HALO: A Hierarchical Memory Access Locality Modeling Technique For Memory System Explorations

Reena Panda
The University of Texas at Austin
reena.panda@utexas.edu

Lizy K. John
The University of Texas at Austin
ljohn@ece.utexas.edu

ABSTRACT

Growing complexity of applications pose new challenges to memory system design due to their data intensive nature, complex access patterns, larger footprints, etc. The slow nature of full-system simulators, challenges of simulators to run deep software stacks of many emerging workloads, proprietary nature of software, etc. pose challenges to fast and accurate microarchitectural explorations of future memory hierarchies. One technique to mitigate this problem is to create spatio-temporal models of access streams and use them to explore memory system trade-offs. However, existing memory stream models have weaknesses such as they only model temporal locality behavior or model spatio-temporal locality using global stride transitions, resulting in high storage/metadata overhead.

In this paper, we propose HALO, a **H**ierarchical memory **A**ccess **LO**cality modeling technique that identifies patterns by isolating global memory references into localized streams and further zooming into each local stream capturing multi-granularity spatial locality patterns. HALO also models the interleaving degree between localized stream accesses leveraging coarse-grained reuse locality. We evaluate HALO's effectiveness in replicating original application performance using over 20K different memory system configurations and show that HALO achieves over 98.3%, 95.6%, 99.3% and 96% accuracy in replicating performance of prefetcher-enabled L1 & L2 caches, TLB and DRAM respectively. HALO outperforms the state-of-the-art memory cloning schemes, WEST and STM, while using ~39X less meta-data storage than STM.

CCS CONCEPTS

• **Computer systems organization → Architectures**;

ACM Reference Format:
Reena Panda and Lizy K. John. 2018. HALO: A Hierarchical Memory Access Locality Modeling Technique For Memory System Explorations. In *ICS '18: 2018 International Conference on Supercomputing, June 12–15, 2018, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3205289.3205323

1 INTRODUCTION

The performance gap between processor and memory system continues to a major concern for computer designers and researchers [25, 29, 46, 49]. Growing complexity of emerging applications poses many new challenges for memory system design due to their data intensive nature, complex access patterns and larger footprints. Furthermore, the growth in number of cores puts a tremendous pressure on the memory hierarchy, making memory system performance one of the biggest bottlenecks of overall system performance [42]. Recent advances in die-stacked DRAMs [8, 9, 17], hybrid memory systems and new memory management proposals have expanded the memory system design space, enabling the use of larger caches, deeper hierarchies, etc. As such, finding the optimal memory hierarchy design is very challenging and needs an in-depth understanding of the memory behavior of end-user workloads.

Unfortunately, many real world applications (e.g., web services on node.js, NoSQL databases) are often so large and complex (with lots of software layers) that it is difficult to run and evaluate them on most early performance simulators. Furthermore, getting access to several end-user workloads (e.g., Google's CNNs, trading algorithms, etc.) is rarely possible due to the proprietary nature of client software or traces [27, 32]. Thus, computer designers face the challenge of getting representative information about complex, long-running or proprietary applications that they can analyze to make targeted design decisions.

A promising alternative to address the above challenges is to use workload cloning [10, 11, 21, 22, 24, 27, 33, 45, 48, 54, 60], a process of extracting a statistical summary of the behavior of end-user's workloads [13, 35] though profiling and then, synthesizing a proxy workload that produces the same statistical behavior. *WEST* [11] and *STM* [10] are two proposals that clone the cache and memory behavior of applications. WEST models temporal locality using per cache-set LRU stack distance distribution based on a baseline cache hierarchy. However, WEST does not model spatial locality making it inadequate to evaluate microarchitectural structures that exploit spatial locality (e.g. prefetchers). STM overcomes this limitation and models spatial locality by capturing global stride-based correlations in the memory reference stream. However, global stride transitions of many SPEC CPU2006 benchmarks cannot be captured even by using a stride history depth as long as ~80-100 [10]. Thus, STM has to maintain significantly long histories to capture the dominant stride transitions, which results in significantly higher meta-data storage overhead. Limiting the stride history depth can reduce storage overhead, but at the expense of significantly poor cloning accuracy. Thus, there is a need to design more accurate and efficient solutions to model memory access locality of applications.

