Directed Statistical Warming through Time Traveling

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

Lieven Eeckhout
lieven.eeckhout@ugent.be
Ghent University
Belgium

Erik Hagersten
erik.hagersten@it.uu.se
Uppsala University
Sweden

Trevor E. Carlson
tcarlson@comp.nus.edu.sg
National University of
Singapore

ABSTRACT

Improving the speed of computer architecture evaluation is of paramount importance to shorten the time-to-market when developing new platforms. Sampling is a widely used methodology to speed up workload analysis and performance evaluation by extrapolating from a set of representative detailed regions. Installing an accurate cache state for each detailed region is critical to achieving high accuracy. Prior work requires either huge amounts of storage (checkpoint-based warming), an excessive number of memory accesses to warm up the cache (functional warming), or the collection of a large number of reuse distances (randomized statistical warming) to accurately predict cache warm-up effects.

This work proposes DeLorean, a novel statistical warming and sampling methodology that builds upon two key contributions: *directed statistical warming* and *time traveling*. Instead of collecting a large number of randomly selected reuse distances as in randomized statistical warming, directed statistical warming collects a select number of key reuse distances, i.e., the most recent reuse distance for each unique memory location referenced in the detailed region. Time traveling leverages virtualized fast-forwarding to quickly 'look into the future' — to determine the key cachelines — and then 'go back in time' — to collect the reuse distances for those key cachelines at near-native hardware speed through virtualized directed profiling.

Directed statistical warming reduces the number of warm-up references by 30× compared to randomized statistical warming. Time traveling translates this reduction into a 5.7× simulation speedup. In addition to improving simulation speed, DeLorean reduces the prediction error from around 9% to around 3% on average. We further demonstrate how to amortize warm-up cost across multiple parallel simulations in design space exploration studies. Implementing DeLorean in gem5 enables detailed cycle-accurate simulation at a speed of 126 MIPS.

CCS CONCEPTS

• **Computing methodologies → Modeling and simulation**; • **Hardware → Analysis and design of emerging devices and systems**; • **Computer systems organization →** *Serial architectures.*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MICRO-52, October 12–16, 2019, Columbus, OH, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358264

KEYWORDS

performance analysis, architectural simulation, sampled simulation, statistical cache modeling, cache warming

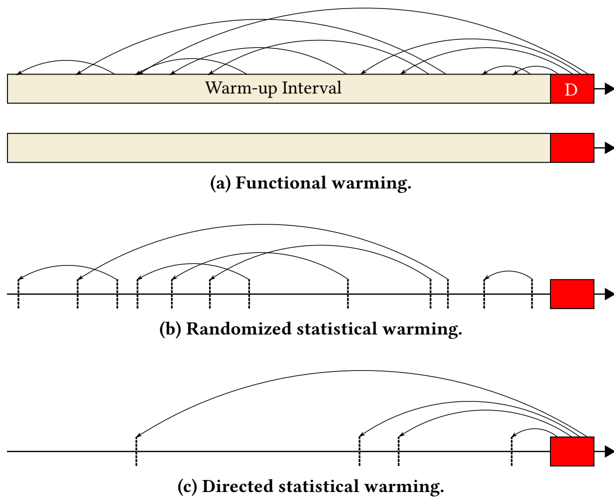
ACM Reference Format:
Nikos Nikoleris, Lieven Eeckhout, Erik Hagersten, and Trevor E. Carlson. 2019. Directed Statistical Warming through Time Traveling. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3352460.3358264

1 INTRODUCTION

Computer architects use a variety of tools and methodologies to analyze workloads and evaluate new design enhancements. Instrumentation and profiling using tools such as Pin [18] and Valgrind [21] are useful to analyze a workload's characteristics, e.g., generate working set curves to visualize cache miss rate as a function of cache size. Cycle-accurate architecture simulation, e.g., gem5 [8], models cycle-by-cycle execution behavior to predict a workload's performance on a particular microprocessor configuration. Although these tools and methodologies have proven their merit, they all suffer from limited speed. Analyzing even a few minutes of real hardware execution leads to days, if not weeks, of experimentation time.

Improving the speed of computer architecture evaluation is critically important. Shortening the design cycle provides a competitive time-to-market advantage in the computer industry. Moreover, a fast evaluation methodology allows for a more thorough exploration of the design space with more workloads, which leads to an improved design. A commonly used technique to speed up architecture evaluation is to sample the workload, i.e., evaluate (a) small region(s) of the execution in detail and then extrapolate to the entire execution [28, 34]. A major challenge in sampled evaluation however is to quickly and accurately warm up the microarchitecture state for each detailed region, which is particularly challenging for the largest structures in the processor, i.e., caches. Although sampling is not a new methodology, fast and accurate cache warming is still an unsolved problem, especially in light of emerging very large DRAM caches.

Although important advances have been made in the past, there are still major limitations. *Checkpointed Warming (CW)* takes a checkpoint of the microarchitecture state prior to each region [32, 33]. Unfortunately, checkpoints require huge amounts of disk space, and are not reusable across software changes (i.e., compiler updates, changing compiler options, dynamically generated code through Just-in-Time optimization, etc.) nor hardware changes — although there exist solutions to make checkpoints transferable across cache structures [1, 32, 33]. *Functional Warming (FW)* on the other hand does not incur any storage overhead and is transferable across both hardware and software changes [26, 34]. Functional warming



(a) Functional warming.

(b) Randomized statistical warming.

(c) Directed statistical warming.

Figure 1: Three approaches to warm up a cache for the detailed region D: (a) functional warming, (b) randomized statistical warming, and (c) directed statistical warming.

warms up the microarchitecture state by simulating the microarchitecture structures in the *warm-up interval* prior to each *detailed region*, without doing any actual performance measurements, see Figure 1(a). Unfortunately, functional warming is slow because it simulates the cache for *every* memory access within the warm-up interval prior to the detailed region [9, 13]. More recently, *Randomized Statistical Warming (RSW)* [23] takes a random set of memory accesses in the warm-up interval for which it computes reuse distances (i.e., number of memory references between two references to the same memory location), see Figure 1(b). Statistical models [11] transform reuse distances into stack distances (i.e., number of *unique* memory references between two references to the same memory location) to then predict miss rates for a range of cache configurations. RSW is substantially faster than FW because fewer memory references need to be profiled [23]. Nevertheless, RSW requires a large number of randomly selected reuse distances, most of which are useless and only a few of which are critical to accurately predict a detailed region's cache behavior.

This paper proposes DeLorean, which builds upon two major contributions: (i) directed statistical warming and (ii) time traveling. *Directed Statistical Warming (DSW)* makes the observation that to obtain an accurate picture of the cache state prior to a detailed region, we only need a few select reuse distances, in contrast to RSW. More specifically, we only need the most recent reuse for each unique memory location referenced in the detailed region, i.e., we do not need to collect reuse distances that fall entirely within the warm-up interval, see Figure 1(c). DSW dramatically reduces the number of reuse distances that need to be collected, thereby reducing total work spent during warm-up compared to RSW.

DSW is not a panacea though: to direct reuse distance collection, we need to know which memory locations to compute the reuse distance for, i.e., these are the ones referenced in the detailed region. The problem now is that we only get to know these key memory locations once we have evaluated the detailed region, which appears after the warm-up interval. This is where *Time Traveling (TT)* comes in. TT first quickly advances to the next detailed region to collect so-called *key cachelines*, which are all the unique cachelines referenced in the region. TT then goes back in time to collect the last reuse distance for each key cacheline as needed for DSW. This is done in an iterative way, using multiple rollbacks if needed, to obtain high simulation speed while collecting reuse distances for all key cachelines. TT is implemented through a combination of virtualized execution (to quickly fast-forward to the next detailed region to determine the key cachelines) and virtualized directed profiling (to sample the key reuse distances at near-native hardware speed). The name 'DeLorean' is chosen after the time-travel vehicle in the 'Back to the Future' feature films to represent going back and forth in time. DeLorean is implemented in the gem5 detailed cycle-level processor simulator using the Linux Kernel Virtual Machine (KVM) to fast-forward between detailed regions and sample reuse distances at near-native hardware speed.

