean relies on time travel to implement an efficient sampled simulation framework capable of simulating huge caches. Delorean simulates the SPEC CPU2006 benchmarks at a rate of 150 MIPS, which is an order of magnitude faster than previous approaches while maintaining the same accuracy. We plan to make the source code and scripts of our simulation framework publicly available.

II. Background

Delorean is a sampled simulation framework that uses cache modeling to eliminate cache warming. Its cache model uses data obtained through profiling. In this section, we introduce a number of key concepts which are used as a basis for this work.

A. Sampled Simulation and Virtualized Simulation

In sampled simulation (e.g., SMARTS [3]), a small number of instructions are simulated in detail while the vast majority of the application is fast-forwarded or skipped. Typically, the architectural state of the application prior to a simulation point is checkpointed and later restored when evaluating the simulation points. This effectively avoids the costly overhead of warming the microarchitectural state. However, this also requires that the simulated system uses the same inputs, binaries and operating system as the checkpointed system. Furthermore, no software updates or changes (updated compilers, JIT or co-designed software and hardware, etc.) are possible and even some microarchitectural changes are not possible.

To tackle the problem Sandberg et al. [2] propose Full Speed Ahead (FSA) as an alternative to checkpoints. They use Virtualized-Fast Forwarding (VFF) which uses hardware virtualization to quickly fast-forward to the next simulation point. FSA provides fast simulation without the need for checkpoints as advancing happens in near-native speed. At the same time, it offers complete flexibility to make changes between two different runs. FSA, however, uses simulation to warm up the caches. As result, it does not address the problem of warming arbitrarily sized caches in a high-performance way. Any performance gains from VFF are entirely lost for large applications running on systems with large caches (e.g., DRAM caches).

B. Statistical Cache Modeling

Statistical cache models rely on rigorous statistics to estimate several key performance metrics. In this paper, we use statistical cache modeling based on the concept of *stack distance* to address the problem of cache warming. Using stack distance analysis, we model LRU caches. Similar metrics can be used for caches with other replacement policies such as random replacement [5] or pLRU and NRU [6].

We use the term *s*tack distance d_s [7] of an access to a cacheline X to refer to the number of distinct cachelines accessed since the last access to X. The stack distance allows for accurate modeling of fully associative, LRU caches. If d_S is larger than the size of the cache in cachelines s then the memory access misses in the cache; otherwise the memory access hits. The epoch from the last access to cacheline X to the current access to cacheline X is referred to as the reuse epoch of X or the vicinity of X.

1) Stack Distance Estimation: Obtaining exact stack distances is a costly operation; to measure the stack distance we need to inspect every memory access between the two consecutive accesses to cacheline X. Instead, Eklov and Hagersten [8] show that the stack distance can be accurately estimated using its corresponding *reuse distance*, which is defined as the number of memory accesses (not *distinct*) since the last access to X. Delorean uses StatStack to determine capacity misses.

2) Reuse Distance Collection: The reuse distance of an access to a cacheline X is the number of memory accesses since the last access to the cacheline X. To obtain the reuse distance, we just need to know the number of memory accesses between the two accesses to X, as opposed to the stack distance that we need to know the addresses of all these accesses.

Berg and Hagersten [5] propose a native, random reuse distance profiler. They use performance counter overflows to pick random memory accesses and cacheline watchpoints to detect reuses. For every detected reuse, they record the corresponding reuse distance using performance counters. As commodity hardware does not provide support for watchpoints on a cacheline granularity, they use the operating system's page protection mechanism to implement cacheline watchpoints. Execution between watchpoints runs at native speed, and watchpoints stop the execution when there is an access to the protected page. Berg and Hagersten [5] show that their sampler can obtain a sparse but representative sample of reuse distances to accurately estimate an application's miss ratio, adding 40% overhead on native execution. Their method, however, is accurate at estimating average cache miss ratio, while Delorean needs to determine if specific memory accesses will miss or hit in the cache.

C. CoolSim: Statistical Cache Modeling in Sampled Simulation

Nikoleris et al. [9] use statistical cache modeling to eliminate cache warming in sampled simulations. Their proposal, *CoolSim* (Figure 2), collects memory reuse information (MRI), consisting of randomly and sparsely collected reuse distances, while using VFF to advance the execution to the next simulation point.

At the simulation point, they switch to detailed simulation and feed the collected MRI to a statistical cache model to determine whether the executed memory accesses hit or miss in the cache. Since the obtained MRI is sparsely sampled, it rarely contains the exact reuse information for the memory accesses in the simulation point. Instead, per-PC reuse distribution is obtained from the MRI, with the hope that each instructions executed during the simulation point will have a large reuse population recorded in MRI.

