Faster Functional Warming with Cache Merging

GUSTAF BORGSTRÖM, Uppsala University
CHRISTIAN ROHNER, Uppsala University
DAVID BLACK-SCHAFFER, Uppsala University, Sweden

SMARTS-like sampled hardware simulation techniques achieve good accuracy by simulating many small portions of an application in detail. However, while this reduces the detailed simulation time, it results in extensive cache warming times, as each of the many simulation points requires warming the whole memory hierarchy. Adaptive Cache Warming reduces this time by iteratively increasing warming until achieving sufficient accuracy. Unfortunately, each time the warming increases, the previous warming must be redone, nearly doubling the required warming. We address re-warming by developing a technique to merge the cache states from the previous and additional warming iterations.

We address re-warming by developing a technique to merge the cache states from the previous and additional warming iterations. We demonstrate our merging approach on multi-level LRU cache hierarchy and evaluate and address the introduced errors. By removing warming redundancy, we expect an ideal 2× warming speedup when using our Cache Merging solution together with Adaptive Cache Warming. Experiments show that Cache Merging delivers an average speedup of 1.44×, 1.84×, and 1.87× for 128kB, 2MB, and 8MB L2 caches, respectively, with 95-percentile absolute IPC errors of only 0.029, 0.015, and 0.006, respectively. These results demonstrate that Cache Merging yields significantly higher simulation speed with minimal losses.

CCS Concepts: • **Computer systems organization** → *Serial architectures*; • **Computing methodologies** → **Discrete-event simulation**; **Simulation environments**.

Additional Key Words and Phrases: functional warming, cache warming, cache merging

1 INTRODUCTION

Computer architects rely on simulators for evaluation and experimentation. However, as simulating is highly time-consuming, a range of techniques have been developed to provide trade-offs in accuracy and speed. On one end, analytical approaches use simplified models for speed but at a loss of precision [8, 11, 14]. Conversely, cycle-accurate simulations use detailed models of the full system but are orders of magnitude slower. Sampled simulation improves the performance of cycle-accurate simulation while controlling the loss of accuracy. SimPoint [19] and SMARTS [22] are the two most common sampling approaches. SimPoint identifies the samples that are needed to represent the overall behavior accurately. By simulating a relatively small number of such SimPoints and weighing them according to their relevance, an accurate result can be obtained with much less simulation. SMARTS [22] simulates sufficiently many uniformly distributed samples to represent the whole simulation statistically. The benefit of this approach is that the sampling error can be statistically bound, and each sample can be much shorter than those of SimPoint. However, this means that the simulation time it takes to move between samples now dominates, which is the focus of this work.

Figure 1 shows how SMARTS achieves speedup by replacing most of the slow, detailed simulation (red) with much faster *functional simulation* (yellow). Functional simulation allows SMARTS to fast-forward to the next point where a slow, detailed simulation sample is required. The faster functional simulation keeps simulation structures that do not need detailed simulation, such as caches and branch predictors, up to date or *warmed*. As a result, its contribution to the simulation is called *functional warming*. However, since functional warming does not keep detailed structures warmed,

Authors' addresses: Gustaf Borgström, Uppsala University, gustaf.borgstrom@it.uu.se; Christian Rohner, Uppsala University, christian.rohner@it.uu.se; David Black-Schaffer, Uppsala University, Department of Information Technology, Lägerhyddsvägen 1, Uppsala, 752 37, Sweden, david.black-schaffer@it.uu.se.

such as the pipeline, scheduler, ROB, etcetera, a short amount of additional detailed simulation called *detailed warming* is executed immediately before the samples. The detailed warming and simulation are each in the order of thousands of instructions, while billions to tens of billions are executed in functional warming. Using SMARTS, Wunderlich et al. show a 0.64% CPI error trade-off for a simulation speedup up to 60× faster than always running in detailed simulation mode.

With SMARTS, most time is spent in the faster functional mode warming the cache (yellow in Figure 1). Reducing the time spent warming speeds up simulation but risks hurting accuracy if the cache is insufficiently warmed. Warming reduction has been extensively investigated [4, 5, 9, 15, 21]. Adaptive Cache Warming [2] addresses this by iteratively increasing the warming for each sample until the detailed simulation reaches a given desired accuracy. (See Figure 1, bottom.) Unfortunately, Adaptive Cache Warming's iterative approach results in re-warming the earlier warming amount at each iteration when it increases the warming time.

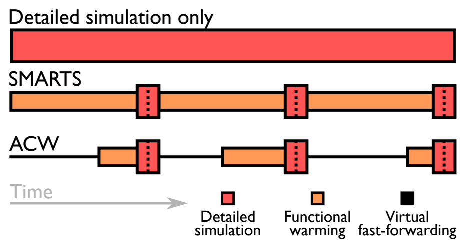


Fig. 1. *Simulation modes (slow detailed vs. faster functional) with SMARTS and Adaptive Cache Warming.* The dashed line in the detailed simulation shows detailed warming vs. detailed simulation.

This work addresses the re-warming overhead of Adaptive Cache Warming's iterative warming increases by *merging* the new warming with the previously warmed cache state. Our Cache Merging avoids re-warming on each iteration and reduces simulation time but opens up new challenges with correctly merging the previous and newly warmed cache states.

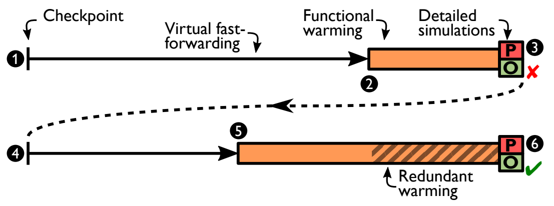
2 HOW ADAPTIVE CACHE WARMING IMPROVES SMARTS

Adaptive Cache Warming [2] (ACW) dramatically improves the performance of SMARTS by dynamically identifying the minimum amount of warming needed for each simulation sample. ACW achieves this with a simulate-and-evaluate process, where the warming amount increases iteratively, followed by an evaluation that determines whether the warming is sufficient for accurate simulation results. This approach improves performance by replacing constant warming with dynamically adjusted warming, resulting in a 6.9 − 18× average speedup (depending on cache size) over SMARTS with a fixed 100M cycle warming before each sample.

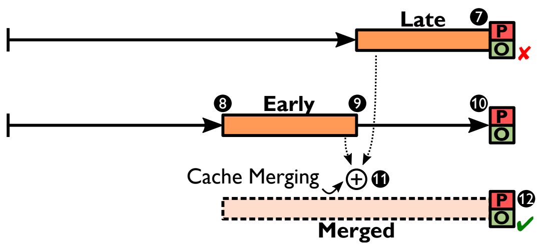
Figure 2a illustrates how ACW iteratively finds the correct warming. It starts with an in-memory checkpoint of the simulated application far before the sample (❶). ACW then "jumps" forward[1] closer to the sample (❷), where it switches to functional warming mode and starts warming the cache. When it reaches the sample, ACW does SMARTS' detailed warming and simulation (❸). However, ACW needs to assess whether the amount of warming was sufficient to trust the detailed simulation results. To do so, ACW estimates the simulation accuracy by executing a *second* detailed simulation that evaluates the impact of accesses to un-warmed cache portions. More warming is needed if there is a significant difference between these two simulations. In that case, ACW restarts from the in-memory checkpoint before the warming (❹) but with increased cache warming (❺). The iterative process repeats until reaching sufficient accuracy (❻).

To determine if the iteration's warming is sufficient (❸ and ❻), ACW tracks *cold misses*, that is, accesses that miss in a cache set that is not yet fully warmed and which might turn into hits with more warming. However, we can not know if the behavior of these accesses plays a significant role in the simulation. To answer this, ACW uses the method

[1] ACW accomplishes this using hardware virtualization (also demonstrated in previous work [7, 18]) and in-memory checkpointing, which allows it to move to different points in the simulation at near-native hardware execution speeds.



(a) *Adaptive Cache Warming.* In the first iteration (top), the simulation fast-forwards to the chosen warming point (2) and starts warming, followed by the optimistic/pessimistic detailed simulation to determine if more warming is needed. If it is, another iteration is simulated (bottom) by restarting from the checkpoint and increasing the amount of warming. As a result, the entirety of the previous iteration is redundantly re-warmed on each subsequent iteration.



(b) ACW *with Cache Merging.* As with Figure 2a, the first iteration (top) has insufficient warming, requiring a second (bottom) iteration with increased warming. With Cache Merging, we only warm the new portion of the warming (Late) in the second iteration and then merge it with the previous warming (Early), thereby avoiding the need to re-warm the portion from the previous iteration (top).

Fig. 2. Left: ACW. Right: ACW with Cache Merging.

proposed by Sandberg et al. [18], making *two* detailed simulations: a pessimistic simulation that assumes cold misses are true misses, regardless of more warming, and an optimistic simulation that assumes they should be hits with sufficient warming. As the true IPC should lie between these two estimates, ACW increases warming until their difference is sufficiently small. While this identifies the correct amount of warming, it also results in ACW re-doing the previous iteration's warming on each new iteration. If warming doubles each iteration, this results in half the warming time being redundant.

In this work, we eliminate ACW's redundant warming by warming the new portion of the simulation on each iteration and *merging* them with the warming from the previous iteration, as shown in Figure 2b. We first save the **Late** *cache state* from the previous iteration's warming ❼. We then start the additional warming ❽ as before but stop right before where the previous warming (**Late**) started. ❾ is then saved as the **Early** state. The simulation then fast-forwards to the start of the detailed warming ❿ and *merges* the **Early** and **Late** cache states to produce a **Merged** state to use with the detailed simulation. If the merging operation is correct, the **Merged** state will contain the same contents as the **Full** cache state resulting from warming the entire time, but without the need to re-warm, the **Late** portion warmed in the previous iteration. We develop the techniques needed to implement this merging for LRU caches, analyze the occurred errors, and evaluate the performance and accuracy trade-offs. Because ACW doubles the amount of warming in each iteration and as Cache Merging eliminates redundant warming, we expect to double the warming speed as half of all warming time is omitted.