DeLorean is widely applicable to speed up architecture evaluation. We experimentally demonstrate the speed and accuracy of DeLorean for collecting working set curves and speeding up sampled simulation, using the SPEC CPU benchmarks and the gem5 simulation infrastructure. DSW reduces the number of warm-up samples (number of collected reuse distances) by 30× and 100,000× compared to RSW and FW, respectively. TT translates this reduction in a simulation speedup of 5.7× compared to RSW and 96× compared to FW. Moreover, because DSW builds upon a microarchitecture-independent characteristic, namely reuse distance, warming overhead can be amortized across multiple parallel simulations during design space exploration studies, further increasing simulation speed advances over traditional simulation-based approaches. In addition to substantially improving simulation speed, DeLorean also improves accuracy: prediction error is reduced from around 9% for RSW to around 3% across different cache sizes. Ultimately, DeLorean enables detailed cycle-accurate gem5 simulations at a speed of 126 MIPS. We make the source code and scripts of our simulation framework publicly available.

2 BACKGROUND

Before describing DeLorean in detail, we first provide critical background information.

2.1 Sampling

Sampling is a widely used technique to speed up workload analysis and computer architecture performance evaluation by considering a select number of representative detailed regions that are evaluated in detail to then extrapolate from [34]. The key challenge in sampling is to get (i) the correct architecture state and (ii) an accurate microarchitecture state at the beginning of each detailed region. Common techniques to obtain the correct architecture state are functional fast-forwarding, checkpointing and virtualized fast-forwarding. Functional fast-forwarding [34] leverages functional simulation to get to the next representative region, which is slow. Checkpointing [32, 33] takes a snapshot of the architecture state for each region, which is fast but does not allow for changes to the

software. Virtualized Fast-Forwarding (VFF) [26] leverages hardware virtualization to quickly get to the next region, while enabling software changes. Time Traveling (TT), as proposed in DeLorean, builds upon virtualized fast-forwarding.

Obtaining the microarchitecture state at the beginning of each region is an accuracy/speed/overhead trade-off. As mentioned in the introduction, checkpointed warming (CW) [32, 33] is fast, requires huge storage overhead, and does not allow for software changes. Functional warming (FW) [26, 34] does not incur any storage overhead, allows for software changes, but is slow. Randomized statistical warming (RSW) [22, 23] shares all the benefits of FW but is faster because many fewer memory references need to be analyzed in the warm-up interval prior to each detailed region. In this work, we make the observation that collecting that many warm-up references under RSW is wasteful, which provides an opportunity to reduce warm-up by an order of magnitude through directed statistical warming (DSW).

2.2 Statistical Cache Modeling

Statistical warming — both RSW and DSW — builds upon statistical cache modeling. By leveraging a workload characteristic that is independent of the underlying microarchitecture and cache hierarchy, statistical cache models estimate cache miss rates for any size caches from a single workload profile. The workload characteristic that underpins statistical cache modeling is *stack distance*, which is defined as the number of unique references (cachelines) between two accesses to the same cacheline [20]. Stack distance allows for accurate modeling of fully-associative, LRU caches [15]: a stack distance larger than the size of the cache leads to a miss; if not, it is a hit.

Unfortunately, obtaining the exact stack distance distribution is a costly operation because one needs to inspect every memory access between the two consecutive accesses to the same cacheline. Eklov and Hagersten [11] provide a solution by showing that stack distance can be accurately estimated using its corresponding *reuse distance*, which is defined as the number of memory accesses (not necessarily *unique* accesses) between two accesses to the same cacheline. Computing reuse distance is much faster than computing stack distance because one only needs to count the number of memory accesses between two accesses to the same memory location, and does not need to process their addresses to determine the unique accesses. Moreover, prior work has shown that the reuse distance distribution can be accurately approximated by tracking a subset of randomly selected reuse distances and memory locations [5]. Finally, statistical cache modeling has been generalized and demonstrated for various cache replacement policies, as well as for multiprogrammed and multi-threaded workloads, as we will elaborate on in Section 4.

2.3 Randomized Statistical Warming

Recent work proposed randomized statistical warming (RSW) to tackle the warm-up problem in sampled evaluation [23]. Instead of warming up the cache by effectively populating the cache with simulated accesses and evictions as in functional warming, randomized statistical warming *predicts* whether a memory access in the detailed region would have been a hit or a miss if the cache would have been perfectly warmed up prior to the detailed region. To this end, RSW computes an (approximate) reuse distance distribution during the warm-up interval, which then serves as input to a statistical cache model to predict hits and misses.

RSW tracks randomly selected memory locations and computes their reuses during the warm-up interval prior to a detailed region. Prior work [5, 23] uses watchpoints to do so during native execution: hardware performance counters are used to count the number of memory accesses between a (randomly selected) reuse. A reuse is detected by setting a watchpoint using the operating system's page protection mechanism; a memory location touched by the workload is randomly selected, and a watchpoint is set on that same address to compute its reuse distance. Execution between watchpoints runs at native speed, and watchpoints stop the execution when there is an access to the protected page. Note that a stop on a watchpoint does not necessarily imply a reuse. Any access to the protected page incurs a watchpoint stop, which could be a false positive. In case a reuse is detected (i.e., true positive), its distance is computed and the watchpoint is removed.

Computing an (approximate) reuse distance distribution is much faster than simulating (and effectively warming up) a cache. Prior work reports that RSW improves detailed simulation speed by 17× compared to functional warming [23]. In this work, though, we find there is substantial room for improvement. To accurately predict hits and misses for specific memory accesses in the detailed region, RSW needs to collect a large number of reuse distances, hoping that a sufficient number of reuse distances will have been collected for all the load PCs that occur in the detailed region. The statistical model that underpins RSW uses reuse distance distributions per load PC and thus needs a sufficiently large number of reuse distances per PC for an accurate prediction. Because the detailed region is relatively small compared to the much larger warm-up interval, the likelihood of sampling a reuse distance in the warm-up interval for a particular load PC in the detailed region is not that high. Hence, RSW needs to collect a large number of reuse distances, many of which turn out to be useless because they are collected for load PCs that do not occur in the detailed region. This inefficiency incurs a non-trivial warm-up cost.

3 DELOREAN

To lower the warm-up cost compared to RSW, a directed approach is needed. Ideally, we would like to reduce the number of reuse distances that we need to track in the warm-up interval (to achieve high speed), while, at the same time, capturing the exact reuse information for every memory access in the detailed region (to maintain high accuracy). This is effectively what DeLorean accomplishes. DeLorean is a novel statistical warming and sampling methodology that builds on two major contributions: (i) directed statistical sampling and (ii) time traveling. We now discuss these in detail.

3.1 Directed Statistical Warming

Directed statistical warming (DSW) reduces the overhead of statistical warming by directing reuse distance collection to a set of so-called *key cachelines*. The key reuse distances and their vicinity distributions are then used to predict warm-up effects using statistical cache models.

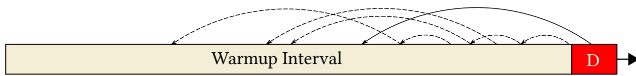


Figure 2: A key cacheline is accessed in the detailed region D. Its key reuse distance is the last reuse in the warmup interval (shown as a solid arc). The reuse distances in the vicinity are shown as dashed arcs.

3.1.1 Key Reuses and their Vicinity. DSW first identifies the small set of cachelines (*key cachelines*) that are accessed during the detailed region, and for each key cacheline, DSW records the reuse distance since it was last accessed *before* the detailed region. More specifically, DSW determines the first access to each unique cacheline in the detailed region and then measures its 'backward' reuse distance, i.e., the nearest previous access to the same cacheline in the warm-up interval. We refer to these reuses as *key reuse distances*. This is illustrated in Figure 2.

In addition to the key reuse distances, we also compute the *vicinity reuse distance distribution*, or the distribution of reuse distances in the vicinity of the key reuse distances, see Figure 2. The vicinity is defined as the interval surrounding the key reuse distance — any reuse that completely falls within or crosses the key reuse distance is part of the vicinity. The vicinity reuse distribution is approximated by taking randomly selected reuse distance samples. Note one key difference with how RSW collects reuse distance samples. As mentioned in the previous section, RSW needs to collect a large number of reuse distances to cover a sufficient number of reuse distances for the load PCs in the detailed region. The vicinity reuse distribution on the other hand needs an order of magnitude fewer reuse distance samples because its sole purpose is to estimate the average behavior of the accesses close to the key reuse distances, and not to estimate the per-PC reuse distances of the key cachelines. The vicinity distribution is used to predict whether the key reuse distances will lead to a hit or a miss in the detailed region, as we will describe next. The order of magnitude reduction in reuse distance samples needed under DSW leads to a substantial speedup during warmup.

DSW has two key advantages over RSW: (i) the exact reuse distances for *all* memory accesses at the detailed region are known (high accuracy), and (ii) much fewer reuse distances need to be collected by focusing on the key reuse distances and their vicinity (high speed).