In this paper, we propose **HALO**, a **H̲**ierarchical memory **A̲**ccess **L̲O̲**cality modeling technique that can statistically capture the spatial and temporal locality of applications, while incurring less metadata storage overhead. HALO leverages the observation that accurate pattern detection within the global memory reference stream

ICS '18, June 12–15, 2018, Beijing, China
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5783-8/18/06…$15.00
https://doi.org/10.1145/3205289.3205323

is often challenging as global memory reference patterns are affected by several factors, such as data-dependent control-flow, data-structure layout and access interleaving, etc. Rather, memory access patterns can be more accurately and succinctly captured by learning patterns at a localized granularity for most applications. HALO discovers patterns by decomposing an application's memory accesses into a set of independent streams that are constrained to a smaller region of memory and capturing fine-grained patterns within localized regions using repeating stride transitions. This allows the representation of complex workloads through the composition of a set of smaller and simpler building blocks. Additionally, different programs have different locality behavior. HALO exploits this observation to achieve higher meta-data storage efficiency by capturing multi-level stride transitions, which are tailored to an application's locality patterns. However, modeling locality within localized streams alone is not sufficient to recreate the original application's memory behavior. What is required further is a mechanism to combine accesses from these decomposed streams to synthesize an ordered proxy sequence. HALO models this by tracking how accesses to the localized streams are interleaved with respect to each other by using coarse-grained temporal locality tracking.

The combination of statistical profiles captured by HALO can accurately mimic memory locality when we study prefetchers, main memory, vary the cache or TLB configuration and even the page size. Apart from enabling to hide the original memory accesses, HALO can scale down the original benchmarks by generating fewer number of accesses in the proxies leading to reduced simulation time and storage requirement. HALO may also scale up the original benchmarks to model futuristic workloads with larger footprints etc. The key contributions made in this paper are as follows:

- We propose HALO, a hierarchical memory locality modeling technique that exploits fine-grained pattern detection within localized streams & coarse-grained reuse tracking across streams to facilitate evaluation of futuristic memory hierarchies.
- We demonstrate that by exploiting application-locality-specific stride pattern detection within localized streams, HALO achieves better accuracy in modeling original performance than global pattern modeling, while incurring ~39X reduction in meta-data storage requirements compared to state-of-the-art techniques.
- We show that by modeling coarse-granularity temporal locality, HALO mimics the memory footprint and TLB performance of original applications with over 99% accuracy.
- We evaluated HALO using ≥20,000 cache/memory configurations and show that HALO achieves over 98.3%, 95.6%, 99.3% and 96% accuracy in modeling prefetcher-enabled L1 & L2 caches, TLB and DRAM performance respectively, while outperforming WEST and STM techniques.

The rest of this paper is organized as follows: we discuss prior work in section 2. In section 3, we describe HALO's methodology. We discuss the experimental framework and results in sections 4 and 5 respectively before concluding the paper in section 6.

2 BACKGROUND AND RELATED WORK

Distilling the inherent patterns in the memory access streams into a small set of statistics is a very challenging problem. Most prior workload cloning proposals [10, 11, 19, 20, 27, 36, 40, 41, 45, 50, 52, 53, 63] exploit some form of temporal and/or spatial locality to model memory access behavior. Locality models are also useful for synthesizing stressmarks [26, 28, 34], to model program resource demands [16], to utilize multiple granularity architectures [31], to estimate performance of emerging memory architectures [30, 55] and to optimize simulations [23]. In this section, we will discuss the state-of-the-art workload cloning proposals and their challenges.

2.1 Prior Work and their Challenges

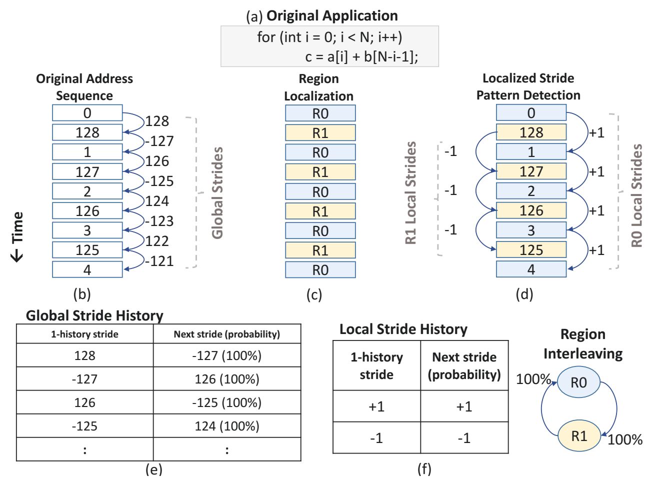
WEST [11] captures temporal locality by tracking per cache-set LRU stack distance distributions, set reuse locality metrics, etc. for every level of a baseline cache hierarchy. It generates a proxy by statistically sampling the collected profiles to create a sequence of accesses to chosen cache sets and ways. However, WEST faces several challenges. First, as WEST's statistics are tightly coupled to the profiled cache configuration, the size overhead of WEST's statistics becomes significantly high for larger caches (e.g., meta-data overhead exceeds 2.5GBs per application for modeling a modern-day 16GB DRAM cache). Second, dependence on the profiled cache configuration causes significant performance cloning inaccuracy when test configurations deviate from the baseline configuration (e.g., changes in cache blocksize) [11]. Finally, as WEST does not model spatial locality, it experiences higher cloning inaccuracies when evaluating prefetchers or the memory system. (see Table 1 for the cloning error of WEST proxies for ≥7000 different prefetcher-enabled last-level cache and TLB configurations across 39 benchmarks; benchmark and configuration details are provided in Sections 4 and 5).