3 CACHE MERGING STRATEGY

3.1 Single-level Cache Merging Strategy

The simplest example of Cache Merging is a single-level LRU cache, as the cache state is maintained solely by the LRU order of the blocks in all sets. (The multi-level cache case is more complex as the state is shared across levels, as discussed in Section 3.3.) Merging the **Early** and **Late** cache states requires correctly choosing cache blocks from the **Early** and **Late** cache states such that the final **Merged** state has the same contents as the reference **Full** state from

Exists in **Early**?	Exists in **Late**?	**Late** is filled?	Should merge?	Exists in **Merged**?
F	F	X	F	F
F	T	X	F	T
T	F	F	T	T
T	F	T	F	F
T	T	X	F	T
Conditions			Action	Result

Table 1. *Truth table for single-level cache merging.* The conditions are whether the data to be merged exists in **Early**, exists in **Late**, and if **Late** is filled yet, i.e., if there is any space left to merge to.

continuous warming would have had. For an LRU cache and whenever **Late**'s warming has not yet filled a set, we merge data in LRU order from that set in **Early** into that set in **Late**. Table 1 shows the specific merging criteria.

Figure 3 shows an example of Cache Merging for a single set. Address streams warming the **Early** and **Late** cache states are at the top. When merging, the cache blocks A, B, and D are present in **Late** and thus kept in the **Merged** state. This example accurately reflects the LRU replacement policy, as the **Late** state is later in the simulation, so its cache entries are added more recently. However, cache block C ❺ is only present in the **Early** state. As there is a remaining unfilled (cold) block in **Late** ❻ , merging will copy cache block C into the **Merged** state. More generally, merging proceeds on a set-by-set basis and adds the most recently used blocks from the **Early** cache state to any unfilled (cold) entries in the **Late** state. If a block is present in both states but in different LRU positions, Cache Merging uses **Late**'s LRU position. The resulting **Merged** cache state is then used with the detailed warming and simulation.

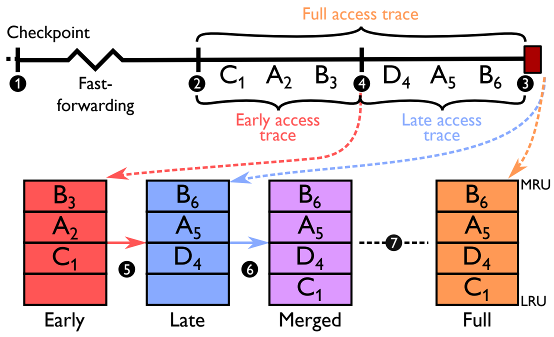


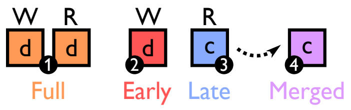
Fig. 3. *Cache Merging.* The "Early access trace" and "Late access trace" (top) are used to warm the **Early** cache state (left, red) and **Late** cache state (middle, blue). The blocks in **Early** are then merged in LRU order (5) into the **Late** (6), starting with **Early**'s MRU position and merged into un-warmed entries in **Late** in LRU order until no more blocks can be merged from **Early** or until **Late** is filled. For reference, the baseline **Full** state comparison shows that **Merged** and **Full** states have the same final cache contents (7).

3.2 Merging Dirty Blocks

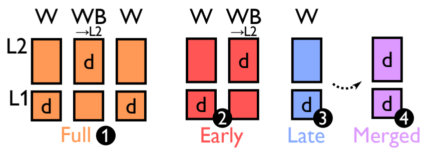
While Cache Merging for a single-level LRU cache results in the correct data blocks in the **Merged** cache state, it is not always possible to determine the correct *dirty status* for each block. Figure 4a shows an example: a write request in **Early** results in a block being marked as dirty ❷ but only accessed by a read request to the the same block in **Late** ❸ . As a result, the **Merged** value will be taken from **Late** and be clean ❹ , while in the **Full** simulation, the block would have remained dirty throughout ❶ . These *merge errors* can lead to (very) minor IPC errors from the resulting detailed simulations, as explored in Section 5.2. A valid approach could have been to set the dirty status when detecting this situation, i.e., whenever the block exists as dirty in **Early** and as clean in **Late**. However, we saw that neither case is statistically much more likely. Merge errors and their corrections are discussed further in Section 3.3. To always ensure that the correct values are used in the detailed simulation regardless of a block's status, Cache Merging will always load the latest values from main memory for every block in **Merged** and before the detailed warming starts.

3.3 Multi-level Cache Merging Strategy

Cache Merging for multi-level cache hierarchies is more complex than single-level ones as copies of blocks in different levels at different times affect data movement on accesses and evictions. While strictly inclusive or exclusive policies



(a) *Mislabeling of dirty bits due to merging in a single-level cache.* When a cache block is written (W) in the **Early** warming it is marked as dirty (2), but if it is only read (R) in the **Late** warming, it will be marked as clean (3). As the **Merged** takes the latest state from the **Late** warming, it will incorrectly mark the block as clean, when comparing (1) to (4).



(b) *Extra Block error introducing incoherence in a multi-level cache.* A write request (W), write-back to L2 (WB), and then write request again results in dirty data in **Full**$_{\mathrm{L1}}$ (1). However, when the write request and write-back occur in **Early** (2), the second write request in **Late** (3), and then finally the cache states are merged, there is a risk of merging the dirty block from **Early**$_{\mathrm{L2}}$ to **Late**$_{\mathrm{L2}}$, resulting in an *incoherent* state (4) as there are two dirty versions of the same block present.

Fig. 4. *Examples of merge errors.*

are predictable and therefore easy to handle, this work addresses a *mostly-inclusive* policy that installs data in all cache levels on read requests, but data may remain in lower caches even if evicted from higher levels. This policy causes additional complexity and uncertainty when merging, as it can result in more valid data placements across the hierarchy. The main effect of this comes from **Late**$_{\mathrm{L1}}$ being empty at the start of the **Late** warming causing accesses to it to miss. The multi-level setup installs data in the **Late**$_{\mathrm{L1}}$ cache and propagates changes further to the **Late**$_{\mathrm{L2}}$ cache, which might not have happened if L1 was warm. In more detail:

- The **Late**$_{\mathrm{L1}}$ warming starts with a cold (empty) L1 cache. As a result, all accesses miss in the L1 and are forwarded to the L2, resulting in installations in both L1 and L2. In **Full**$_{\mathrm{L1}}$ most of these accesses would hit to the L1 and be *filtered* so that they did not reach the L2. The resulting lack of filtering in a cold **Late**$_{\mathrm{L1}}$ is that blocks are installed in **Late**$_{\mathrm{L2}}$ that a warm **Full**$_{\mathrm{L1}}$ would have filtered.
- Dirty blocks in the L1 are written back to the L2 when they are evicted due to other accesses to the L1. However, by splitting the warming up into **Early** and **Late**, we often find that there are not enough accesses to **Early** to cause this block eviction to L2. At the same time, the block is absent from **Late**$_{\mathrm{L1}}$, so it is neither written back to L2 from there. As a result, the dirty data remains in L1 instead of being moved to L2.

We refer to errors stemming from a cold **Late**$_{\mathrm{L1}}$ as *cold-start effects*. These effects lead to an incorrect **Late** state with the following errors:

(1) **Extra Blocks (from warming).** Data may be present in L2 that should have been absent.
(2) **Missing Blocks (from warming).** Dirty data in **Early**$_{\mathrm{L1}}$ that should have been evicted to L2.
(3) **Dirty Status Mislabeling (from warming).** Data may be marked as dirty in **Full** but clean in **Merged** (or vice versa). This merge error may also occur in single-level Cache Merging.

In turn, the merging exacerbates the error, producing an incorrect **Merged** cache state:

(1) **Extra Blocks (from merging).** Extra Blocks are merged from **Early** to **Late**.
(2) **Missing Blocks (from merging).** Blocks are prevented from being merged by Extra Blocks that should not have been in the cache.

We refer to the union of the errors originating from warming or merging as *merge errors*. As discussed in Section 5, these merge errors can significantly impact simulation accuracy. To address this, we now discuss mitigation strategies that extend cache merging to take these effects into account more intelligently.

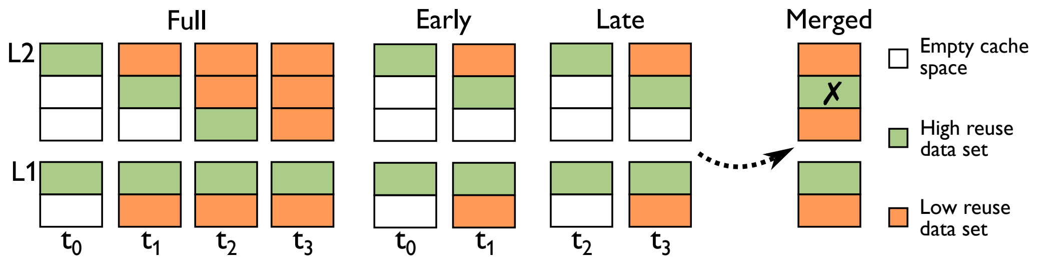


Fig. 5. *Extra- and Missing Block example.* Data can be evicted from the L2 but remain in the L1 if it is frequently enough accessed (green, high reuse) compared to other data that streams through the L1 and evicts it from the L2 (orange, low reuse). In the longer baseline **Full** this leads to the high reuse (green) data remaining in the L1 but being evicted from the L2, but in the shorter **Early** and **Late** warmings, there is not enough time to evict it from the L2. This results in the high reuse (green) data showing up as Extra Blocks in the merged L2 and low reuse (orange) data not being present (Missing Blocks).