3.1.2 Statistical Warming. We use the key reuse distances and their vicinity distributions to statistically 'warm' the caches prior to a detailed region. DSW does not actually warm up the cache by emulating the cache's operation through accesses and evictions. Instead, DSW predicts whether a specific memory access in the detailed region is a hit or a miss based on the distribution of the key reuse distances and their vicinities.

Statistical cache warming builds upon the key concept of the *warming miss*, which is a request in the detailed region that misses due to lack of warming. While cold, capacity and conflict misses correspond to real workload behavior, warming misses are a sampling artifact. The insight behind statistical warming is to determine the

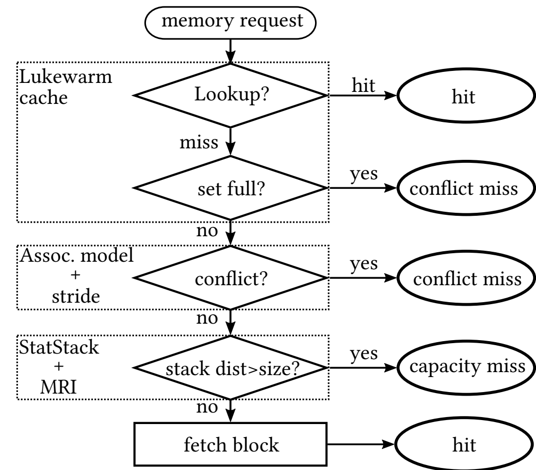


Figure 3: Statistical warming. *The lukewarm cache determines accesses with short reuses as cache and MSHR hits. Then, the limited-associativity model determines conflict misses and the statistical cache model determines capacity misses. All other accesses that appear to be misses are due to insufficient warming and are treated as hits.*

warming misses and handle them as cache hits. We now describe how we determine warming misses, see also Figure 3.

Lukewarm Cache and MSHR Hits: It is common in sampled simulation to warm up the microarchitecture state (processor pipeline, predictors, prefetchers, caches) through a *detailed warm-up* using a small number of instructions (e.g., 30,000) prior to the detailed region [34]. With this small amount of warming, only a small part of the cache state is warm, which we refer to as the *lukewarm cache*. A hit in the lukewarm cache in the detailed region would definitely have been a hit in the cache if it were perfectly warmed up. Our experiments show that for the SPEC CPU2006 benchmarks, the hit rate for a detailed region of 10,000 instructions and a 64 KiB lukewarm D-cache ranges between 27.5% and 100% with an average of 93.5%.

A number of memory accesses miss in the lukewarm cache when there is already an outstanding miss for the same cacheline. These accesses are typically handled as *Miss Status Holding Register (MSHR)* hits [3]. DSW models MSHR hits as a cache hit (in case of cache simulation) or a delayed hit (in case of processor simulation). Our experiments show that between 46.1% and 100% of the requests (96.7% on average) are hits or delayed hits in a lukewarm 64 KiB D-cache with 8 MSHRs.

Conflict Misses: If a referenced set in the lukewarm cache is full in the detailed region, the access certainly is a conflict miss. Hence, the access is modeled as a miss.

For some (outlier) benchmarks, we note that some load PCs exhibit a dominant large stride, which results in uneven usage of the cache sets. For example, a 512-byte stride will only touch upon one eighth of the cache sets assuming an 64-byte cacheline. In other words, dominant large strides limit the effective usage of the cache, which ultimately leads to conflict misses. We leverage the previously proposed limited-associativity model [23] to determine such conflict misses.

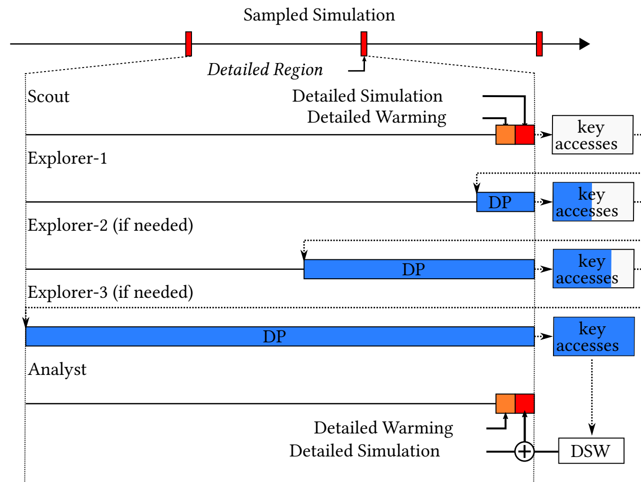


Figure 4: Time Traveling. *Scout leverages VFF to quickly advance to identify key cachelines in the next detailed region. Then, the Explorers go back in time and collect the key reuse distances and the respective vicinity reuse distributions. Finally, the Analyst uses DSW to evaluate the detailed region.*

Capacity Misses: DSW uses the key reuse distances and their vicinity distributions to determine capacity misses. We first convert the key reuse distances and their vicinity reuse distance distributions into stack distances and stack distance distributions, respectively, using the well-established statistical model [11] as previously described in Section 2.2. If the stack distance of the memory access is larger than the total number of cachelines in the cache, the memory access is classified and modeled as a capacity miss.

Warming Misses: All remaining memory accesses are warming misses. They miss in the lukewarm cache and MSHRs, and are not predicted to be conflict nor capacity misses. Instead, they are an artifact for insufficient cache warming. DSW models warming misses as cache hits.

3.2 Time Traveling

DSW reduces the number of reuse distances to collect by an order of magnitude compared to RSW, as we will quantify in the evaluation section. One key implicit assumption underlying DSW though is that we need future knowledge, i.e., we need to know the key cachelines in the detailed region to know which reuse distances to collect in the warm-up interval. This is a non-trivial problem to solve in a single evaluation run while maintaining (very) high evaluation speed.

Time traveling (TT) solves exactly this problem by using multiple passes, as illustrated in Figure 4. We utilize a *Scout* pass, multiple *Explorer* passes, and finally an *Analyst* pass. Each of the passes is a separate instance (process) of the same evaluation. TT performs all passes for each detailed region and it does so in a pipelined fashion, i.e., it first performs Scout, then Explorer-1 through Explorer-N, and finally Analyst. However, the evaluation of subsequent detailed regions is parallelized over time. As soon as a pass finishes its current detailed region, it moves on to the next detailed region. For example, when Scout is done with detailed region m, it moves on to region $m + 1$, while Explorer-1 processes region m, etc. This way we can pipeline the evaluation process as long as we have enough cores to run the different passes concurrently. OS pipes are used to synchronize and communicate between passes. We now describe the different passes.

Scout: The Scout identifies the key cachelines in the detailed region. It uses *virtualized fast-forwarding (VFF)* at near-native speed to advance the execution to the next detailed region, at which point it switches to functional simulation to record the set of key cachelines. These are the unique cachelines accessed within the detailed region. Interestingly, the number of key cachelines is rather small: for SPEC CPU2006 and a detailed region of 10,000 instructions, we find that the number of key cachelines varies between 1 and 2,907, with an average of 151.

Explorers: The Explorers collect the key reuse distances and their vicinity distributions. For each key cacheline recorded by the Scout, the Explorers determine its last access prior to the detailed region. To do this, the Explorers set watchpoints on the key cachelines — a technique which we call *directed profiling (DP)*. As the number of key cachelines for a detailed region is relatively small, it may appear that the task of measuring reuse distances is fairly trivial. However, we need to find the *last* access to the cacheline before the detailed region. This implies that we keep DP active for the entire warm-up interval. A naive implementation that effectively keeps DP active for the entire warm-up interval is too slow.

Instead, we use multiple Explorers, as shown in Figure 4. Explorer-1 fast-forwards using VFF and switches to DP 5 M instructions before the detailed region. As soon as it reaches the end of the warm-up interval, it determines which key reuse distances have been computed. The remaining key cachelines have a reuse distance that is larger than 5 M instructions — these key cachelines (if any) are then passed on to Explorer-2.

Explorer-2 switches to DP 50 M instructions before the detailed region. Its task is to find the reuses for the key cachelines that were outside the reach of Explorer-1. The subset of undiscovered reuses is not only significantly smaller than the original set of key cachelines, it also tends to contain cachelines with lower temporal locality. As a result, watchpoints for these key cachelines do not trigger as often, hence overhead is lower. Key cachelines that had reuses outside of the reach of Explorer-2 (if any) are fed to Explorer-3 which triggers DP 100 M instructions before the detailed region. This iterative process is stopped as soon as the whole set of key cachelines has been covered. We find that a small number of Explorers is typically enough — our implementation considers at most four Explorers — and for many benchmarks and detailed regions, we find that only a few Explorers are needed, as we will quantify in the evaluation section.