CoolSim's profiler uses software watchpoints and performance counters to obtain MRI, while fast-forwarding the execution to the next simulation point using hardware virtualization. To achieve high accuracy, the obtained MRI needs to contain information about every memory instruction (PC) in the next simulation point. As its profiler is random and cannot target specific memory instructions, CoolSim relies on rather high sampling rates to achieve the necessary per-PC information. While fast-forwarding using hardware virtualization can be done in near-native speed, their MRI profiler adds substantial overhead mainly due to the density of the required MRI. CoolSim can simulate systems with caches of any size at a speed of 17 MIPS, two orders of magnitude slower than native execution. Further, as the obtained MRI is a dependent on the evaluated software, rather than on the evaluated hardware, the profiling

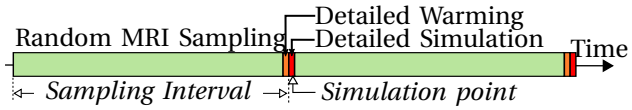

Fig. 2: CoolSim obtains a random sample of MRI while advancing to the next simulation point in a simple pass. Delorean, on the other hand, uses virtualized directed profiling while advancing to the next simulation point in multiple passes.

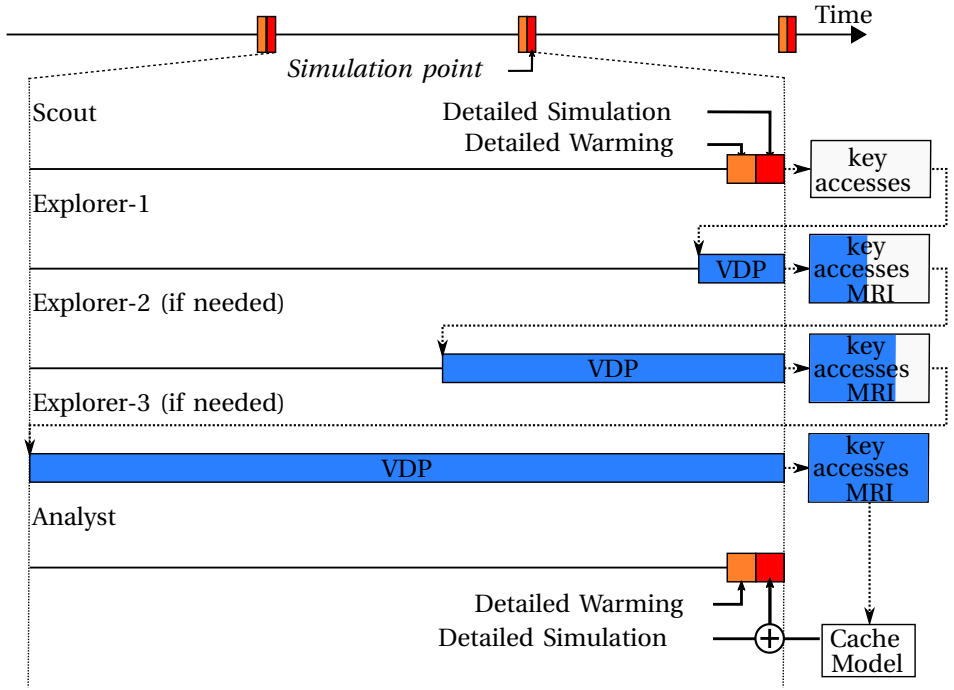

Fig. 3: Delorean uses multiple passes to quickly and accurately measure performance. First, the Scout advances to the next simulation point, where it identifies the key memory accesses. Then, the Explorers collect memory reuses for key memory accesses. In the last pass, the Analyst uses the collected MRI with statistical cache modeling and detailed simulation to evaluate the performance at the simulation point.

cost can be amortized without sacrificing flexibility to make changes in the simulated hardware. The collected MRI allows flexibility for changes in the hardware of the simulated system, including changes of the size of the caches.

In order to lower the overhead for MRI collection, a more directed method for collecting MRI is needed. Ideally, we would like to reduce the MRI sampling rate (lower overhead) while, at the same time, capturing the exact reuse for every memory access of the simulation point, rather than relying on the per-PC reuse distribution (improved accuracy).

III. Delorean

Delorean is similar to CoolSim in that, 1) prior to a simulation point, it collects MRI while advancing the execution using virtualization (VFF) and 2) at the simulation point, it estimates the cache behaviour using the collected MRI while performing detailed simulation (gem5). However, Delorean uses directed profiling to select which reuse distances to collect in a more intelligent way than CoolSim. This reduces the MRI sampling rate compared to CoolSim. In this section, first, we describe Delorean's directed MRI profiler and then, its cache model.

A. Directed MRI Profiling

Delorean's directed profiler first identifies the small set of cachelines (*key cachelines*) that are accessed during the next simulation point (*key accesses*) and for each key cacheline, records the distance since it was last accessed (*before* the simulation point). This way, it first selects the memory accesses to monitor and then measures its "backwards" reuse distance, i.e., the nearest previous access to the same cacheline. We refer to these reuses as key reuse distances.