3.4 Correcting Merge Errors

After identifying what the merge errors are and their effects, we now describe their respective origins to explore possible corrections:

(1) **Extra Block errors**
  (a) *Cold-start Effects.* Reads while warming **Late** may install data into **Late**$_{\text{L2}}$ that should not be present due to a lack of filtering from the cold **Late**$_{\text{L1}}$. Figure 5 shows an example where the low reuse data set should have evicted the block from the high reuse data set in L2, but because the **Late**$_{\text{L1}}$ does not filter the access, that block will exist as an Extra Block in **Merged**$_{\text{L2}}$.
  (b) *Merging.* Data read and later evicted from **Full**$_{\text{L2}}$ is expected to be absent. However, if the block was loaded when warming **Early**, but there were not enough accesses to evict it later, it will still be present. If there are not enough accesses to fill **Late**, there will be cold space left there such that the block will be merged from **Early**, resulting in an Extra Block in **Merged**.
  In a complementary scenario, merging Extra Blocks may lead to an incoherent cache state with a dirty block in several caches simultaneously. Figure 4b shows an example of merging a dirty block from **Early**$_{\text{L2}}$ into **Late**$_{\text{L2}}$ while the same dirty block is already present in **Late**$_{\text{L1}}$ ❹. This case is essential to address to ensure program correctness. Furthermore, this also gives a good hint about where in the hierarchy the block belongs in **Late**.

(2) **Missing Block errors**
  (a) *Cold-start Effects.* A dirty block in **Early**$_{\text{L1}}$ is supposed to be written back to L2 during **Late**'s warming but is not present when **Late** starts warming, resulting in the write-back never occurring. As a result, the block is missing from **Late**$_{\text{L2}}$ and, therefore, **Merged**$_{\text{L2}}$.
  (b) *Merging.* Extra Blocks that prevent merges from **Early**$_{\text{L2}}$ to **Late**$_{\text{L2}}$. Figure 5 shows an example where the Extra Block from a high reuse data set in **Late**$_{\text{L2}}$ prevents the correct merging of the block from the streaming data set, which will be missing from **Merged**$_{\text{L2}}$.

(3) **Dirty Status Mislabeling errors.** *Cold-start Effects.* If a write request happens in **Early** and a read request to the same data in **Late**, then the read request will miss in the cold **Late**$_{\text{L1}}$ and install *clean* data into L1 (instead of hitting to the dirty data in L1, as was the case in **Full**). The example error shown in Figure 4a is in a single-level hierarchy, but the principle is the same for multi-level hierarchies.

The ability to identify and correct these merge errors falls into the following categories:

(1) *Always correctable.* Merge errors that can be accurately detected and whose corrections are unambiguous.
(2) *Statistically correctable.* Merge errors whose detection or correction can be ambiguous but where the outcome is heavily biased. Here we can apply the statistically more likely correction for a better overall outcome but may introduce other false-positive errors.
(3) *Statistically non-correctable.* Merge errors whose correction is ambiguous and not heavily biased in a particular direction cannot be corrected without introducing more errors than they address.
(4) *Undetectable.* Merge errors that can not be accurately detected, e.g., which blocks would be missing from **Merged** after merging.

Correcting the merge errors, therefore, depends on whether or not one or more valid alternatives are possible and, if so, whether one of them is significantly more likely to occur (we address this further in Section 5.3.)

3.5 Merging Invalidated Blocks

During warming and merging, we must distinguish between *cold* blocks (never accessed during warming) and *invalidated* blocks (accessed but later invalidated). This distinction is because invalidated blocks from **Late** should survive into **Merged** and not be filled with blocks from **Early**, while cold ones should be filled. Figure 6 illustrates this, where a write request installs a block copy in both L1 and L2 ❶. However, as the write is exclusive in the L1, the gem5 simulator immediately invalidates all copies except L1's copy ❷. Evicting the block from the L1 should write it back to the L2. On eviction from the L1, this block should be written back to the L2. However, since we should not merge over invalidated blocks, we need to be careful that such write-backs prioritize replacing invalid blocks in the L2 ❹, or we will cause subsequent merges to be incorrect ❸. This distinction leads to two policies:

- *During Warming:* Write-backs from L1 are first installed into invalidated blocks in L2 before writing to cold space ❹.
- *During Merging:* Only merge into cold blocks in **Late**. Invalidated blocks are interpreted as a direct effect of the **Late** warming and retained.

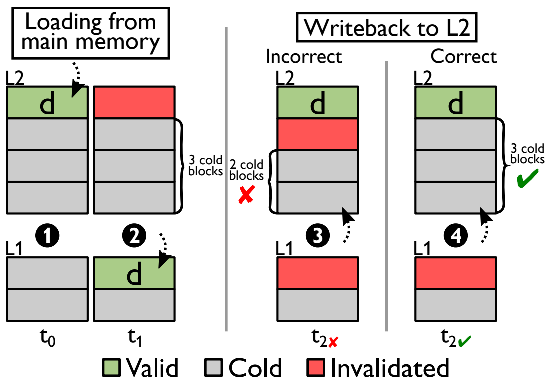


Fig. 6. *Handling invalidated blocks.* The gem5 cache policy installs data into both L2 (1) and L1 (2) on a write but immediately invalidates the block in L2. However, merging must treat invalid blocks distinctly from cold blocks as we expect to have invalid blocks in the baseline **Full** and not replace them with merged blocks from **Early**. This means that during warming we must prioritize placing writebacks into previously invalidated blocks (4) to avoid later problems with merging (3).

3.6 Cache Merging in a Multi-core Environment

When using Cache Merging with a multi-threaded program running on a multi-core setup, we need to ensure 1) program correctness with a **Merged** cache state and 2) that simulation performance measurements are accurate.

For correctness, Cache Merging must ensure not introducing data races into a data-race-free (DRF) program. DRF programs rely on atomic operations to determine which thread gets access to shared data (i.e., critical section). For example, whenever two cores want to access the critical section simultaneously, the atomic operations guarantee that only one core will have exclusive access to memory. Meanwhile, the other core will see that the same data is not accessible and wait without the risk of both cores entering the critical section simultaneously. For Cache Merging to uphold this guarantee in a multi-core environment, the values of the atomic variables in the respective caches must be correct when merging the cache states. Cache Merging ensures this by using up-to-date values loaded directly from memory (see Section 3.2). Thus, the correct atomic value will be found in the caches, and the program will proceed correctly.

For performance accuracy, merge errors may result in blocks having the wrong coherence state or being in the wrong cache. For example, if core **A** operates on the critical section, but blocks from the critical section were merged erroneously to core **B**'s cache, then core **A** will see a miss to its cache. However, the coherence protocol will move the blocks to core **A** as they are accessed. This merge error will result in a correct execution but may cause an increased memory latency and, in turn, lower IPC accuracy. However, as Cache Merging restores each cache's contents individually from the **Merged** cache state, there will be no block placement into other caches within the same cache level[2]. As a result, we do not expect such misplacements to occur frequently or impact performance, as the local hot data will likely be in the correct local cache after the merge.

3.7 Cache Merging With Alternative Cache Replacement Policies

The Cache Merging algorithm presented so far addresses a LRU replacement policy. Merging for other policies presents different challenges:

- *Random Replacement.* Cache Merging would select blocks to merge from **Early** to cold blocks in **Late** randomly, resulting in a statistically correct, but not deterministic merge. Note that other policies employing some degree of randomness in their policies will have similar behavior.
- *Not Most Recently Used (NMRU).* Cache Merging would ensure that the MRU block in **Late** is retained, but fill any cold blocks with randomly chosen blocks from **Early**, as with random replacement.
- *Not Recently Used (NRU).* This policy clears every block's MRU bit on installation and a hit. If all blocks in a set have their MRU bits cleared, then all are set. Replacements are chosen randomly from blocks whose MRU bit is set. Cache Merging would pick blocks from **Early** with their MRU bits cleared at random and merge them into cold space in **Late**.
- *DRRIP (Dynamic Re-Reference Interval Prediction)* [12]. DRRIP provides trash- and scan resistance by avoiding always marking new data as most recently used. DRRIP extends NRU by using multi-bit status values to set the eviction order. To address both thrashing and scanning, DRRIP chooses dynamically between two sub-policies by set dueling across a few sampled sets. This poses two challenges for merging: determining which policy would be applied with full warming and merging the cache states based on the policy.

[2]Multi-level merging places blocks in other caches but only concerning merges across but not within levels (see Sections 3.4 and 5.3).

- *Determining the policy.* For short warming amounts the sampled sets may not be warm enough to accurately determine which policy should be applied. In these cases, the **Early**/**Late** warming would need to be done twice, once for each of the two sub-policies, and the best-performing policy could then be selected because **Full** set-dueling would have made the same choice.
- *Merging.* The cache state's blocks are then merged in decreasing status value order. However, as the invariant in DRRIP is to increase all block's status values until at least one block has its status value at a maximum, we cannot know if a block in **Late** has a high status value because it comes from a streaming data set (inserted at a high value originally) or if it was increased as the result of an eviction of other blocks. A possible solution would be to keep track of the per-set *minimum reached status value* during the warming of **Late**. When merging with **Early**, a low minimum status value would indicate that the data in the cache had high reuse that was reset upon some eviction, while a high minimum value would indicate that this is a streaming data set.