The vicinity reuse distances are recorded as previously described in Section 2.3, i.e., a sparse set of randomly chosen memory accesses are selected and their next (forward) reuse distance is recorded using watchpoints. Once a reuse within the vicinity has been recorded, the corresponding watchpoint is removed and its reuse distance is recorded.

Analyst: Finally, Explorer-N feeds the obtained key reuse distances and vicinity distributions to the Analyst. This final evaluation pass uses DSW to predict the impact of warming on the detailed region.

The evaluation in the detailed region could be functional cache simulation (e.g., to determine the number of hits and misses) or detailed cycle-accurate processor simulation (e.g., to determine an application's IPC on a particular microarchitecture configuration).

3.3 Implementation Details

Time Traveling enables DeLorean to collect all the key reuse distances and the vicinity distributions while limiting the number of watchpoints that need to be triggered. By doing so, DeLorean maintains high accuracy while improving evaluation speed by an order of magnitude compared to the state-of-the-art.

gem5 and KVM: We implement DeLorean using KVM [16] and the gem5 cycle-accurate full-system simulation infrastructure [8]. DeLorean switches and exchanges full-system state between KVM and gem5 at the boundaries of each detailed region. For example, at the end of a warm-up interval, KVM transfers the entire system state to gem5 after which gem5 takes over to conduct cycle-accurate simulation of the detailed region; similarly, the state is transferred back from gem5 to KVM at the end the detailed region. As DeLorean uses full-system simulation, to align the location of the detailed regions across the different passes, we count the number of dynamically executed user-space instructions [33].

Explorers: We develop multiple instances of gem5: Scout, Explorers and Analyst. We use 4 Explorers to profile 5 M, 50 M, 100 M and 1 B instructions before each detailed region. The actual implementation of directed profiling (DP) varies across the different Explorers. Explorer-1 profiles a relatively short interval of 5 M instructions for the full set of key cachelines. We use functional simulation to implement DP in Explorer-1 using gem5's so-called 'atomic' CPU model. For the other Explorers, we use *virtualized directed profiling (VDP)* which sets watchpoints by leveraging the operating system's page protection mechanism, as previously explained in Section 2.3. This is implemented in KVM [16]. VDP advances between watchpoints at near-native speed. A watchpoint stops the native execution when there is an access to a protected page, at which point the reuse distance is computed if the watchpoint is a true positive (i.e., a reuse). To minimize the overhead from false positives, and to achieve overall high evaluation speed with TT, we limit VDP to Explorer-2 through 4, and use functional simulation for Explorer-1.

In addition to collecting key reuse distances, all Explorers also collect 'vicinity' reuse distances. These reuses are randomly collected with a sample rate of 1 over 100 k memory instructions — we quantify the impact of the sample rate on accuracy and evaluation speed in the results section. The same sample rate is used by all Explorers.

RSW versus DSW: There is a subtle but important difference when implementing RSW and DSW using watchpoints and VDP. As previously discussed in Section 2.3, RSW sets watchpoints at random memory locations. Once a reuse is detected, the watchpoint is removed. RSW however needs to collect many reuses per load PC to accurately predict warming misses for the particular load PCs in the detailed regions.

In DSW on the other hand, watchpoints for the key reuse distances are not set at random memory locations[1]. Instead, watchpoints are set at specific memory locations, namely the key cachelines. For each key cacheline, DSW needs to collect the nearest previous reuse in the warm-up interval, i.e., the key reuse distance. This implies that the watchpoints need to be on during the entire warm-up interval to compute the last reuse for each key cacheline. The overhead for collecting key reuse distances is very high, up to the point that it negates the benefit from having to collect fewer reuse distances under DSW compared to RSW. The reason for the high overhead is that DSW detects many reuses for a single key cacheline out of which it only needs the last one. Hence, although DSW needs to collect few key reuse distances, the overhead for collecting them in a naive implementation is high.

This is why TT is needed to translate the small number of key reuse distances under DSW into a substantially higher evaluation speed compared to RSW. Employing a multi-pass approach allows for progressively reducing the number of key cachelines to track in the different Explorers: Explorer-1 sets watchpoints for all key cachelines for a short interval; Explorer-2 then sets watchpoints for a smaller number of cachelines (i.e., the remaining key cachelines after Explorer-1) for a longer interval; follow-on Explorers set even fewer watchpoints for progressively longer intervals. By doing so, DeLorean limits the number of watchpoints that need to be set while being able to collect all key reuse distances. This translates into a substantial improvement in evaluation speed compared to RSW.

Design Space Exploration: It is interesting to note that the overhead due to warm-up can be amortized across multiple parallel simulations, which is particularly appealing when performing design space exploration studies, as we will demonstrate and quantify in the evaluation section. In particular, DeLorean allows for running multiple detailed evaluations concurrently and warm them all up from the same warm-up. In TT terminology, this means there is a single Scout and a single set of Explorers that feed a number of Analysts, with each Analyst simulating a different cache configuration or processor architecture configuration. As soon as Explorer-N has reached the beginning of a detailed region, control is transferred to the different Analysts to simulate the different cache and processor configurations in parallel.

This is possible because the reuse distance which underpins DSW is independent of the underlying cache hierarchy. Hence, we need to collect the reuse distances in the warm-up interval only once, after which we can predict warming misses in the detailed region for a range of cache and processor configurations. Collecting the reuse distance warm-up information is thus amortized across multiple parallel detailed evaluations.

4 GENERAL APPLICABILITY

The core contributions in DeLorean — DSW and TT — build upon statistical cache modeling which has been demonstrated for a range of architectures and configurations, including (i) cache replacement policies, (ii) multiprogrammed workloads, and (iii) multi-threaded

[1]Watchpoints for collecting the vicinity distribution are set at random memory locations, however, an order of magnitude fewer reuse distances need to be collected than for RSW, as mentioned before.

execution. We hence believe that DeLorean is more generally applicable beyond the concrete implementation in this paper which considers LRU cache replacement and single-threaded execution. (In fact, the evaluation section includes one example to illustrate DeLorean's wider applicability.)

4.1 Cache Replacement

The very first work demonstrating how sparse (approximate) reuse distance distributions can be used to statistically model caches considered caches with random replacement [6]. Follow-on work demonstrates similar accuracy for LRU caches [11]. More recent work extends statistical cache modeling to cover other replacement algorithms, including pseudo-LRU and NMRU [25]. Sen and Wood [27] suggest that stack distance — which can be estimated using the reuse distance distribution — can be used to model other replacement algorithms as well. Beckmann and Sanchez [2] propose probabilistic methods to model age-based replacement policies, such as RRIP. This body of prior work makes us confident that techniques to model other replacement algorithms are already available or can be designed.

4.2 Multi-Programming

StatCC [10] is a method in which sparse reuse information, collected separately for each application, can be used to model how several independent applications in a multi-programmed workload interact when sharing a cache. StatCC recursively uses a simplistic CPU performance model to determine the CPI for each application in the workload mix. The per-application's reuse information is scaled according to its CPI, which results in a new CPI for each application to be used in the next iterative CPI estimation. After a few iterations already, a stable solution is found. Combining StatCC with DeLorean will likely improve the accuracy of StatCC by replacing the simplistic CPI estimation with DeLorean's detailed simulation to estimate the per-application CPI for the next iteration.

4.3 Multi-Threading

StatCache-MP [7] shows how sparse reuse information, collected from a single execution of multiple threads, can be used to model the execution of multi-threaded applications across a wide selection of different cache sizes and cache topologies with MSI coherence. StatCache-MP models both constructive and destructive cache sharing in a way that fits DeLorean's approach very well. If a key access by thread A during detailed simulation is known to be preceded by a write to the same memory location by another thread B, and threads A and B do not share the cache in the modeled topology, detailed simulation should model a coherence miss for thread A. However, if both threads share the modeled cache, a constructive sharing cache hit should be modeled — provided that the reuse distance between the two accesses is short enough, otherwise a capacity cache miss should be modeled. We believe that this approach can be further extended to model other coherence states (e.g., O and E states) but this is left for future work.