Table 1: Error between WEST proxies and original applications in terms of cache and TLB miss rates.

	LLC miss rates	TLB miss rates
Average Error	19% (avg)	9.3% (avg)
Maximum Error	44% (max)	22% (max)

Spatio-Temporal Memory (STM) [10] leverages application's spatial and temporal locality behavior to create memory proxies. STM captures temporal locality using per cache-set LRU stack distance distribution of a 16KB, 2-way cache. STM further captures spatial locality patterns within the references that miss in the profiled cache by tracking global stride transitions in a stride history table. Past research has shown that a history length of even 100 is also insufficient to capture dominant stride transitions in the global memory access sequence of many SPEC CPU2006 benchmarks (e.g., h264ref) [10]. Maintaining long history-based stride tables significantly increases STM's meta-data storage overhead. We evaluated STM across 39 different benchmarks and we observed that in 12 out of the 39 benchmarks (31% cases), STM's overhead exceeds the original trace size by ~2X. Additionally, across 24 other benchmarks, STM's meta-data size exceeds the original gzip-compressed trace size. Limiting STM's history length can reduce its overhead but it increases aliasing in the global stride tables resulting in poor cloning accuracy. In our experiments, limiting the history length to 40 causes STM proxies to experience up to 24% and 32% error in replicating the TLB miss rate and memory footprint.

Bell et al. [14], Joshi et al. [36] create workload clones by modeling instruction-level behavior. But they use a single dominant stride for every memory instruction to model locality and thus, can not model complex patterns. SLAB [53] models the performance of



1-history stride	Next stride (probability)
128	-127 (100%)
-127	126 (100%)
126	-125 (100%)
-125	124 (100%)
:	:

1-history stride	Next stride (probability)
+1	+1
-1	-1

Figure 1: Global versus local stride transition tracking.

shared last-level caches and main memory by using an approximate stack-distance metric, however, SLAB's statistics do not hold similar correlations in modeling upper-level cache performance. MEMST [12] clones DRAM performance by modeling statistics such as bank conflicts, row buffer hit rate, etc. and is tied to the profiled DRAM parameters. Metoo [62] generates workload clones by replicating instruction-level timing behavior, but the memory addresses are based on WEST's methodology. Maeda et al. [40] model cache performance by modeling temporal locality at 64B and 4KB granularity. However, they use an approximate technique to model spatial locality by using the probability of address bit transitions between consecutive references, which makes them unsuitable for studying performance of prefetchers, main memory etc.

2.2 Overcoming the Challenges

Global stride probability statistics may not be effective in capturing memory access behavior because accesses to different structures are often interleaved and mask the patterns within individual streams. This can be shown using an example (see Figure 1a). This simple program adds two array data-structures ($a[64]$ and $b[64]$), leading to a memory reference and stride pattern sequence shown in Figure 1b (assuming, 1 array entry = 1 byte = 1 cache-block). We can observe that the global stride patterns are non-repetitive. Still, capturing this global stride sequence is feasible even with a 1-length global stride history table (see Figure 1e), but it would require saving every individual stride transition, which is almost equivalent to saving the entire memory trace. However, we can also observe that accesses to the individual data-structures have significant regularity (+1 and -1 strides respectively, see Figure 1d), which is not otherwise discernible by looking at the global memory sequence alone.