4 EXPERIMENTS SETUP AND DESCRIPTION

We evaluated Cache Merging using the gem5 simulator [1] in full-system mode with the SPEC2006 [10] benchmark suite and input workloads for 55 benchmark-input pairs. We followed the Smarts methodology for each benchmark-input pair and took ten uniformly distributed checkpoints after skipping the first 1B instructions. After removing one faulty checkpoint, this gave us a total of 549 simulation checkpoints. For each checkpoint, we warmed for ten warming amounts from 195k to 100M instructions by multiples of 2. We applied merging to these pairs for nine merged warming amounts, with the smallest being 195k+195k=390k. We evaluated two single-level configurations (the data and instruction caches of sizes 32kB and 1MB) and three multi-level configurations (32kB/128kB, 32kB/2MB, and 32kB/8MB L1/L2 cache sizes), yielding data points from 9,882 single-level simulation experiments and 14,823 multi-level simulation experiments. A Tournament branch predictor[16] is used in the detailed simulation. To focus specifically on the cache effects of warming, we always use the same unwarmed branch predictor state. The snoop filter employed by gem5 employs to simplify cache state analysis is disabled as our benchmarks are all single-threaded. We measure accuracy as the difference in simulated IPC for the Smarts detailed simulation phase between using **Full** (continuous) warmed cache states and **Merged** cache states, each of the same effective warming size. In L1, both data- and instruction caches are merged[3]. Speedup from using Cache Merging with acw is evaluated by measuring the execution rate of different simulation phases (vFF, functional simulation, and detailed simulation) on a machine with an AMD Phenom II X6 3.2 GHz processor and 8GB main memory (we show the speedup as relative times, so we expect speedup to be roughly the same across different machines)[4]. When evaluating speedup, the smallest warming size

Frequency	2.5 GHz
F/D/R/I/W/C Widths	8 / 8 / 8 / 8 / 8 / 8
ROB/IQ/LQ/SQ	192 / 64 / 32 / 32
Int. / FP Registers	256 / 256
DRAM	SimpleMemory, 3GB, 30ns
Atomic / O3 CPU TLB entries	64 / 512
Single-level cache setups	
L1 caches, data- & instruction-	Two sizes: 32kB and 1MB 64B, 8-way, LRU, 4c
Multi-level cache setups	
L1 cache, data- & instruction setup	32kB, 64B, 8-way, LRU, 4c
L2 cache	Three sizes: 128kB, 2MB and 8MB 64B, 8-way, LRU, 6c
# of warming and simulation instructions	
acw functional warming	100M, 50M, 25M, . . . , 391k and 195k
Detailed warming and simulation	20k and 30k

Table 2. *Simulation parameters.*

[3]It is also possible to merge the page-walker caches and TLBs, but we did not explore this and they are cleared and warmed as part of the Smarts detailed warming.

[4]For performance, acw uses in-memory checkpoints from which simulations are restarted and copy-on-write when advancing from the checkpoints using hardware virtualization (KVM [13]). We do not implement this, but instead, emulate them to retrieve results for our analysis.

is 195k instructions (in contrast to when evaluating accuracy), as ACW might estimate a sufficient warming amount immediately after evaluating the first warming iteration.

To evaluate the accuracy, we first look at the IPC error for the single-level merging and compare the **Merged** and **Full** contents to identify and explore merge errors (Section 5.2). We then look at the multi-level caches, propose corrections for the more complicated multi-level merging and evaluate the impact of these corrections (Sections 5.3 and 5.4). Finally, we look at the tradeoff in speedup and accuracy from adding Cache Merging to ACW (Section 6).

5 ACCURACY EVALUATION OF CACHE MERGING

To analyze which merge error types are most common, we enumerate the possible cache state combinations of **Early**, **Late**, the resulting **Merged**, and the baseline **Full** into so-called *simulation state vectors*. Every such vector uniquely identifies how every block resides across the caches and is effective for identifying merge errors. Specifically, by counting all observed errors from our benchmarks, the errors can be classified by their cause and how to handle them. While the errors for the single-level case are so infrequent as to be essentially negligible, the multi-level case exhibits significantly more merge errors, leading to decreased accuracy. From this error analysis, we can identify which errors are *statistically correctable* and evaluate the accuracy impact of such corrections in Section 5.4.

5.1 Enumeration of Simulation States

To classify all possible merges and errors, we build a *simulation state vector* for each cache block that combines the block's status (dirty=d, clean=c, absent=-) for each cache level (L1, L2) in each warming period (**Early (E)**, **Late (L)**, **Merged (M)**, and **Full (F)**)[5] as such:

$$[\mathbf{E}_{L1}\ \mathbf{E}_{L2}]\,[\mathbf{L}_{L1}\ \mathbf{L}_{L2}]\,[\mathbf{M}_{L1}\ \mathbf{M}_{L2}]\,[\mathbf{F}_{L1}\ \mathbf{F}_{L2}]$$

For example, the simulation state vector `[dc] [-d] [dd] [-d]` indicates that

- the block is dirty in $\mathbf{Early}_{L1}$ and clean in $\mathbf{Early}_{L2}$;
- the block is absent in $\mathbf{Late}_{L1}$ but is present as dirty in $\mathbf{Late}_{L2}$;
- as the block is present in $\mathbf{Merged}_{L1}$, it means it was merged from $\mathbf{Early}_{L1}$;
- as the block is already present in $\mathbf{Late}_{L2}$ (marked dirty), the clean block in $\mathbf{Early}_{L2}$ was not merged;
- the **Merged** cache state has the block marked as dirty in both L1 and L2.
- an *incoherent* state as dirty data now exists simultaneously in more than one cache level (the **Full** state denotes that the block should only have been in L2, as it was in **Late**).

In the following sections, we use simulation state vectors for all blocks across the caches, and all applications and warming amounts to collect statistics about the types and frequencies of merges and errors.

5.2 Accuracy Evaluation of Single-level Cache Merging

To analyze the accuracy of the single-level Cache Merging strategy, we examine IPC error- and merge error statistics for both a 32kB LRU cache and a 1MB LRU cache across all of our application- and warming amount pairs. We measure simulation accuracy as the percent difference in IPC between the detailed simulation using a **Full** warmed cache state

[5]We omit the L1 instruction and page walker caches for both data and instructions as they have essentially no errors (at most 0.02% for the page walker caches).

and those using the **Merged** cache state. Out of all 9,882 simulation experiments, only 3 have an IPC error, with a maximum error of 3%, demonstrating that Cache Merging is exceptionally accurate for single-level caches.

We find the origins of these errors by studying the difference between the **Merged** and the **Full** cache states in the collected simulation state vector statistics across the simulation experiments. This analysis shows that a merge error may occur throughout the warming, where a block may be mislabeled as clean in **Merged** when it should have been dirty in **Full** (denoted as `[dc][cd]` in the corresponding simulation state vectors). Out of all possible simulation state vectors, this merge error occurs in 0.04%/0.49% of all blocks and is spread across 16%/84% of all simulation experiments in the 32kB/1MB setups, respectively, showing that the merge error is overall widespread (especially in the larger cache size), but still a rare occasion in a single-level hierarchy. Figure 4a shows an example of the events leading to this merge error. As this merge error led to only 3 out of 9,882 simulation experiments having minor IPC errors, we conclude that mislabeling seldom affects the simulation precision in a single-level cache hierarchy. This mislabeling error does not affect the IPC because while clean and dirty evictions differ (dirty data is written to L2, while clean data is not), this will still have the same latency in a single-level cache hierarchy, so there is no effect on IPC accuracy. Of the three non-zero IPC error simulation experiments we observed, one was due to the page-walker data cache snooping other caches after a miss, which hit to the L1 data cache. If the data in L1-data is clean, the gem5 PWC will not use the data in the cache and instead retrieve the data from the next memory level (main memory), while if the data is dirty in the L1 cache, a shared copy will be transferred directly to the page-walker-data cache, and thereby *reduce* latency. As IPC errors are exceedingly rare (3/9,882 in our simulation experiments), we can conclude that single-level cache merging is highly accurate.

5.3 Accuracy Evaluation of Multi-level Cache Merging

We now analyze the more complex Cache Merging strategy in a multi-level cache hierarchy (as described in Section 3.3) in a simulation setup with 32kB L1 instruction and data caches and three sizes of shared (instruction and data) L2 LRU caches: 128kB, 2MB, and 8MB. In a multi-level setup, IPC errors due to merge errors are not only due to an incorrectly set dirty status of blocks (as in the single-level setup) but also due to incorrectly absent or present blocks in the L2 cache. In particular, while incorrect dirty status blocks had little impact on simulation accuracy in the single-level case, one needs to be careful when merging dirty data in a multi-level cache hierarchy not to simultaneously place dirty data into several levels. If done incorrectly, we will have an *incoherent* cache state which could, in turn, result in undefined program behavior if the reading erroneous values later.