4.4 ISA Extensions and Accelerators

DeLorean leverages hardware virtualization which is widely supported across ISAs. Research into ISA extensions can be handled through emulation of unimplemented instructions in KVM. Workloads that offload (a) part(s) of their execution to an accelerator can also leverage DeLorean. One can fast-forward the execution and then, for the portion of the workload that is executed on the accelerator, switch to detailed simulation with statistical modeling of the on-chip caches using DeLorean to reduce simulation time.

	gem5's default OoO x86 CPU	
Pipeline	ROB	192 entries
	IQ	64 entries
	SQ	64 entries
	LQ	64 entries
	Issue	8 wide
Branch Predictor	Tournament	2 bit choice counters, 8 k entries
	Local	2 bit counters, 2 k entries
	Global	2 bit counters, 8 k entries
	BTB	4 k entries
Caches	L1-I	64 KiB, 2-way LRU, 64 B line
	L1-D	64 KiB, 2-way LRU, 64 B line
	LLC	1 MiB to 512 MiB, 8-way LRU, 64 B line
	MSHRs	4 (L1-I), 8 (L1-D), 20 (LLC)

Table 1: Simulated processor architecture.

5 EXPERIMENTAL SETUP

We perform full-system gem5 simulations of a 64-bit x86 superscalar 8-wide out-of-order processor with split 2-way 64 KiB L1 instruction and data caches and a unified 8-way LLC with sizes ranging from 1 MiB to 512 MiB. A summary of the important simulation parameters can be found in Table 1. We use gem5 revision 2c9f7ebca: we use the default superscalar out-of-order model as our timing model in the detailed regions with the 'classic' memory system. We use KVM hardware virtualization on the simulation host machine to fast-forward between detailed regions at near-native hardware speed. We run Ubuntu 12.04.5 LTS running Linux 3.2.44 in full-system simulation.

We consider the SPEC CPU2006 benchmarks with reference inputs in the evaluation[2]. All benchmarks were compiled with GCC 4.6.3 and optimization flag `-O2`. All simulations are started from the same checkpoint of a booted system that has executed 100 B instructions. Evaluation speed is measured on a dual-socket Intel Xeon E5520 with 4 cores per CPU and 2 threads per core, running at 2.26 GHz.

Due to the high overhead of the reference experiments, we use 10 detailed regions spread uniformly across 10 B instructions (1 B instructions apart). For each detailed region, we use gem5's out-of-order CPU model and run for 10,000 instructions. Prior research shows that the highest accuracy is achieved for small detailed regions [34]; larger detailed regions will likely make DeLorean even more accurate since small regions make the penalty for mispredicting the outcome of a single key access high. Prior to each detailed region, we warm up microarchitecture state (processor pipeline, caches, branch predictor) for 30,000 instructions.

[2]We were unable to run 403.gcc, 433.milc., 447.dealII, 481.wrf and 482.sphinx3 because these benchmarks either produced outputs that could not be verified with the reference input or did not run to completion.

6 EVALUATION

We now evaluate DeLorean's speed and accuracy. DeLorean's key advantage over the state-of-the-art is its significantly improved evaluation speed while delivering similar accuracy. We also present a sensitivity analysis and several use cases. We consider the following sampling strategies in the evaluation:

- SMARTS: Functional Warming (FW) is used to keep the caches warm using functional simulation in-between detailed regions as done in SMARTS [34].
- CoolSim: Randomized Statistical Warming (RSW) is employed to collect randomly selected reuse distances at near-native hardware speed in-between detailed regions. This is the state-of-the-art statistical cache warming strategy called CoolSim [23]. We pick the best possible configuration for CoolSim, which encompasses an adaptive sampling strategy: sample one memory location every 40k memory instructions for the first 750M instructions, then one every 20k for the next 200M instructions, and finally one every 10k for the last 50M instructions.
- DeLorean employs DSW to determine the key reuse distances which are collected through TT.

6.1 Speed

Figure 5 reports simulation speed normalized to SMARTS. DeLorean improves simulation speed by 96× on average compared to SMARTS. Compared to CoolSim, DeLorean improves simulation speed by 5.7× on average. We note there is some variation across benchmarks in simulation speedup compared to CoolSim. The highest speedup (49×) is reported for bwaves: this benchmark features a small number of key accesses, and the corresponding key reuse distances are small so that they can all be profiled by Explorer-1. This implies that DeLorean essentially fast-forwards through most of the benchmark. CoolSim on the other hand collects a large number of reuse distances which are not needed to accurately model cache warm-up. The smallest speedups are reported for povray (1.05×) and gems (1.4×). The reason is slightly different though for the two benchmarks. gems features a large working set and key accesses with very long reuse distances; hence, it engages all four Explorers. The working set size for povray is much smaller but there is one detailed region with a few key accesses with very long reuse distances, which engages all four Explorers. These long reuse distances, in addition, incur a large number of false positive watchpoint triggers. This is due to the use of the page protection mechanism to collect the reuse distances at near-native hardware speed, as previously described, i.e., cachelines with large reuses map into the same physical page as cachelines with very short reuses, making the collection of those longer reuses expensive due to false positives.

These simulation speedups lead to high absolute simulation speeds. We report that DeLorean achieves a simulation speed of 126 MIPS, compared to 21.9 MIPS for CoolSim and 1.3 MIPS for SMARTS. In other words, DeLorean is within one order of magnitude compared to native hardware execution. This high simulation speed enables a new range of experiments to be run with much longer running, and hence more realistic, workloads than the relatively small workloads that are typically run with detailed cycle-accurate simulation.

The reason for DeLorean's high simulation speed is a result of both DSW and TT, the two key contributions in this paper. We provide deeper insight where the improved simulation speed is coming from in the next two subsections.

6.1.1 DSW. Directed statistical warming (DSW) substantially reduces the number of reuse distances that need to be collected during the warm-up interval compared to RSW. Figure 6 quantifies the total number of collected reuse distances across the 10 detailed regions (note the logarithmic scale): DSW reduces the number of reuse distances by 30× on average (and up to 6,800×) compared to RSW. Whereas RSW collects 340,000 reuse distance on average, DSW collects 11,000 reuse distances. The reason is that DSW collects a select number of reuse distances, namely the key reuse distances and the vicinity reuse distances, whereas RSW collects a much larger number of random reuse distances.

6.1.2 Time Traveling. Time traveling (TT) translates the reduction in reuse distances collected during warm-up through DSW into a significant evaluation speedup. Recall that the number of engaged Explorers depends on the number of key accesses and their reuse distances. Figure 7 breaks down the key reuse distances as they are collected by the respective Explorers. Most key reuse distances are collected by Explorer-1, however, additional Explorers are engaged for a number of benchmarks. Figure 8 quantifies the average number of Explorers engaged. The number of Explorers varies across the benchmarks depending on how long the reuse distances are. For example, bwaves features short key reuse distances, hence the number of Explorers needed is small, even less than one on average (vast majority of memory operations hit in the lukewarm cache or MSHRs). In contrast, benchmarks such as zeus, cactus, gems and lbm feature a relatively large number of long reuse distances, hence they require up to four Explorers on average. A couple benchmarks have few long reuse distances across all detailed regions, see for example mcf, gromacs, leslie3d, sjeng and astar, hence they also engage a relatively large number of Explorers. calculix is an exception having a relatively large number of long reuse distances, yet the number of Explorers is relatively small; the reason is that the long reuse distances originate from a single detailed region, hence we need to engage up to four Explorers only for a single detailed region and not the other regions.

6.2 Accuracy

Figures 9 and 10 quantify DeLorean's simulation accuracy for predicting CPI for two cache sizes to reflect a modern-day LLC size (8 MiB) as well as a large-scale DRAM cache (512 MiB), respectively. Note that SMARTS is our reference here because of the full cache warming done in-between detailed regions. DeLorean predicts CPI with an average error of 3.5% and 2.9% for the 8 MiB and 512 MiB LLCs, respectively. DeLorean is substantially more accurate than CoolSim for soplex and gems. The average error for CoolSim equals 9.1% and 9.3% on average for the 8 MiB and 512 MiB LLCs, respectively. The reason for the high error for CoolSim for soplex and gems is a result of an overestimation of the number of LLC misses.