Although simple, this example shows how many simple access patterns cannot be captured using global stride patterns effectively. More number of data-structures with greater degree of interleaving is likely to cause greater aliasing in the stride tables (with limited global history), leading to poor cloning accuracy. In this paper, we propose "***HALO***", a statistical workload cloning methodology that captures the cache and memory behavior of applications and models them to create miniature proxy benchmarks. Our goal is to accurately mimic the spatial locality, temporal locality and memory footprint of an application, without incurring high meta-data profile storage overhead. HALO leverages the observation that different data-structures have different locality properties and their access patterns can be detected more easily by analyzing localized access patterns. Thus, HALO discovers patterns by first decomposing memory references into localized address regions and then identifying access patterns within individual regions using repeating stride transitions. In this example, HALO localizes addresses into two regions (*R0 & R1*) and learns stride transitions within the localized regions as shown in Figures 1c & 1d respectively (a memory region = 64 cache-blocks). However, capturing intra-region locality patterns alone is not sufficient to recreate the original memory access behavior in the proxy benchmark. What is equally important is to capture how accesses to these individual regions are interleaved with respect to each other. HALO models the interleaving information by exploiting coarse-grained temporal locality patterns and uses it to synthesize an ordered proxy reference sequence from individual localized stream accesses (see Figure 1f).

3 METHODOLOGY

Figure 2 shows an overview of HALO's memory locality modeling framework. During the profiling phase ①, HALO characterizes the application's inherent memory access patterns to create a statistical workload-specific profile ②. HALO discovers memory access patterns by decomposing the original references into different regions (***"region localization"*** Ⓐ) and capturing fine-grained access patterns within individual regions using repeating stride transitions (***"intra-region stride locality"*** Ⓒ). In particular, HALO captures ***multi-level stride transition probability distributions***, which are tailored to the locality behavior of different applications, to achieve higher cloning accuracy and meta-data storage efficiency. HALO further captures how accesses to these individual localized regions are interleaved with respect to each other by tracking coarse-grained temporal locality patterns (***"inter-region reuse locality"*** Ⓑ). During the proxy synthesis phase ③, HALO adopts a systematic methodology to create a miniature memory access clone of the original application based on the captured workload-specific profile, which can then be used to drive cache hierarchy, TLB and memory system performance exploration. To do so, HALO first generates proxy accesses within localized memory regions by leveraging the collected intra-region stride statistics (***"intra-region access generation"*** Ⓓ) and then, interleaves accesses from the localized streams using the captured reuse locality statistics (***"inter-region interleaving reconstruction"*** Ⓔ) to create an ordered proxy reference sequence. Next, we will discuss HALO's workload characterization methodology followed by its proxy generation algorithm.

3.1 Region Localization

During the *region localization* step, HALO divides the address space into fixed-size segments called ***regions*** and assigns the original memory references to different regions, based on the higher-order address bits. The key idea behind region localization is that for most applications, similar data-structures (with similar access patterns) are often laid out in continuous address segments. Accesses

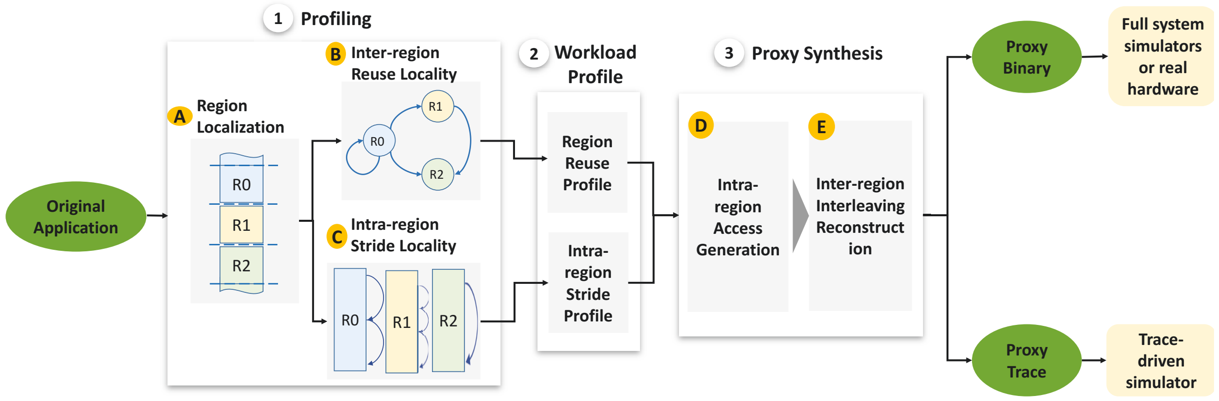

1 Profiling
A Region Localization
R0
R1
R2
B Inter-region Reuse Locality
C Intra-region Stride Locality
Original Application
2 Workload Profile
Region Reuse Profile
Intra-region Stride Profile
3 Proxy Synthesis
D Intra-region Access Generation
E Inter-region Interleaving Reconstruction
Proxy Binary
Proxy Trace
Full system simulators or real hardware
Trace-driven simulator

Figure 2: HALO's workload cloning methodology.