To address these more complex (and recurring) errors, we use the simulation state vector statistics to identify merge errors and propose *corrections* (e.g., by merging across cache levels) based on the error statistics. From the simulation state vectors across all benchmarks, we observe that most merge errors (90%) are due to a small number of erroneous merge states (15). We can then identify the merge error origin (Extra Block, Missing Block, or Dirty Status Mislabeling) for each of the 15 states and determine how correctable the error is (Always correctable, Statistically correctable, Statistically non-correctable, or Undetectable). This approach allows us to propose corrections to improve the merge results. Table 3 shows a categorized selection of these 15 states; a description of how to undertake their corrections as follows:

- *Statistically corrected.* (4 of 15) Three are *Dirty Status Mislabeling* errors where it is statistically likely that a block is present in **Late** whose dirty status needs to be switched. See example (1) in Table 3: the data loaded as dirty in **Early** was only read throughout **Late**'s warming period, thus not setting the data status. In the

Error category	Simulation state vector example Notation: [E][L][M][F]	Correction category	Ratio vs. all merge errors (128kB/2MB/8MB setup)
(1) Dirty Status Mislabeling	`[-d] [-c] [-c] [-d]`	Statistically correctable.	2.1%/50.0%/78.0%
(2) Missing Block (from warming)	`[d-] [--] [--] [-d]`	Statistically correctable.	16.4%/10.7%/5.0%
(3) Extra Block (from merging)	`[-c] [--] [-c] [--]`	Statistically non-correctable.	11.6%/2.4%/0.5%
(4) Extra Block (from merging)	`[-d] [d-] [dd] [d-]`	Always correctable.	4.0%/13.9%/6.7%
(5) Missing Block (form warming)	`[-c] [--] [--] [-c]`	Undetectable.	11.6%/2.4%/0.5%

Table 3. *Different merge errors with different causes and corrections.*

2MB/8MB L2 cache setups, the L2 data is more commonly dirty, while in the 128kB setup, clean data is more common. The reason is simply that in smaller L2 cache sizes, the data is evicted from L2 and later read as a clean copy. As the general case is that the data is still in the cache and as the data will be evicted (and read again later) from the smaller cache, it is statistically more beneficial to implement a specific correction rule for this merge error. This error is an example of where *cache size matters* for what correction decision we make, as it is 0.4/1.1/2.3× as likely in the 128kB/2MB/8MB setups that switching the block's status to "dirty" would be correct. Such cache size-dependent cases need additional motivation for how to handle them. In this case, Table 3 shows that the ratio within the experiments per cache size is prevalent in the 8MB cache setup while comparably rare in the 128kB cache setup. As we load the block values from memory before detailed simulation (see Section 3.1), an eventual dirty status set erroneously will not affect program correctness but eventually simulation accuracy. Therefore, it is more beneficial to classify this error as *Statistically correctable*.

  - The fourth statistically corrected merge error is a *Missing Block error from warming* due to **Late**'s cold start effects (e.g., dirty data in $\textbf{Early}_{\text{L1}}$ should have been evicted during **Late**'s warming but was not). Correcting this error demands *cross-level merging* from $\textbf{Early}_{\text{L1}}$ to $\textbf{Late}_{\text{L2}}$. Row (2) in Table 3 shows an example where dirty data exist simultaneously in $\textbf{Early}_{\text{L2}}$ and $\textbf{Late}_{\text{L1}}$ and is thus an incoherent state. This occurs when the L2 cache is merged incorrectly (see demonstration in Figure 4b). To correct this error, *cross-level* merging from L1 to L2 is therefore needed. In the smallest 128kB L2 cache setup, it is more common that the cross-level merge should not occur simply because the data was already previously evicted. As the general case is an eviction to L2, it is statistically correct to cross-merge the data.

- *Statistically non-corrected.* These are 2 of the 15 most common merge errors, both being *Extra Block errors from merging*. While these errors occur, the results show they are not statistically frequent enough to be beneficial to constantly correct them (or else introduce more errors than we correct). In other words, it is possible to implement specific corrections for these merge errors, but they will more likely decrease overall accuracy. Example (3) in Table 3 shows how a block is not present in $\textbf{Late}_{\text{L2}}$, but as space is available, the block is merged from $\textbf{Early}_{\text{L2}}$. According to **Full**, the merge should not have occurred in this case. However, it is at least 17× more common that the merge *should* have occurred among the cache sizes, so it is statistically more beneficial to keep the original merging strategy.
- *Always corrected.* This category is 1 of the 15 most common merge errors and is the only corrected *Extra Block error from merging*. The merge error occurs whenever dirty data is merged into **Late**, already present in another cache level. Merging blocks such that multiple dirty copies exist throughout the cache hierarchy introduces incoherence. The solution is to check *all* caches in **Late** to see if the dirty block is already present and avoid merging. This solution has no ambiguity and can, therefore, always be applied. Row (4) in Table 3 shows an example where the merged dirty data from $\textbf{Early}_{\text{L2}}$ causes incoherence due to multiple dirty data throughout the hierarchy, a error avoided by checking *the whole* hierarchy before merging dirty data from **Early**.

- *Undetectable.* These cases are either *Extra Blocks from warming* or *Missing Blocks from merging*. After finishing warming the **Late** cache state, it is impossible to determine which Extra Blocks should not have been present (compared to **Full**). Furthermore, if filling the set in **Late** with such Extra Blocks since the warming, merges from the corresponding set in **Early** cannot occur. As the merging algorithm cannot determine which blocks are Extra Blocks in **Late** or would be Missing Blocks from **Early**, this situation is impossible to resolve. Row (5) in Table 3 shows an example where a block in $\mathbf{Early}_{\mathrm{L2}}$ is not merged as the set in $\mathbf{Late}_{\mathrm{L2}}$ is filled (indicated by noting that $\mathbf{Merged}_{\mathrm{L2}}$ is empty in the simulation state vector even when **Early**L2 had a block). However, the block should have been present according to the **Full** cache state. As it is impossible to know what other blocks prevent the merge in $\mathbf{Late}_{\mathrm{L2}}$, there is no way to correct this merge error.

5.4 Results From Corrections on Multi-level Merging

Figure 7a shows the impact of the above corrections on the IPC error in multi-level cache hierarchies (both relative (left) and the absolute (right) IPC error).To simplify the visualization, we plot the 4% simulation experiments that together make up 90% of the total IPC error across all L2 cache size setups. The x-axis shows the percent of merge errors in the simulation experiment's **Merged** cache state that are among the 15 top simulation state vectors, i.e., 100% indicates that *all* of its merge errors are in the topmost. The data shows that the corrections (right) significantly improve the merging accuracy (points move down) in multi-level cache setups. The corrections considerably reduce the maximum absolute IPC errors (0.567 to 0.228) and the mean absolute IPC errors (0.024 to 0.017). Besides implemented corrections for specific topmost merge errors, yet another 12 merge errors were also fully corrected.

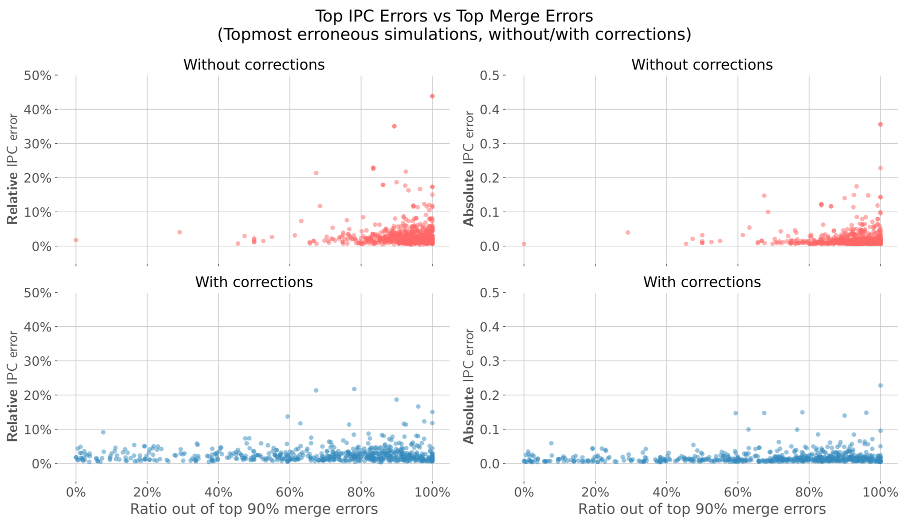
Figure 7b shows the distribution of merge errors across the 15 top state vectors (red) and the relative reduction from applying our corrections (blue). Of the four most common errors (86% of all merge errors), two are *Dirty Status Mislabeling* errors (completely corrected), one is an *Extra Block* error (completely corrected), and one is a *Missing Block* error (less than $< 0.1\%$ remaining after correction). Bars showing no difference between corrected and non-corrected merge errors are examples of *Statistically Non-corrected* and *Undetectable* errors. The *Others* bar depicts all merge errors not among the top 90%. The merge errors decreased by 19%/40%/61% after the corrections in the 128kB/2MB/8MB setups, showing how the corrections improve Cache Merging.

The top merge error *after* corrections came from misclassifying example (1) in Table 3. In the 128kB cache setup, it is 2.4× more likely that the dirty status should *not* be corrected (switched from a "clean" to a "dirty" state), while in the 8MB cache, it is 2.3× more likely that the status *should* be corrected (with a 2MB cache setup it is 1.1× more likely that the status should be corrected). As these errors make up 78% for the 8MB cache vs. 2.1% for the 128kB cache, we choose to correct the error. The rest of the errors are either *Undetectable* (and thus not corrected) or infrequent enough that we did not analyze them.