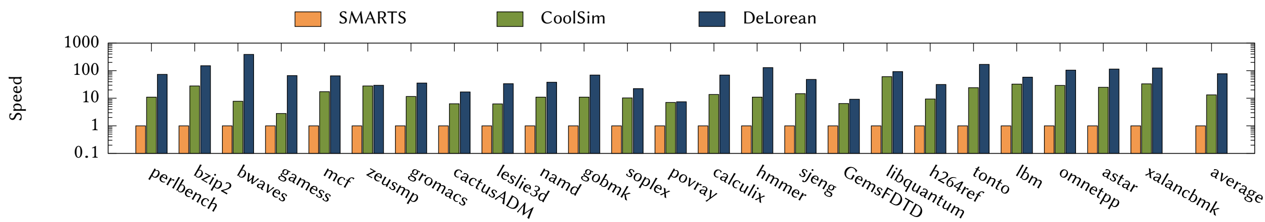


Figure 5: Normalized simulation speed for DeLorean, CoolSim and SMARTS. *DeLorean improves simulation speed by* 96× *on average compared to SMARTS, and by* 5.7× *compared to CoolSim.*

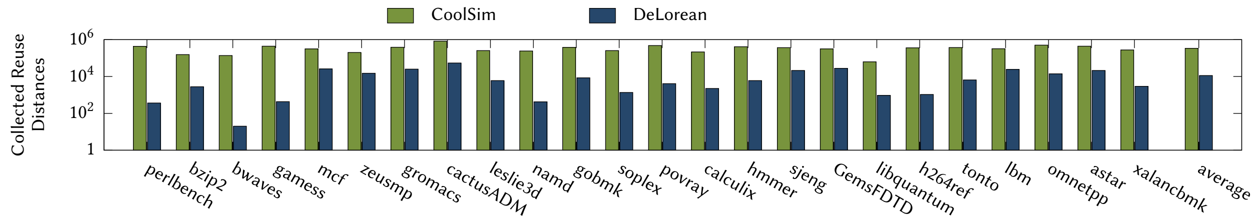


Figure 6: Number of reuse distances collected by CoolSim (RSW) versus DeLorean (DSW). *DeLorean reduces the number of reuse distances by* 30× *on average compared to CoolSim.*

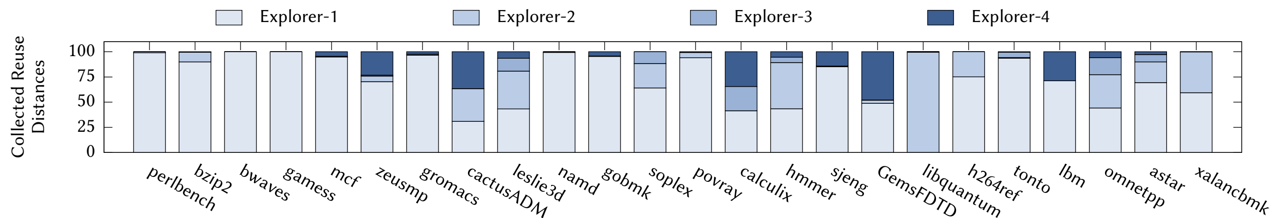


Figure 7: Key reuse distances as they are collected by the different DeLorean Explorers. *Most key reuse distances are collected by Explorer-1, however, for a few benchmarks we need to engage additional Explorers, up to Explorer-4.*

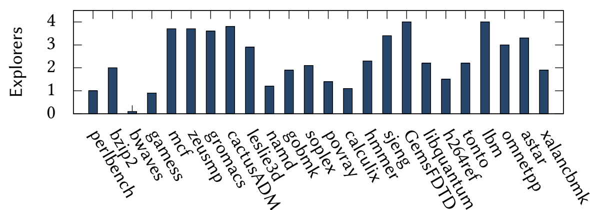


Figure 8: Average number of Explorers to collect the key reuse distances with DeLorean. *The number of Explorers varies across benchmarks depending on how many key reuse distances are needed and how long the reuse distances are.*

6.3 Sensitivity Analyses

We now consider two sensitivity analyses related to the vicinity distributions and hardware prefetching.

6.3.1 Vicinity Reuse Distance Distribution. Recall that DeLorean samples the reuse distance distribution in the vicinity as input to statistical cache modeling. The sampling rate is a parameter that can be freely set. The default sampling rate is set to 1 out of 100 k memory instructions. We now evaluate DeLorean's sensitivity in terms of simulation speed and accuracy with respect to sampling density in the vicinity. Increasing sampling density improves accuracy by collecting more reuse distances. On the other hand, higher density also increases the profiling overhead of the Explorers. Figure 11 quantifies this trade-off in simulation speed versus accuracy for an 8 MB LLC. With a density of 1 over 100 k memory instructions, we can simulate at 126 MIPS with an error of 3.5%. Increasing density to 1 over 10 k instructions brings down the error to 2.2% at a simulation speed of 71.3 MIPS.

6.3.2 Hardware Prefetching. As argued in Section 4, DeLorean is broadly applicable because it builds upon statistical cache modeling which has been demonstrated for a range of different architectures and configurations. We now consider hardware prefetching to illustrate DeLorean's broader applicability. Hardware prefetching improves performance by speculatively fetching cachelines before the application actually needs it. Hardware prefetch requests are typically triggered by cache misses, i.e., cache misses with particular patterns (e.g., a stride) trigger prefetch requests. We extend DeLorean to trigger the hardware prefetcher using misses as predicted by the statistical cache model. In other words, DeLorean feeds the hardware prefetcher with *predicted* cache miss information rather than *actual* (simulated) cache miss information. Likewise, prefetch requests to cachelines that *are* in the cache already (in detailed

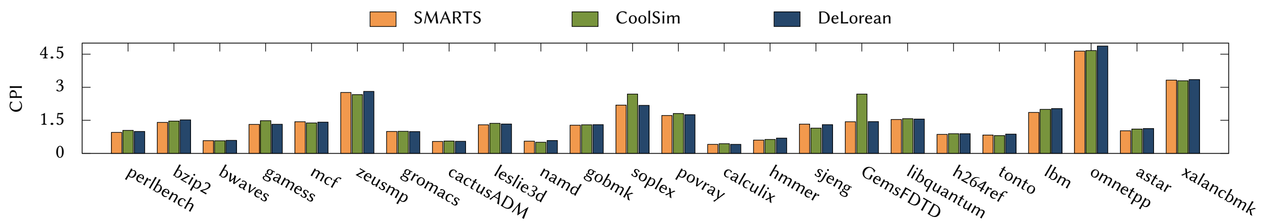


Figure 9: Reported CPI for DeLorean, CoolSim and SMARTS (reference) for an 8 MB **LLC.** *DeLorean's simulation accuracy is within* 3.5% *on average for CPI compared to SMARTS.*

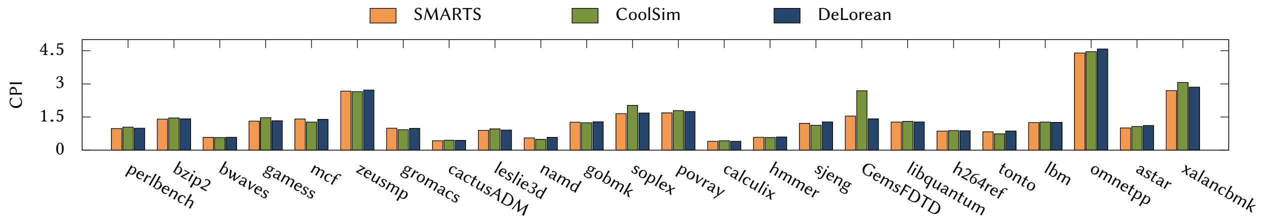


Figure 10: Reported CPI for DeLorean, CoolSim and SMARTS (reference) for a 512 MB **LLC.** *DeLorean's simulation accuracy is within* 2.9% *on average for CPI compared to SMARTS.*

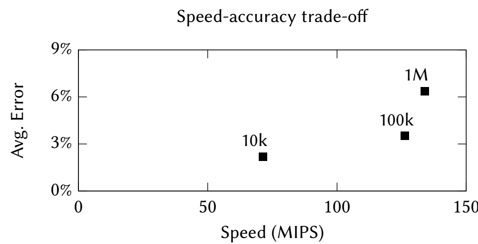


Figure 11: Speed-accuracy trade-off for an 8 MB **LLC by changing the sampling density in the vicinity.** *A denser vicinity increases accuracy, but results in reduced simulation speed.*

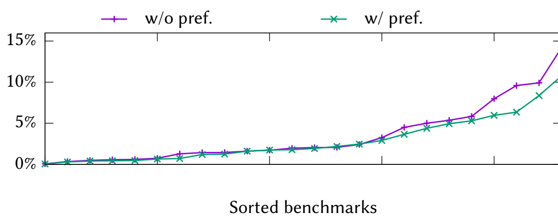


Figure 12: CPI error with and without prefetching for an 8 MB **LLC.** *DeLorean is slightly more accurate for an architecture with hardware prefetching.*

simulation) versus *predicted to be* in the cache (for DeLorean) are nullified to save memory bandwidth. Figure 12 reports CPI error for our baseline processor with and without an LLC stride prefetcher with 8 streams. We conclude that DeLorean is slightly more accurate for an architecture with hardware prefetching because there are fewer cache misses to be predicted in the first place.