In addition to the key reuse distances, we also obtain a sparse representative set of reuses near the simulation point, *the vicinity reuse distances*. This scheme has two advantages: 1) The exact reuse distance for *all* memory accesses at the simulation point is known (better accuracy), and 2) fewer ruse distances need to be collected (since the per-PC distribution no longer is needed). Delorean achieves the same accuracy as CoolSim, while at same time, it lowers the MRI sample rate requirements by an order of magnitude.

To collect its MRI, Delorean uses multiple passes. As Figure 3 illustrates, each of the passes uses a separate instance (process) of the same simulation, i.e., checkpointed state of the virtualized execution is used to rewind the execution of each phase to the starting point.

In the first pass, the **Scout** identifies the key accesses. It first advances the execution to the next simulation point using VFF at near-native speed. At the simulation point, it switches to detailed simulation and records the key cachelines. For each of the key cacheline, the Scout also records the program counter (PC), the memory address, the size of the access and the current instruction count, i.e., information sufficient for subsequent passes to identify the exact same memory access. To find the key accesses the Scout uses the lukewarm cache and the MSHRs, techniques which are explained in subsection III-B in more detail. As we show later the number of key accesses is rather small. For the SPEC 2006 benchmarks and for a simulation point of 10,000 instructions, the average number of key accesses is 144.

In subsequent passes, **The Explorers**, which are separate simulation instances (three Explorers are shown in Figure 3) implement Virtualized Directed Profiling (VDP) to collect MRI. Each of the Explorers restores from the same checkpoint and obtains key reuse distances for the key accesses and a few randomly chosen vicinity reuse distances.

The vicinity reuses are recorded similarly to previous reuse profilers [5, 9], i.e., a sparse set of randomly chosen memory accesses are selected and their next (forward) reuse is recorded using watchpoints. Once a reuse within the vicinity has been recorded, the corresponding watchpoint is removed and its reuse distance recorded.

For each key cacheline recorded by the Scout, the Explorers are looking for its last access prior to the simulation point. To do this, the Explorers set watchpoints on the key cachelines. As the number of key cachelines for a simulation point is relatively small, it may appear that the task of measuring their (backwards) reuse distance is fairly trivial. However, to find the *last* access to the cacheline before the simulation point, the watchpoint needs to remain active until the end of the simulation point. As a result, a single watchpoint might trigger 1000s of times and each time it will interrupt the execution. The interrupts can quickly dominate performance and diminish the benefits of the virtualized fast-forwarding. To overcome this limitation, we use multiple Explorers, as shown in Figure 3.

Explorer-1 fast-forwards using VFF and switches to VDP, 5 M instructions before the simulation point. As soon as it reaches the end of the simulation point, it feeds the obtained MRI along with the key accesses that have not been profiled already – because their reuses are larger than 5 M instructions – to another instance, Explorer-2.

Explorer-2 switches to VDP 5 M instructions before the simulation point. Its task is to find the reuses for the key cachelines that were outside of the reach of Explorer-1 (55% reduction). The subset of undiscovered reuses is not only significantly smaller than the full set of key cachelines, but also tends to contain cachelines with lower temporal locality. As a result, watchpoints for these do not trigger as often. Key cachelines that had reuses outside of the reach of Explorer-2 (if any) are fed to **Explorer-3**. The same process is repeated until the whole set of key cachelines is covered. A small number of Explorers 4 is very effective at reducing the number of triggered watchpoint. The process, however, can stop before all 4 Explorers have finished as soon as the whole set of key reuses has been obtained.

This multi-pass approach allows Delorean to 1) profile only as far back as it is needed, 2) without profiling much further back than what is needed and 3) without profiling the same information twice. As a result, the number of times a watchpoint triggers is reduced significantly and Delorean can leverage the speed of hardware virtualization, to obtain MRI one order of magnitude faster than previous approaches.

Finally, the last pass, Explorer-n, feeds the obtained MRI to yet another simulation instance, the **Analyst**. The Analyst uses detailed simulation and cache modeling with the obtained MRI to evaluate the simulation point without any prior cache warming.

Note that as soon as each pass finishes with the current simulation point, it can move on to the next simulation point. This way we can pipeline the process as long as we have enough cores to run the Scout, the Explorers and the Analyst.

B. Statistical Cache Modeling

Cache requests normally miss either due the limited capacity, i.e., capacity misses, or the limited associativity, i.e., conflict misses. The Analyst simulates without the need to warm up the cache, anqd handles requests that miss

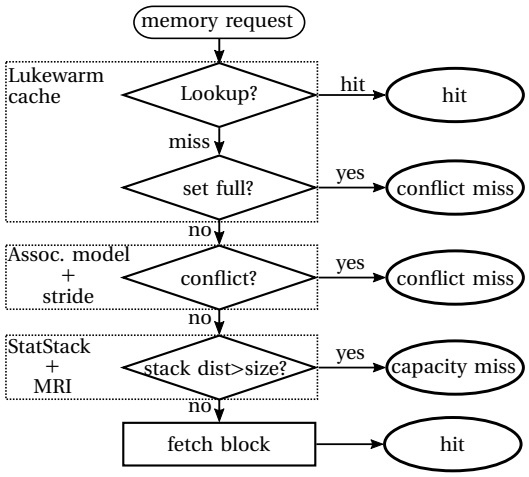

Fig. 4: Delorean's statistical cache model. The lukewarm cache readily determines accesses with short reuses as cache and MSHR hits. Delorean, then, uses StatStack to determine capacity misses and the limited associativity model to determine conflict misses. All other accesses that appear to be misses are due to insufficient warming and are treated as hits.