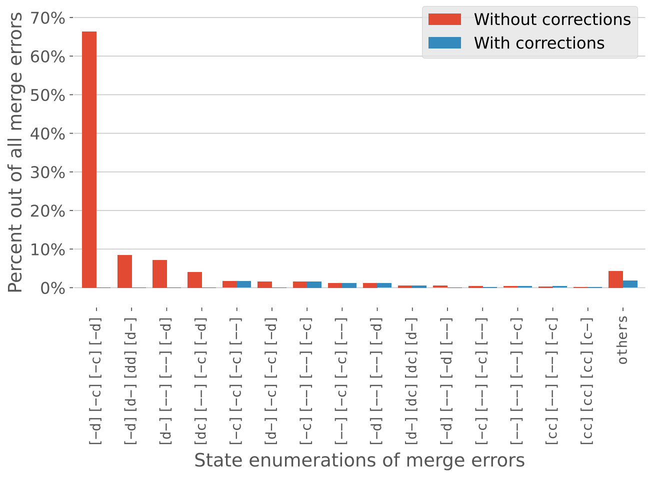
Figures 7a and 7b together show that we can successfully target and correct most merge errors and that this significantly improves simulation accuracy, particularly for those benchmark/warming combinations that show exceptionally high IPC errors.

6 USING CACHE MERGING WITH ADAPTIVE CACHE WARMING

The previous section evaluated the accuracy of merging individual pairs of **Early** and **Late** warmings. To use Cache Merging to accelerate ACW (ACW+CM), we need to investigate how well we can *cumulatively* warm the cache, i.e., by repeatedly merging with another cache state after additional warming. Specifically, as ACW doubles the amount of warming in each iteration, we may need to merge up to ten separate warmings per simulation experiment cumulatively.



(a) *Accuracy without- and with multi-level merging corrections.* The x-axis depicts how many of the simulation experiment's merge errors are in the 90% most common merge errors, and the y-axis is the simulation experiment's IPC error. The data without corrections (top) is clustered at the bottom right, indicating that most IPC errors are from the topmost common merge errors. With the corrections (bottom), we notice: 1) the overall IPC error decreases: mean by 27% (from 0.024 to 0.017), and the overall maximum was 44% without, is 22% with corrections; 2) the most common errors are less likely to be among the top errors (left shift, since many of those were corrected); and 3) there are still a significant number of most common errors (points to the right), indicating that not all were corrected.



(b) *Histogram of the impact of correcting the top merge errors, before and after.*

Fig. 7. *Correction impact from multi-level merging.*

However, as Cache Merging can introduce errors, we expect that some errors will accumulate across the merges leading to a more complex trade-off between accuracy and the 50% potential performance increase from eliminating redundant warmings. We investigate these trade-offs by looking at four metrics:

- Accuracy: The impacts of merging multiple cache states and the accumulation of merge errors.
- Accuracy: ACW+CM vs. ACW with the baseline redundant warming.
- Required Warming Estimates: ACW estimates of how much warming is required with/without Cache Merging.
- Speedup: Cache Merging's impact on the amount of warming (and hence performance) of ACW.

6.1 Removal of Trivially Error-Free Simulations

Previously we have included data from all simulation experiments when analyzing accuracy to give a general overview of the simulation- and merge error impact. However, when determining the accuracy and speedup over ACW, many simulation experiments with long warming lengths (relative to the cache sizes) result in thoroughly warmed **Late** cache states. That much warming results in no merged blocks from **Early**; therefore, **Late** is identical to **Merged**. This results in no blocks being merged from **Early**, and therefore **Late** is identical to **Merged**. To avoid biasing the ACW error analysis by such merge-less simulations, we use the baseline ACW to determine the required warming for every checkpoint and *filter* out simulations that require more warming from our metrics. The black line in Figure 8 shows the importance of this filtering: nearly all warmings of more than 12.5M instructions (for a 128kB cache) and over 70% of them for the 2MB and 8MB caches do not result in merges. If included in the error analysis, the filtered values would heavily bias our results by the configurations not used in ACW and where no merging occurs.

6.2 Analysis of Merging Cache States Cumulatively

In every ACW iteration, the new additional **Early** cache state is merged with the **Merged** cache state from the prior ACW iteration(s). If Cache Merging was perfect, such cumulative merging would give the same result as a single merge of an **Early** and **Late** pair. However, Cache Merging introduces merge errors that accumulate with every iteration and merge, likely leading to higher overall merge errors than in the pairwise merges analyzed earlier. This accumulation of errors occurs because once a merge places a cache block into the **Merged** cache state, blocks from additional **Early** states will not replace it as all blocks already inside **Merged** will be younger. This accumulation results in any blocks merged incorrectly in prior iterations will not change, so merge errors will accumulate with increasing warming until the **Merged** cache state is filled. Besides affecting simulation accuracy, this may also lead to ACW misestimating the amount of warming needed.

We analyze the impact of cumulative merging on accuracy by comparing the accuracy from simulation experiments using cache states merged cumulatively to those merging only a single pair of states. Figure 8 shows the average IPC error comparison from cumulative (red) and non-cumulative (blue) merging for the three cache sizes. As expected, cumulative merging has a higher IPC error regardless of L2 cache size and warming amount, except for the smallest merged amount, as nothing is merged cumulatively at that amount. We see the most significant difference in the 128kB cache size at 50M warming, where cumulative warming yields a 0.036 mean IPC error vs. 0.016 for non-cumulative. While cumulative merging increases the simulation error, it is clear that it remains very close to the baseline, particularly for larger cache sizes.

Finally, the smaller cache sizes have more significant inaccuracy, as seen by the different y-axis ranges chosen for each plot. We hypothesize that this effect is because the larger caches are less filled at merging than smaller caches.

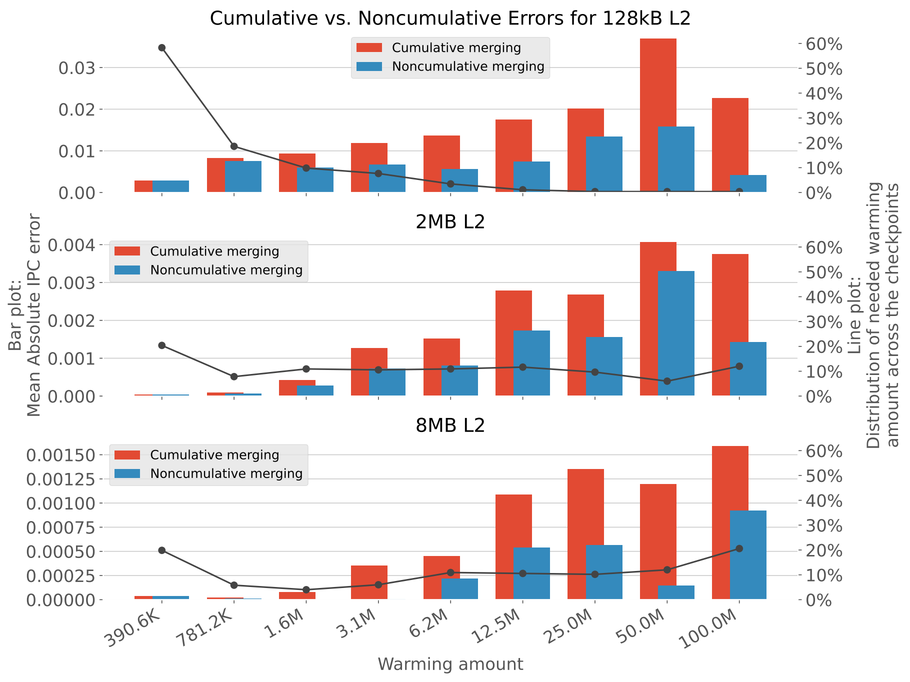


Fig. 8. *Simulation accuracy (IPC error) across warming amounts for cumulative and non-cumulative merging.* Cumulative merging sees a higher IPC error as it accumulates merge errors across multiple merges (bar plot, red vs. blue, left axis). The line (right axis) shows the distribution of warming required for each cache size to show the importance of filtering warmings that do not lead to merges. For example, for the 128kB cache, essentially all warmings over 12.5M are filtered out as they are unnecessary for such a small cache, and including them would bias the results towards the cases where no merging occurred.

This less filling results from ACW stopping when *sufficient* data is present in the cache for an accurate simulation, as opposed to when the cache is *filled*. It is easier to merge correctly in a large (more empty) cache than in a small (more filled) cache. For example, the number of Extra Blocks may be proportionally higher in a smaller cache than in a larger cache, which prevents merging blocks from **Early**, leading to a proportionally higher number of Missing Blocks.

6.3 Accuracy Analysis of Cache Merging with Adaptive Cache Warming

ACW uses the IPC results from simulation experiments for two purposes: first, to determine how much warming is needed, and second, to report the final simulation results once meeting the required warming. Errors in the IPC estimate stemming from merge errors can thus affect both the simulation results and the estimated warming amount. If ACW+CM *overestimates* the warming (i.e., *over-warming*), the result is a loss in performance (extra time spent warming) and accuracy, as any (or both) of the optimistic/pessimistic simulation's IPC may be different such that the final IPC result may not match the ACW's warming estimate or/and the reported IPC with that warming estimate. Conversely, an *underestimated* warming (i.e., *under-warming*) estimate may increase simulation errors due to the under-warmed cache. To investigate these effects, we look at the IPC accuracy of ACW+CM for both warming estimates and merging, the accuracy of the warming estimates themselves when using Cache Merging, and, finally, whether the accuracy losses come from choosing the wrong warming estimate (but not from Cache Merging) or from Cache Merging, even if estimating the correct warming[6].