6.4 Use Cases

Having demonstrated the speed and accuracy of DeLorean, we now consider two case studies to illustrate its usage in practical design studies. We consider application working set characterization and design space exploration.

6.4.1 Working Set Curves. Working set curves are widely used to characterize an application's working set size. A working set curve shows cache miss rate (or MPKI) as a function of cache size, and typically incurs a point (cache size), or multiple points, at which the miss rate falls off. This is commonly referred to as the 'knee' of the curve. This knee indicates the working set size of the application at hand.

Figure 13 provides a number of interesting examples, i.e., cactus, leslie3d and lbm; the solid line shows the reference curve while the dashed line shows the curve obtained through DeLorean. (We observe similar results for the other benchmarks; not provided because of space constraints.) The key observation is that DeLorean tracks the reference curves obtained using SMARTS well. For example, lbm has a knee in the curve around 8 MiB and 512 MiB, which is accurately predicted by DeLorean; cactus and leslie3d, on the other hand, do not have a pronounced knee in the curve, which is also accurately predicted by DeLorean.

6.4.2 Design Space Exploration. DeLorean is a fast simulation methodology, significantly speeding up design space exploration studies. Figure 14 shows performance curves (CPI) as a function of cache size for the same set of benchmarks as in Figure 13. DeLorean tracks the reference (SMARTS) accurately and accurately predicts performance sensitivity to LLC size.

Note that all 10 points in Figure 14 were obtained from the same warm-up in a parallel simulation run. As explained in Section 3.3, warm-up cost can be amortized across multiple parallel simulations by feeding multiple parallel Analysts from a single Scout and a single set of Explorers. Collecting reuse distances takes up most of the

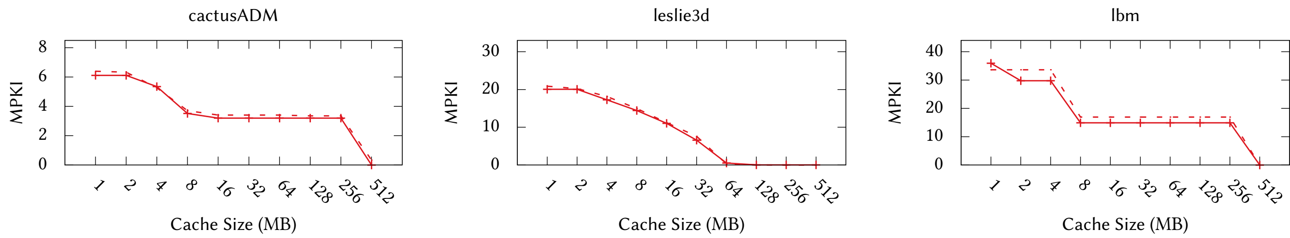


Figure 13: Working set curves for three example benchmarks. The reference (SMARTS) is shown as a solid line whereas DeLorean is shown as a dashed line. *DeLorean tracks the reference working set curves well.*

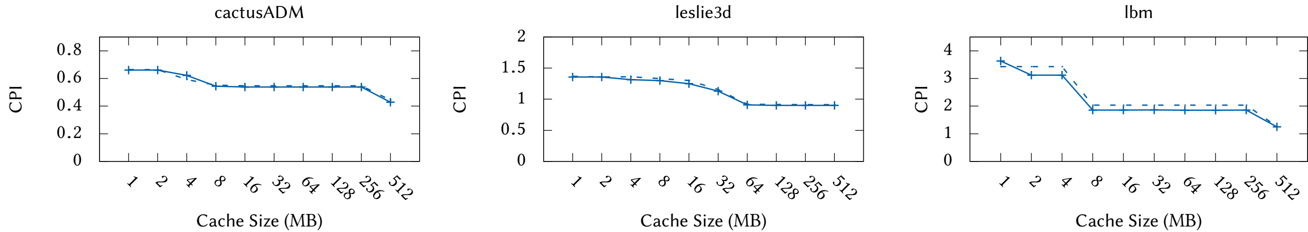


Figure 14: Performance (CPI) as a function of cache size for three example benchmarks. The reference (SMARTS) is shown as a solid line whereas DeLorean is shown as a dashed line. *DeLorean tracks the reference curves well.*

time and most of the simulation host resources compared to timing simulation of the detailed regions — the time spent in warming versus detailed simulation is about a factor 235× for DeLorean. The marginal cost for parallel simulation in DeLorean is thus small in terms of required simulation resources — less than 1.05× for 10 parallel Analysts. This is much smaller than the 10× marginal cost for parallelizing 10 detailed simulations as is commonly done.

7 RELATED WORK

Checkpointed Warming. One popular approach to the cache warm-up problem in sampled processor evaluation is checkpointed warming, which takes a checkpoint of the microarchitecture state prior to each detailed region. Wenisch et al. [32] store the state of the caches and other micro-architectural structures in checkpoints. These so-called *Flex points* eliminate warming overhead in SMARTS. Flex points are large (20 MiB to 100 MiB). In follow-up work, they introduce *Live points* [33] and reduce the space requirements for each checkpoint down to 142 KiB. Van Biesbrouck et al. [30] propose a similar approach called memory hierarchy state (MHS) to minimize checkpoint size, in the context of SimPoint [28]. Barr et al. [1] propose the Memory Timestamp Record (MTR), a method to record memory patterns, compress and store them for use in checkpoints for sampled simulation of multi-threaded workloads on multi-processor system. Hassani et al. [14] use sampled simulation combined with MTR and in-memory checkpoints to evaluate benchmarks in a few seconds by simulating detailed regions in parallel. A major limitation of checkpointed warming is that is not re-usable across software changes. In contrast, DeLorean does not incur any storage overhead and is re-usable across software and hardware changes.

Functional Warming. Instead to storing a checkpoint on disk, functional warming warms up microarchitecture state on the fly by simulating microarchitecture structures prior to each detailed region. Functional warming does not incur any storage overhead and is re-usable across software changes. Traditionally, functional warming warms up microarchitecture state using all memory references between two consecutive detailed regions, which is very slow [34]. Various approaches have been proposed to reduce the warm-up length prior to each detailed region. Haskins and Skadron [12] and Luo et al. [19] use heuristics to find the minimum number of instructions needed to warm a cache of specified size. Haskins and Skadron [13] 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 that provides a large enough cumulative distribution of MRRLs is used as the warming interval. Eeckhout et al. [9] introduce the *Boundary Line Reuse Latency (BLRL)* which extends the MRRL concept. They apply similar heuristics to find a shorter warm-up period. Van Ertvelde et al. [31] extend on the concept of BLRL using a form of hardware state checkpoints. Sandberg et al. [26] propose a method that uses two parallel simulations, pessimistic and optimistic, to bound the maximum error due to warming. While minimizing the number of instructions needed to warm up microarchitecture state improves evaluation speed, all of these functional warming techniques suffer from the inherent limitation that they need to simulate *all* memory references in the warm-up interval. In other words, even though the interval is shortened, these techniques still need to simulate all of them, and most of these references are not needed to accurately warm up the cache hierarchy. In contrast, DeLorean limits the number of warm-up references that need to be inspected, dramatically improving warm-up efficiency.

Statistical Cache Modeling. Stack distance analysis has been extensively used to model caches. Mattson et al. [20] use a simple stack algorithm to inspect every access and collect stack distance information. To improve performance, researchers use k-ary [4] and AVL [24] trees instead of linked lists. However, all of the proposed methods have to inspect all memory accesses and measure stack

distance. Other works [29] have proposed hardware-accelerated stack distance collection. Liu and Mellor-Crummey [17] 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 [17].

A major limitation of stack distance analysis is that measuring stack distances is computationally demanding because all references between two reuses of the same cacheline need to be inspected to compute the number of unique cachelines. Eklov and Hagersten [11] demonstrate how reuse distance, which is computationally less demanding to collect, can be used to predict stack distance. Reuse distance based statistical cache modeling was successfully demonstrated to model caches with different replacement policies [25], multi-programming workloads [10] as well as multi-threaded workloads [7]. Randomized statistical warming leverages statistical cache models to predict which memory references in a detailed region are warming misses. As extensively argued in this paper, randomized statistical warming requires a large number of reuse distances, many of which are useless to accurately predict warm-up effects during sampled evaluation. Directed statistical warming dramatically reduces the number of reuse distances that need to be collected.

8 CONCLUSION

Sampling allows for realistic workload evaluation by reducing the number of instructions that need to be evaluated in detail. Unfortunately, warming caches dominates simulation overhead and prevents simulation frameworks from realizing a proportional reduction in simulation time while maintaining flexibility to make changes in software and hardware.