due to the lack of warming. We refer to this type of miss as *warming misses*. While capacity and conflict misses correspond to the real behavior of the workload, warming misses are an artifact of the sampled simulation, i.e., the fact that the simulator has started executing the workload at a simulation point and not from the start. To accurately simulate without cache warm up in a sampled simulation framework, the Analyst has to determine the warming misses and handle them as cache hits (Figure 4).

1) The Lukewarm Cache: While the Analyst does not perform any cache warming, prior to the simulation point, it simulates a small number of instructions (30,000) necessary to warm up the microarchitectural state of the core. With this small amount of warming, only a small part of the cache state is warm. Nikoleris et al. [9] refer to the cache at this state as the *lukewarm cache*. Our experiments show that for the SPEC CPU 2006 benchmarks, the hit ratio in the lukewarm data cache (64 KiB) is on average 88%.

A number of memory accesses miss in the lukewarm cache when there is already an outstanding miss for the same cacheline. These types of accesses are typically referred to as *Miss Status Holding Register (MSHR)* hits or delayed hits [10]. The Analyst models these types of accesses using the lukewarm cache, provided that it has determined the first accesses to that cache block to be a miss. Our experiments show that for the SPEC CPU 2006 benchmarks, using a 64 KiB data cache after 30,000 instructions, 95% of the requests either hit in the lukewarm data cache or in its MSHRs.

The Analyst uses the lukewarm cache to accurately handle timing for a large fraction of memory requests that either hit in the cache or in the MSHRs. Requests that miss both in the cache and the MSHRs are the key accesses as recorded by the Scout. The Analyst uses cache modeling with the MRI that the Explorers collected to determine if they are capacity, conflict or warming misses. When an access is ruled out as a capacity or a conflict miss, it is a warming miss. Therefore, it is handled as a cache hit.

2) Capacity Misses: For an access that misses in the lukewarm cache, the Analyst first determines whether it is a capacity miss. The Analyst uses the obtained MRI and StatStack to estimate its stack distance. If the estimated stack distance is greater than the total number of blocks in the cache then the access is classified as a capacity miss.

3) Conflict Misses: As StatStack models a fully associative cache, the Analyst needs to determine conflict misses. To do that, it first examines the lukewarm cache. If the referenced set in the lukewarm cache is full, the access is certainly a conflict miss. However, due to the lack of warming, many sets of the lukewarm cache might incorrectly appear not to be full.

The Analyst uses the limited associativity model [9] to determine conflict misses. The limited associativity model inspects memory instructions with uniform access patterns. Such patterns, dominated by large strides, result in uneven usage of the cache sets and consequently, use only a fraction of the cache. If the working set of that particular instruction does not fit in the determined fraction of the cache, it will result in a conflict miss [9].

Method	HW changes	SW changes	Large Caches	Speed
SMARTS	✓	✗	✗	+
FSA	✓	✓	✗	+/+++
CoolSim	✓	✓	✓	++
Delorean	✓	✓	✓	+++

TABLE I: Comparison of simulation frameworks. Delorean is more flexible, allowing for software and hardware changes without sacrificing performance due to long cache warm-ups.

Pipeline	gem5's default OoO x86 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 KiB, 2-way LRU
	L1D	64 KiB, 2-way LRU
	L2	1 MiB to 512 MiB, 8-way LRU

TABLE II: Summary of simulation parameters.

Using the limited associativity model, the Analyst models instructions with uniform access patterns and big strides using a smaller cache. For such an access, if the stack distance analysis determines that it misses in the smaller cache, it is treated as a conflict miss.

4) Warming Misses: The lukewarm cache directly handles accesses that are classified as capacity or conflict misses. For a memory access that is determined to be a warming miss, i.e., is an artifact of the insufficient cache warming, the Analyst brings the data from the memory and populates the cacheline without any delay (functional access). The request that would otherwise miss, now finds the block in the cache and is serviced as a cache hit.

IV. Evaluation

In this section we look into the accuracy and the performance of our simulation framework. In our experiments, we simulate a 64-bit x86 system (Ubuntu 12.04.5 LTS running Linux 3.2.44) with split 2-way 64 KiB L1 instruction and data caches and a unified 8-way L2 cache with sizes ranging from 1 MiB to 512 MiB. A summary of the important simulation parameters can be found in Table II. All benchmarks were compiled with GCC 4.6.3. We evaluate the system using the SPEC CPU2006 benchmarks[1] 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 measured on an Intel Xeon E5520.