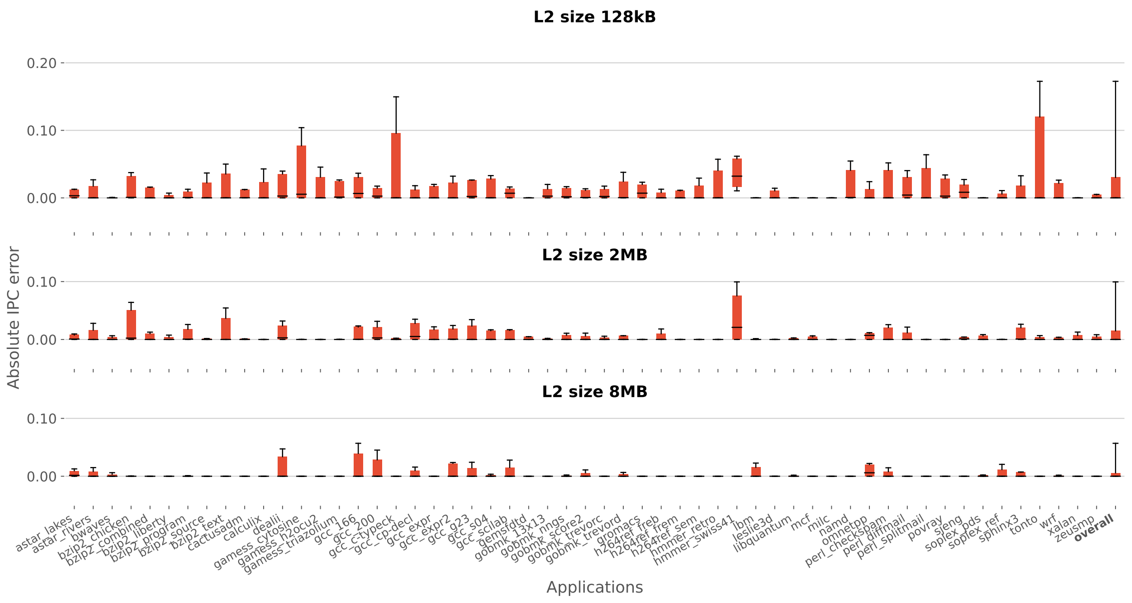


Fig. 9. *Absolute IPC error when using Cache Merging with ACW.* The boxes show a 95th percentile range with whiskers at maximum value. The "overall" bar shows that the 95th percentile IPC error is only 0.03/0.02/0.01 for the 128kB/2MB/8MB cache sizes.

	Mean	95%	Max.
128kB	0.006	0.029	0.173
2MB	0.003	0.015	0.099
8MB	0.001	0.006	0.057

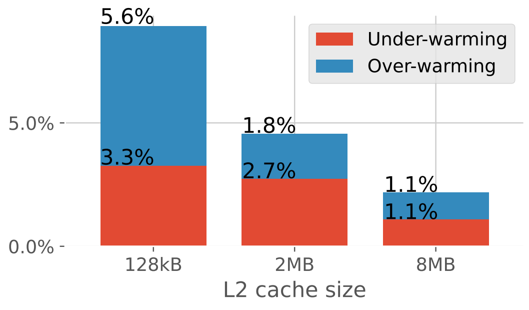
Table 4. *Mean, 95th percentile, and maximum IPC absolute error between ACW+CM and ACW simulations.*

6.3.1 Overall Accuracy. Figure 9 shows the IPC error between simulation experiments with ACW and ACW+CM and their respective warming estimates. Table 4 summarizes the overall numbers, showing that adding Cache Merging to ACW has little significance on mean IPC error (0.006% mean/0.173% max to 0.001% mean/0.057% max) across the different cache sizes. Furthermore, a comparison between the 95th percentile and the maximums shows that the "tails" of the error distributions are reasonably long (the differences being 0.144/0.084/0.051) for all cache sizes, showing that higher errors are uncommon. Analysis of simulation experiments having a significantly higher error than others (the maximum measurement being `tonto` in the 128kB cache setup, whose maximum IPC error reaches 0.173) shows that an absolute IPC error > 0.1 occurs only in less than 1% of the 128kB cache cases and none of the larger caches.

6.3.2 Over- and Under-Warming. Merge errors may also affect ACW+CM's ability to estimate how much warming is required accurately. ACW can be particularly sensitive to this as it uses a 0.01 IPC threshold (between optimistic and pessimistic estimates) to determine if sufficient warming is reached, meaning an IPC error of 0.01 caused by merge errors can result in over- or under-warming, e.g., when ACW+CM estimates a higher or lower warming amount than what the reference ACW estimate would have been.

[6]We do not present results for the last analysis as the error was so small as to be uninteresting.

Figure 10a shows the total percentage of simulation experiments that were over- or under-warmed when using ACW+CM. The 128kB cache configuration sees nearly twice a high rate of over- or under-estimated warming compared to the larger configurations. This effect reflects the same as discussed earlier in Section 6.2: smaller caches may have a higher proportion of merge errors than larger caches. In turn, the optimistic- and pessimistic simulation estimates are more prone to errors with smaller caches, leading to a higher probability of over- and under-estimated warming.



(a) *Percent over- and under-warmed checkpoints with ACW+CM.*

	128kB	2MB	8MB
A.mean absolute IPC error from			
• ...over-warming	0.02	0.01	0.01
• ...under-warming	0.02	0.01	0.01
Maximum absolute IPC error from			
• ...over-warming	0.17	0.03	0.01
• ...under-warming	0.06	0.03	0.05
G.mean speedup within			
• ...over-warmed checkpoints	0.36×	0.23×	0.14×
• ...under-warmed checkpoints	3.59×	3.17×	7.13×

(b) *IPC error and speedup within over- and under-warmed checkpoints from ACW+CM simulation experiments.*
Over **all** checkpoints, under- and over-warming yields a speedup loss is $-0.01\times$ in the 128kB cache setup and none in the larger caches.

Fig. 10. *Frequency (a) and impact (b) of over- and under-warming in ACW due to merging.*

6.4 Speedup of Cache Merging with Adaptive Cache Warming Across Applications

As Cache Merging avoids re-warming, we expect ACW+CM to spend half as much time warming as ACW, but to have the overhead of the merge itself and the fast-forwarding from the end of the added **Early** warmings to the simulation point (previously, those fast-forwardings were not needed as warming was done continuously from the start of the new warmings to the simulation point.) To compute the speedup, we collect the average execution rate of the different simulation modes (vFF, functional warming, detailed warming, and detailed simulation) and the merging itself and compute the expected execution time for each benchmark[7]. For reference, merging cache states for the 8MB cache took roughly as much time as simulating 12k instructions in functional simulation mode.

Figure 11 shows the mean simulation time across the checkpoints per application. Notably:

- ACW+CM is faster than ACW for 52.8%/93.3%/94.0% of all checkpoints in the 128kB/2MB/8MB setups. In particular, the larger cache sizes see more benefit because the baseline can avoid larger re-warmings. The smallest cache size is only faster in 52.8% of cases because 42% of the checkpoints need only the minimum 195k warming, meaning no speedup is possible, while 19% need only 390k, e.g., two iterations. This case significantly limits the potential for speedup as there are few opportunities to merge, and because the warming amounts are small, the relative overhead of the merging vs. the saved warming time is low. For the 2MB/8MB cache setups, only 21%/20% need ≤390k instructions, so there is more potential for benefit from merging. The overhead from ACW+CM also becomes smaller as the amount of warming needed increases.
- The geometric mean speedup is 1.44×/1.84×/1.87× for the 128kB/2MB/8MB cases, demonstrating that for the larger cache sizes, Cache Merging enables us to achieve nearly the full 2.0× speedup potential.

[7]We exclude 8 out of 14,823 simulation experiments from the results that crashed during vFF.

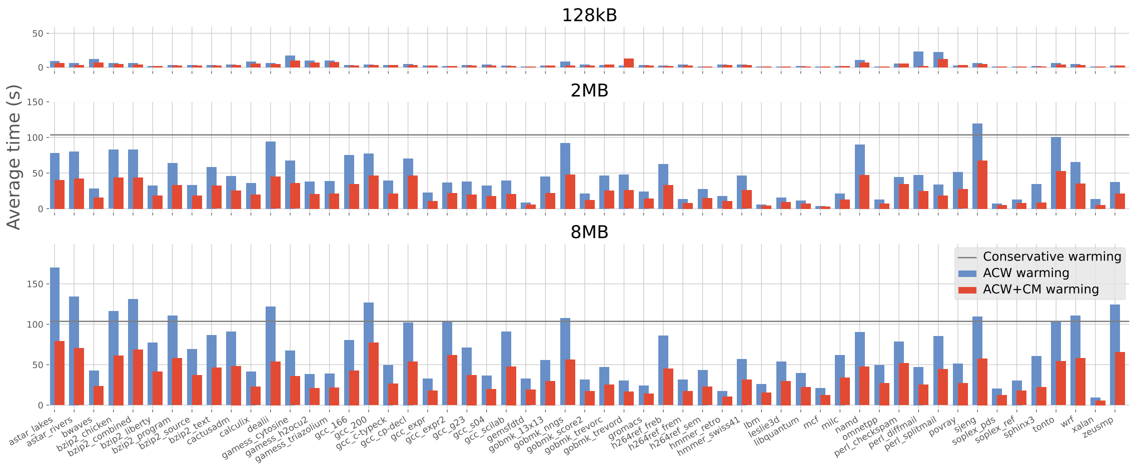


Fig. 11. *The mean time of using ACW+CM vs ACW per application.* ACW defines 100M instructions as the *conservative warming* amount, i.e., sufficient for any checkpoint, whose warming time is shown as a gray line for reference.

- ACW had the problem that a warming estimate equal to the *conservative warming* amount needed 2× longer time than simply warming conservatively, i.e., very warming demanding applications would lead to slowdowns. Cache Merging solves this problem as the redundant warming in that iteration is removed, so ACW+CM can never be slower than conservative warming. Figure 11 shows an example in the 8MB graph and the `astar/lakes` application. The gray line at the top denotes the time it takes for conservative warming to run. While the average runtime for ACW is higher than this line (i.e., it takes longer time to run than conservative warming), ACW+CM's runtime will never be at risk of running for longer than conservative warming.