DeLorean delivers a substantial simulation speedup compared to the state-of-the-art through two key innovations: directed statistical warming and time traveling. Directed statistical warming collects a select number of key reuse distances and their vicinity distributions — a reduction by 30× compared to the state-of-the-art CoolSim. Time traveling measures these key reuse distances in a single evaluation run by first quickly looking into the future (through virtualized fast-forwarding) to determine the key cachelines, and then going back in time to compute the key reuse distances and vicinity distributions at near-native hardware speed (through virtualized directed profiling). Time traveling translates the reduction in reuse distances that need to be profiled into a simulation speedup of 5.7× compared to CoolSim. Warm-up cost can be amortized across multiple parallel simulations when conducting design space exploration studies. DeLorean also improves simulation accuracy: prediction error is reduced from around 9% for CoolSim to around 3% on average. Ultimately, DeLorean enables detailed cycle-accurate gem5 simulations at a speed of 126 MIPS.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback. This work was supported by the Swedish Foundation for Strategic Research CoDeR-MP project and the Swedish Research Council Linnaeus centre of excellence UPMARC. The simulations were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at UPPMAX and NSC. Additional support is provided through European Research Council (ERC) Advanced Grant agreement no. 741097, and FWO grants no. G.0434.16N and G.0144.17N. This work was also supported by a Start-up Grant from the National University of Singapore (NUS).

REFERENCES

[1] Kenneth C. Barr, Heidi Pan, Michael Zhang, and Krste Asanovic. 2005. Accelerating Multiprocessor Simulation with a Memory Timestamp Record. In *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 66–77. https://doi.org/10.1109/ISPASS.2005.1430560

[2] Nathan Beckmann and Daniel Sanchez. 2016. Modeling Cache Performance Beyond LRU. In *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, 225–236. https://doi.org/10.1109/HPCA.2016.7446067

[3] Samson Belayneh and David R. Kaeli. 1996. A Discussion on Non-blocking/Lockup-free Caches. *ACM SIGARCH Computer Architecture News* 24, 3 (June 1996), 18–25. https://doi.org/10.1145/381718.381727

[4] Brian T. Bennett and Vincent J. Kruskal. 1975. LRU Stack Processing. *IBM Journal of Research and Development* 19, 4 (July 1975), 353–357. https://doi.org/10.1147/rd.194.0353

[5] Erik Berg and Erik Hagersten. 2004. StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE Computer Society, 20–27. https://doi.org/10.1109/ISPASS.2004.1291352

[6] Erik Berg and Erik Hagersten. 2005. Fast Data-Locality Profiling of Native Execution. In *Proc. Internationcal Conference on Measuring and Modeling of Computer Systems (SIGMETRICS)*. ACM, 169–180. https://doi.org/10.1145/1064212.1064232

[7] Erik Berg, Håkan Zeffer, and Erik Hagersten. 2006. A Statistical Multiprocessor Cache Model. In *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE Computer Society, 89–99. https://doi.org/10.1109/ISPASS.2006.1620793

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 Simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[9] Lieven Eeckhout, Yue Luo, Koen De Bosschere, and Lizy K. John. 2005. BLRL: Accurate and Efficient Warmup for Sampled Processor Simulation. *Computer* 48, 4 (July 2005), 451–459. https://doi.org/10.1093/comjnl/bxh103

[10] David Eklov, David Black-Schaffer, and Erik Hagersten. 2010. StatCC: A Statistical Cache Contention Model. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 551–552. https://doi.org/10.1145/1854273.1854347

[11] David Eklov and Erik Hagersten. 2010. StatStack: Efficient Modeling of LRU Caches. In *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE Computer Society, 55–65. https://doi.org/10.1109/ISPASS.2010.5452069

[12] John W. Haskins and Kevin Skadron. 2001. Minimal Subset Evaluation: Rapid Warm-Up for Simulated Hardware State. In *Proc. International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*. IEEE Computer Society, 32–39. https://doi.org/10.1109/ICCD.2001.955000

[13] John W. Haskins and Kevin Skadron. 2003. Memory Reference Reuse Latency: Accelerated Warmup for Sampled Microarchitecture Simulation. In *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE Computer Society, 195–203. https://doi.org/10.1109/ISPASS.2003.1190246

[14] Sina Hassani, Gabriel Southern, and Jose Renau. 2016. LiveSim: Going Live with Microarchitecture Simulation. In *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, 606–617. https://doi.org/10.1109/HPCA.2016.7446098

[15] Mark D. Hill and Alan J. Smith. 1989. Evaluating Associativity in CPU Caches. *IEEE Trans. Comput.* 38, 12 (Dec. 1989), 1612–1630. https://doi.org/10.1109/12.40842

[16] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. KVM: The Linux virtual machine monitor. In *Proc. Linux Symposium*. 225–230.

[17] Xu Liu and John Mellor-Crummey. 2013. Pinpointing Data Locality Bottlenecks with Low Overhead. In *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE Computer Society, 183–193. https://doi.org/10.1109/ISPASS.2013.6557169

[18] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 190–200. https://doi.org/10.1145/1065010.1065034

[19] Yue Luo, Lizy K. John, and Lieven Eeckhout. 2004. Self-Monitored Adaptive Cache Warm-Up for Microprocessor Simulation. In *Proc. Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE Computer Society, 10–17. https://doi.org/10.1109/SBAC-PAD.2004.38

[20] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irvine L. Traiger. 1970. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal* 9, 2 (June 1970), 78–117. https://doi.org/10.1147/sj.92.0078

[21] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 89–100. https://doi.org/10.1145/1250734.1250746

[22] Nikos Nikoleris, David Eklov, and Erik Hagersten. 2014. Extending Statistical Cache Models to Support Detailed Pipeline Simulators. In *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE Computer Society, 86–95. https://doi.org/10.1109/ISPASS.2014.6844464

[23] Nikos Nikoleris, Andreas Sandberg, Erik Hagersten, and Trevor E. Carlson. 2016. CoolSim: Statistical Techniques to Replace Cache Warming with Efficient, Virtualized Profiling. In *Proc. Symposium on Systems, Architectures, Modeling, and Simulation (SAMOS)*. IEEE Computer Society, 106–115. https://doi.org/10.1109/SAMOS.2016.7818337

[24] Frank Olken. 1981. *Efficient Methods for Calculating the Success Function of Fixed Space Replacement Policies*. Master's thesis. Lawrence Berkeley Laboratory, University of California. https://doi.org/10.2172/6051879

[25] Xiaoyue Pan and Bengt Jonsson. 2015. A Modeling Framework for Reuse Distance-based Estimation of Cache Performance. In *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE Computer Society, 62–71. https://doi.org/10.1109/ISPASS.2015.7095785

[26] Andreas Sandberg, Nikos Nikoleris, Trevor E. Carlson, Erik Hagersten, Stefanos Kaxiras, and David Black-Schaffer. 2015. Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed. In *Proc. International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, 183–192. https://doi.org/10.1109/IISWC.2015.29

[27] Rathijit Sen and David A. Wood. 2013. Reuse-based Online Models for Caches. In *Proc. Internationcal Conference on Measuring and Modeling of Computer Systems (SIGMETRICS)*. ACM, 279–292. https://doi.org/10.1145/2465529.2465756

[28] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 45–57. https://doi.org/10.1145/605397.605403

[29] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. 2009. 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)*. ACM, 121–132. https://doi.org/10.1145/1508244.1508259

[30] Michael Van Biesbrouck, Brad Calder, and Lieven Eeckhout. 2006. Efficient Sampling Startup for SimPoint. *IEEE Micro* 26, 4 (July 2006), 32–42. https://doi.org/10.1109/MM.2006.68

[31] Luk Van Ertvelde, Filip Hellebaut, Lieven Eeckhout, and Koen De Bosschere. 2006. NSL-BLRL: Efficient CacheWarmup for Sampled Processor Simulation. In *Proc. Annual Symposium on Simulation*. IEEE Computer Society, 168–177. https://doi.org/10.1109/ANSS.2006.33

[32] Thomas F. Wenisch, Roland E. Wunderlich, Babak Falsafi, and James C. Hoe. 2005. TurboSMARTS: Accurate Microarchitecture Simulation Sampling in Minutes. *ACM SIGMETRICS Performance Evaluation Review* 33, 1 (June 2005), 408–409. https://doi.org/10.1145/1071690.1064278

[33] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. 2006. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro* 26, 4 (July 2006), 18–31. https://doi.org/10.1109/MM.2006.79

[34] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. 2003. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *Proc. International Symposium on Computer Architecture (ISCA)*. ACM, 84–97. https://doi.org/10.1145/859618.859629