Due to the high overhead of the reference experiments, we use 10 simulation points spread uniformly across 10 B instructions (1 B instructions apart). For each simulation point, we use gem5's O3CPU and run for 30,000 instructions to warm the microarchitectural state and for 10,000 instructions to measure performance.

Delorean uses multiple instances of gem5: the Scout, the Explorers and the Analyst. For the chosen simulation sample size and period, we use 4 Explorers which profile for 5 M, 50 M, 100 M and 1 B instructions before each simulation point. All Explorers use VDP, except for Explorer-1, that uses directed functional profiling. Explorer-1 profiles a relatively short interval of 5 M instructions for the full set of key accesses and therefore places many watchpoints. The watchpoints interrupt the short execution of Explorer-1 too often to make VDP worthwhile. As a result, Explorer-1 implements directed profiling using gem5's atomic CPU model.

All Explorers collect MRI for the key accesses and a representative sample of the reuses of the vicinity. The reuses of the vicinity are randomly collected with a sampling rate of 1 over 100 k memory instructions.

A. Accuracy

To evaluate Delorean's accuracy, we compare the results from simulations with different L2 cache sizes. Our reference uses functional simulation between the simulation points to keep the cache warm.

[1]We were unable to run 403.gcc, 433.milc., 447.dealII, 481.wrf and 482.sphinx3 these benchmarks either produced outputs that could not be verified with the reference or did not run to completion and therefore we did not use them in our evaluation.

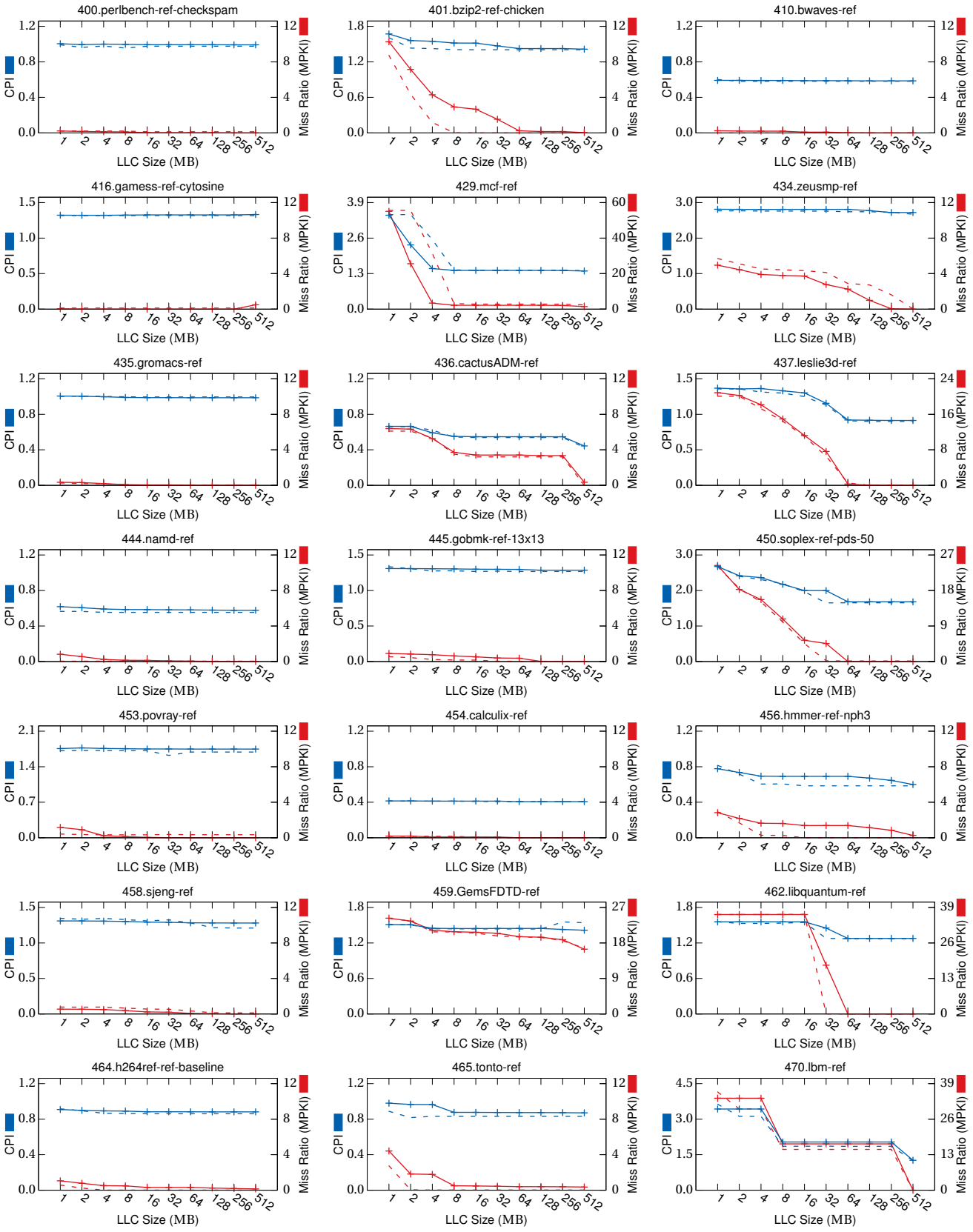

Fig. 5: CPI and L2 miss ratio measured for the SPEC CPU2006 benchmarks for L2 sizes from 1 MiB to 512 MiB. The reference is shown with dotted lines and simulation that uses Delorean with solid lines.