7 RELATED WORK

We elaborate on other work, related to making detailed simulation faster or to determining the sufficient warming amount, that Cache Merging could be applied beneficially.

With SimPoint [20], Sherwood et al. identified the most important phases in a benchmark and provide weights for combining results from simulation experiments of just those phases to produce an accurate overall result. SimPoint is typically used with extensive functional warming for each phase (often over 100M instructions) followed by a long detailed simulation of the phases themselves (often 100M to 1B instructions). While Cache Merging could be used in combination with ACW to reduce the initial functional warming, the majority of the simulation time will still be dominated by the much slower detailed simulation itself.

Memory Reference Reuse Latency (MRRL) [9] and Boundary Line Reuse Latency (BLRL) [5] determine the amount of warming needed statistically by tracking *reuse distances* (RDs, i.e., the number of distinct accesses between two accesses to the same block) both before and during the detailed simulation. MRRL collects statistics of the reuse distances observed during warming to determine how much warming would be needed to cover a particular percent of all uses. BLRL extends this strategy by only looking at the reuses that cross between the warming and the detailed simulation, as those are the only ones that directly affect the simulation experiment. Similarly, No-State-Loss (NSL) [3] creates an LRU stack of the latest access to any address during the warming. LRU caches are then warmed using this LRU stack before detailed simulation. Later work has extended MRRL and BLRL with NSL, showing how warming periods can be shortened even further when specifically assuming LRU caches [4, 21].

Iterative search and merge methods similar to what we proposed for ACW could be used for these techniques by merging **Early** and **Late** *histograms* instead of cache states. Reuse histograms are created by recording the reuse distance between two accesses to a cache block during warming. However, there are two challenges to merging histograms. First, as we do not have the **Late** access trace when merging, reuses that *cross* **Early** and **Late** are not captured. This could be solved by a hybrid cache/histogram approach that records tags and timestamps of accesses during **Late**'s warming along with the histogram and then uses this information to detect reuses that cross from **Early**. The second challenge is that we must determine when the longest reuse distance so far is sufficiently long, i.e., when we have reached sufficient warming for good simulation accuracy. One approach could be to pair accesses during detailed simulation with the cache's contents from warming to estimate whether additional warming is needed upon misses (similar to *Delorean* [17]).

SMA [15] counts "cold start references" to the cache similarly to the way ACW counts "cold set misses". The difference is that SMA continuously keeps track of cold start references *during warming* and uses that to start the detailed simulation as soon as it determines the cache is sufficiently warm. As pointed out by the authors, the drawback is that the user can not pick the precise point for the sample, which is possible with ACW. In turn, SMA does not report any warming estimate that requires explicit searching, so Cache Merging will not be helpful with SMA.

Delorean [17] warms a fully associative cache by fast-forwarding directly to the sample (using hardware virtualization), applies detailed warming (30,000 instructions for short-term accesses), and then scan for *key accesses* to memory during detailed simulation. The blocks installed from these key accesses that were not found in the cache after detailed warming are found by searching backward in the simulation in an incremental fashion (similarly to ACW) until all key accesses are found. When all key accesses are found, their corresponding reuse distances are compiled into a histogram. The profiles from the short-term warming and key access search are then used with additional statistics from detailed simulation in a statistical cache model (StatStack [6]) to estimate simulated performance (IPC and MPKI). The authors present a speedup of 150× over SMARTS at an average 3.89% error. As the short-term warming is fixed and the list of key accesses is searched for incrementally until all are found, there is no need to merge.

8 CONCLUSIONS

This work has demonstrated how Cache Merging allows us to merge previously warmed ones with newly warmed cache states to avoid redundantly re-warming when we increase the amount of cache warming before a sample. Cache Merging is beneficial when spending much of the simulation time re-warming caches. We used Cache Merging to improve the simulation speed of Adaptive Cache Warming, which dynamically adjusts the amount of warming based on the sample. Our results show that merging does introduce errors in the cache state but that these errors have a minor impact on the resulting estimates of how much warming is needed and the final accuracy while allowing us to avoid much of the re-warming and significantly reduce the simulation time.

We have analyzed merging both single- and multi-level hierarchies. While the rare merge errors in single-level hierarchies came in the form of incorrect dirty bits, the more complex multi-level hierarchies led to a wide variety of merge errors, including extra blocks, missing blocks, and incoherent configurations. These errors significantly impacted IPC accuracy (from a maximum of almost zero IPC error in single-level cache size setups to up to 0.567 IPC error in the multi-level setups). We investigated these error sources by enumerating the cache states and categorizing and explaining the error sources and their frequency. From that analysis, we were able to identify common errors that we could correct in all cases and those which we could statistically correct: 5 merge errors explicitly and another 12 statistically, bringing down the maximum absolute IPC errors from 0.567 to 0.228 and mean absolute IPC errors from 0.024 to 0.017.

We have demonstrated the value of Cache Merging by using it to improve the performance (1.44×/1.84×/1.87× geomean speedup for the 128kB/2MB/8MB caches) of Adaptive Cache Warming, which is narrow to the expected (ideal) speedup (i.e., 2×). We achieve this speedup with a minimum mean/95-percentile absolute loss of IPC accuracy of only 0.006/0.029, 0.003/0.015, and 0.001/0.006) IPC error. In doing so, we investigated how the merge errors lead directly to simulation errors and indirectly to simulation errors by causing over- or under-warming. Our analysis showed that overall, the simulation experiments are very accurate, but that for the smallest cache size, the proportion of over- or under-warming estimates was up to 9% (with an average −0.01× slowdown overall for that cache size). These results demonstrate that combining Cache Merging with techniques such as Adaptive Cache Warming can significantly reduce simulation time while retaining accuracy.

9 ACKNOWLEDGMENTS

Gustaf wants to thank the UART group for generous input, especially Mihail Popov for helpful comments, Ricardo Alves for useful discussions and Christos Sakalis and Chang Hyun Park for reviewing this paper.

This work was supported by the Knut and Alice Wallenberg Foundation through the Wallenberg Academy Fellows Program (grant No 2015.0153), the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant No. 715283). The computations and data handling were enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC) at NSC (2021/22-435) and UPPMAX (2021/23-626) partially funded by the Swedish Research Council through grant agreement no. 2018-05973.

REFERENCES

[1] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[2] Gustaf Borgström, Andreas Sembrant, and David Black-Schaffer. 2017. Adaptive cache warming for faster simulations. ACM Press. http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-310625

[3] T. M. Conte, M. A. Hirsch, and W.-W. Hwu. 1998. Combining trace sampling with single pass methods for efficient cache simulation. *IEEE Trans. Comput.* 47, 6 (Jun 1998), 714–720. https://doi.org/10.1109/12.689650

[4] Lieven Eeckhout and Koen De Bosschere. 2006. Yet shorter warmup by combining no-state-loss and MRRL for sampled LRU cache simulation. *Journal of Systems and Software* 79, 5 (May 2006), 645–652. https://doi.org/10.1016/j.jss.2005.06.016

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

[6] David Eklov and Erik Hagersten. 2010. StatStack: Efficient modeling of LRU caches. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 55–65.

[7] A. Falcon, P. Faraboschi, and D. Ortega. 2007. Combining Simulation and Virtualization through Dynamic Sampling. In *2007 IEEE International Symposium on Performance Analysis of Systems Software*. 72–83. https://doi.org/10.1109/ISPASS.2007.363738

[8] Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. 2010. Interval simulation: Raising the level of abstraction in architectural simulation. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 1–12.

[9] J. W. Haskins and K. Skadron. 2003. Memory Reference Reuse Latency: Accelerated Warmup for Sampled Microarchitecture Simulation. In *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*.

[10] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[11] Engin Ïpek, Sally A McKee, Rich Caruana, Bronis R de Supinski, and Martin Schulz. 2006. Efficiently exploring architectural design spaces via predictive modeling. *ACM SIGOPS Operating Systems Review* 40, 5 (2006), 195–206.

[12] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). *ACM SIGARCH Computer Architecture News* 38, 3 (2010), 60–71.

[13] KVM. 2016. Main Page — KVM,. https://www.linux-kvm.org/index.php?title=Main_Page&oldid=173792 [Online; accessed 22-June-2022].

[14] Benjamin C Lee, Jamison Collins, Hong Wang, and David Brooks. 2008. CPR: Composable performance regression for scalable multiprocessor models. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE, 270–281.

[15] Yue Luo, Lizy K. John, and Lieven Eeckhout. 2005. SMA: A Self-Monitored Adaptive Cache Warm-Up Scheme for Microprocessor Simulation. *International Journal of Parallel Programming* 33, 5 (Oct 2005), 561–581. https://doi.org/10.1007/s10766-005-7305-9

[16] Scott McFarling. 1993. *Combining branch predictors*. Technical Report. Technical Report TN-36, Digital Western Research Laboratory.

[17] Nikos Nikoleris, Lieven Eeckhout, Erik Hagersten, and Trevor E Carlson. 2019. Directed statistical warming through time traveling. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 1037–1049.

[18] 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)*.

[19] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. ACM, 45–57. https://doi.org/10.1145/605397.605403

[20] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *Proc. Internationl Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[21] Luk Van Ertvelde, Filip Hellebaut, Lieven Eeckhout, and Koen De Bosschere. 2006. Nsl-blrl: Efficient cachewarmup for sampled processor simulation. In *Proceedings of the 39th annual Symposium on Simulation*. IEEE Computer Society, 168–177.

[22] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. 2003. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *Proc. International Symposium on Computer Architecture (ISCA)*.