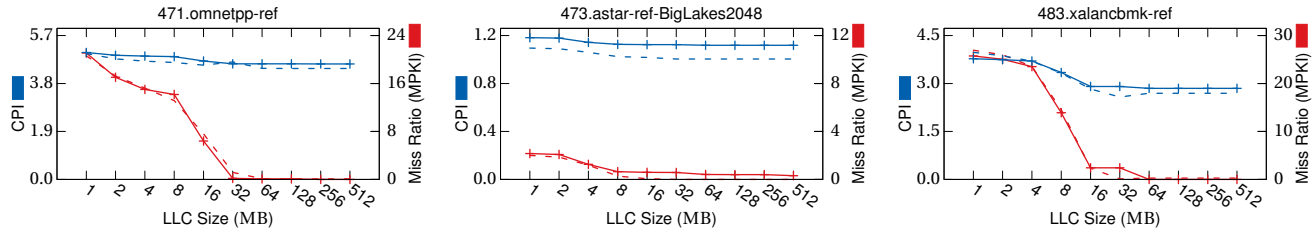

Fig. 5: CPI and L2 miss ratio measured for the SPEC CPU2006 benchmarks for L2 sizes from 1 MiB to 512 MiB. The reference is shown with dashed lines and simulation that uses Delorean with solid lines.

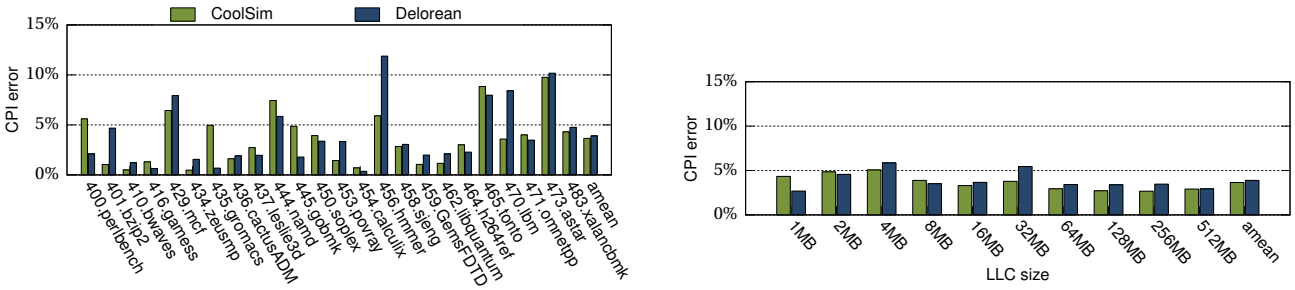

Fig. 6: Average CPI error for the 27 SPEC CPU2006 benchmarks and cache sizes from 1 MiB to 512 MiB. Delorean has comparable CPI error to CoolSim but is 9×faster.

Figure 5 shows the CPI and the miss ratio estimations compared to the reference for the 24 benchmarks SPEC CPU2006 benchmarks. 456.hmmer-ref-nph3 and 473.astar-BigLakes2048 show the largest average error. For 473.astar-BigLakes2048, the largest part of the error comes from the L1D miss ratio, where Delorean fails to track it accurately. The L1D is only 64 KiB where statistical errors are more pronounced. Delorean is accurate across the whole range of cache sizes, with the exception of 429.mcf-ref and 401.bzip2-ref-chicken that show errors for smaller caches. Increasing the density of the vicinity reuses improves the accuracy for these two benchmarks as well as for 456.hmmer-ref-nph3 and brings their average error down from 2.09% to 1.09% for 401.bzip2-ref-chicken, from 7.9% to 4.67% for 429.mcf-ref and form 11.8% to 2.7% for 456.hmmer-ref-nph3. Overall, the average error across all cache sizes for the SPEC CPU 2006 is 3.89% comparable to CoolSim which is 3.64%.

B. Performance

Due to the way the passes pipeline, Delorean's performance is determined by the slowest of its instances, similar to a multi-threaded application where the slowest thread determines the overall performance.

Using multiple passes, we reduce the number of active watchpoints for longer profiling intervals and consequently the number of watchpoint hits. For example, using a single Explorer for 429.mcf we see 58,094,363 watchpoint hits

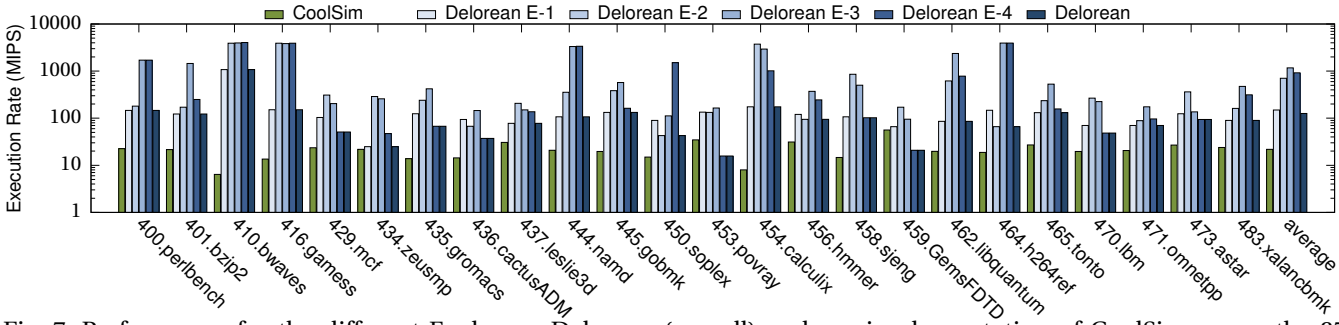

Fig. 7: Performance for the different Explorers, Delorean (overall) and our implementation of CoolSim across the 27 SPEC CPU2006 benchmarks. For clarity, we omit the performance of the Scout and the Analyst, as they fast-forward between simulation points using VFF and always perform better.

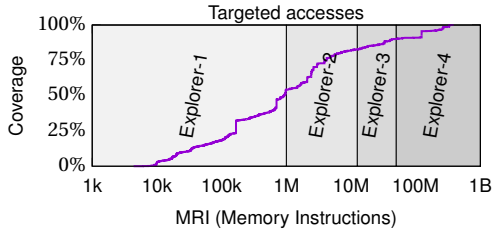

Fig. 8: Cumulative distribution histogram for the reuse distances of the key accesses as obtained by the Explorers across all SPEC CPU2006 benchmarks.

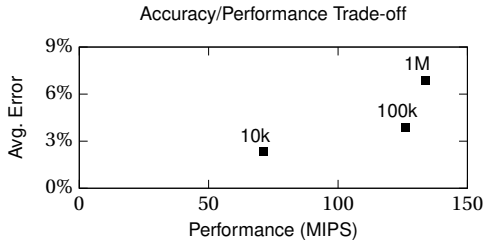

Fig. 9: Performance-accuracy trade-off. A denser vicinity reuse sampler increases accuracy, but slows down Explorers resulting in lower overall performance.

with a performance of 11.3 MIPS; on the other hand, using 4 Explorers we trigger 462,974 watchpoint hits and we achieve overall performance of 50 MIPS.

Figure 7 shows the performance of the Explorers. The Scout and the Analyst always fast-forward between the simulation points (1 B instructions) at near-native speed and perform detailed simulation for just 40.000 instructions and naturally outperform the Explorers. The overall performance for Delorean is determined by the slowest Explorer which runs on average at 148 MIPS. Some Explorers do not perform any profiling. They fast-forward through the entire workload as the key accesses have already been profiled by previous Explorers. For example, Explorer-3 and Explorer-4 run at full speed for 400.perlbench-ref-checkspam, as Explorer-1 and Explorer-2 have already obtained the MRI for the key accesses. Overall, Delorean is 9× faster than CoolSim which runs on average at 17 MIPS.

The amount of profiling for each of the Explorers is shown in Figure 8. Across the 24 SPEC CPU2006 benchmarks, more than 55% of the key accesses have reuse distance smaller than 1 M memory instructions which is typically within reach of Explorer-1. 82% of the key accesses have a reuse distance smaller 12.5 M memory instructions; in the range of Explorer-2, that profiles for 50 M instructions. Explorer-3 profiles reuses smaller than 25 M memory instructions and increases the coverage to 85%. Finally, Explorer-4 obtains the remaining reuses.

C. Performance-Accuracy Trade-off

Delorean's accuracy and performance are dependent on the density of the collected vicinity reuses. The density of the vicinity can improve the accuracy of the statistical model and Delorean's overall accuracy. However, higher density increases the profiling overhead of the Explorers. Figure 9 shows the trade-off between accuracy and performance. With a density of 1 over 100 k instructions we can simulate at 148 MIPS with an error of 3.89%,

V. Related Work

Cache warming has received a lot of attention since the introduction of sampled simulation. Wenisch et al. [11] store the state of the caches and other micro-architectural structures in checkpoints that they call *Flex points*. They eliminate warming in SMARTS, reducing the overall overhead. Flex points are large (20 MiB to 100 MiB) and impractical. In follow-up work, they introduce *Live points* [12] and reduce the space requirements for each checkpoint down to 142 KiB. Barr et al. [13] propose, The Memory Timestamp Record (MTR), a method to record memory patterns, compress and store them for use in checkpoints. In contrast, Delorean's obtained MRI is only 26 KiB per simulation point and can be used to model a cache of any size. Hassani et al. [14] use sampled simulation combined with MTR and in-memory checkpoints to evaluate benchmarks in a few seconds.

Other works focus on minimizing the amount of warming needed to produce accurate results. Haskins and Skadron [15] and Luo et al. [16] use heuristics to find the minimum number of instructions needed to warm a cache of specified

size. Burugula and Skadron [17] 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 which give a large enough cumulative distribution of MRRLs is used as the warming interval. Eeckhout et al. [18] introduce the concept of *Boundary Line Reuse Latency (BLRL)* which extends the concept of MRRLs. They apply similar heuristics to find the optimal warm-up period. Van Ertvelde et al. [19] extend on the concept of BLRL using a form of hardware state checkpoints. Sandberg et al. [2] use a limited warming of 5M and 25M instructions to warm 2 MiB and 8 MiB caches. They propose a method that uses 2 parallel simulations, pessimistic and optimistic, to bound the maximum error due to warming. While minimizing warming improves simulation efficiency, the minimum amount of warming needed for very large caches continues to be very long, and dominates the simulation time for traditional methodologies.

Stack distance analysis has been extensively used to model caches. Mattson et al. [7] use a simple stack algorithm and inspect every access to collect the stack distance information. To improve performance, researchers use k-ary [20] and AVL [21] trees instead of linked lists. However, all of the proposed methods have to inspect all memory accesses and measure the stack distance. Other works [22] have proposed hardware accelerated stack distance collection. Liu and Mellor-Crummey [23] use a technique based on shadow profiling that forks off a redundant copy of an application, instrumented by Pin, to measure the stack distances for a selected set of references [23]. 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.

VI. Conclusion

Sampled simulation allows for realistic workload evaluation through a reduction in instruction count by 3 orders of magnitude. Cache warming, however, dominates the simulation overhead and prevents simulation frameworks from realizing a proportional reduction in simulation time while maintaining flexibility to make changes in software and significant changes in hardware.

Delorean eliminates cache warming using statistical cache modeling. It uses VDP to advance the execution to the next simulation point at near-native speed and profile the application. Its multi-pass profiler identifies key accesses, obtains their MRI and uses it to accurately model cache behavior at each simulation point.

Delorean demonstrates high accuracy across a wide range of cache sizes (1 MiB to 512 MiB) and can estimate an application's CPI with an average error of just 3.89%. At the same time, it uses VDP to improve simulation performance compared to prior art, at 148 MIPS, 150× faster than traditional SMARTS simulations that use of functional cache warming.

References

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *ACM SIGARCH Computer Architecture News*, 2011.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[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] D. Eklov and E. Hagersten, "StatStack: Efficient Modeling of LRU Caches," in *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2010.

[9] N. Nikoleris, A. Sandberg, E. Hagersten, and T. E. Carlson, "CoolSim: Statistical Techniques to Replace Cache Warming with Efficient, Virtualized Profiling," in *SAMOS*, 2016.

[10] S. Belayneh and D. R. Kaeli, "A Discussion on Non-blocking/Lockup-free Caches," *ACM SIGARCH Computer Architecture News*, vol. 24, no. 3, 1996.
[11] 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.
[12] 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*, vol. 26, no. 4, 2006.
[13] 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.
[14] S. Hassani, G. Southern, and J. Renau, "LiveSim: Going Live with Microarchitecture Simulation," in *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*, 2016.
[15] J. Haskins, J.W. and K. Skadron, "Minimal Subset Evaluation: Rapid Warm-Up for Simulated Hardware State," in *Proc. International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, 2001.
[16] Y. Luo, L. K. John, and L. Eeckhout, "Self-Monitored Adaptive Cache Warm-Up for Microprocessor Simulation," in *SBAC-PAD*, 2004.
[17] 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.
[18] L. Eeckhout, Y. Luo, K. De Bosschere, and L. K. John, "BLRL: Accurate and Efficient Warmup for Sampled Processor Simulation," *Computer*, vol. 48, no. 4, 2005.
[19] L. Van Ertvelde, F. Hellebaut, L. Eeckhout, and K. De Bosschere, "NSL-BLRL: Efficient CacheWarmup for Sampled Processor Simulation," in *Proc. Annual Symposium on Simulation*, 2006.
[20] B. T. Bennett and V. J. Kruskal, "LRU Stack Processing," *IBM J. Res. Dev.*, vol. 19, no. 4, 1975.
[21] F. Olken, "Efficient Methods for Calculating the Success Function of Fixed Space Replacement Policies," Ph.D. dissertation, 1981, m.Sc. : Calif. Univ. Berkeley.
[22] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm, "RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations," in *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.
[23] 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.