to such regions or data-structures often have different patterns as compared to other regions or data-structures that are accessed together. Detecting patterns within a single global access stream is usually not effective or has higher storage overhead because effects such as data-dependent control flow, program complexity, data-structure access pattern differences, data layout, etc. lead to increased entropy in the global reference patterns. In contrast, using localized pattern detection can lead to more accurate representation of access patterns. Localized pattern correlation is also leveraged by many prefetchers [38, 44, 47, 59, 61] for making prefetch predictions. HALO considers each memory region to be a contiguous 4KB segment in the memory space. Adjacent regions with similar intra-region stride patterns can be merged to form larger regions to account for varying program locality, as we will discuss in the next section.

3.2 Intra-region Stride Locality Tracking

After localizing the original memory accesses into different regions, HALO captures fine-grained access patterns within individual regions using intra-region stride probability distributions. However, what stride history length can efficiently capture dominant intra-region stride locality behavior across different applications?

Figure 3 shows the cumulative fraction of intra-region stride transitions (y-axis) that can be captured using increasing history-length based stride transition tables (x-axis) without having any

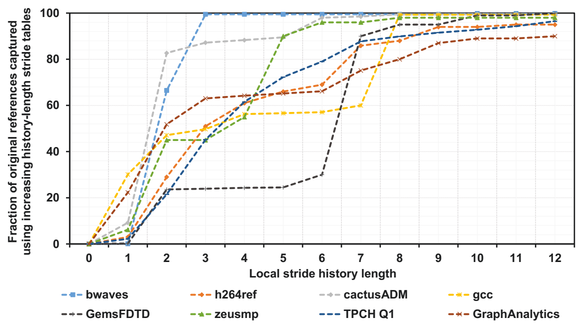
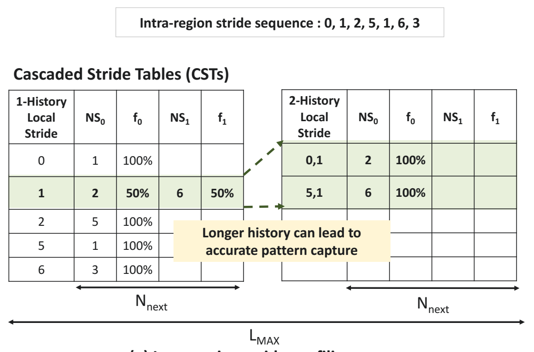


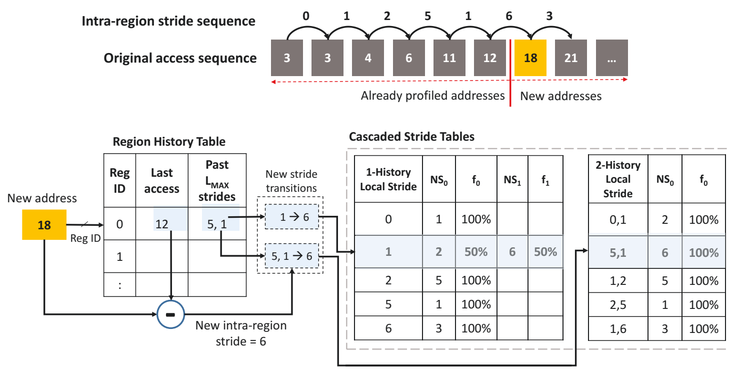
Figure 3: Fraction of original reference patterns captured using increasing history length stride tables.

aliasing effects for 8 applications. We can see that applications have diverse locality behavior. For example, for bwaves benchmark with highly strided access patterns, more than 98% of the intra-region stride transitions can be summarized using a history length of 3. Similarly, both cactusADM and zeusmp benchmarks operate on a 3D array/grid and have fairly strided access patterns. However, while cactusADM iterates over the grid points in one dimension, zeusmp iterates over data points in all three dimensions. Thus, most dominant intra-region access patterns of cactusADM can be summarized using a history length of 2, but zeusmp requires slightly longer stride history length ($\sim 4 - 6$). On the other hand, for benchmarks such as graph analytics, which consists of many complex indirect references, using a local history length of 10 also suffers from aliasing effects in a few memory regions. In any case, it should be noted that the localized patterns can be captured using much shorter history lengths as compared to global memory patterns. For example, in h264ref benchmark (see Figure 3), most intra-region stride transitions can be captured using a local history length of 8, while even ~100 history length is not enough to capture the dominant global stride transitions [10].

To leverage the diverse program locality to improve cloning accuracy and meta-data storage efficiency, HALO proposes to tailor the stride history length based on the application's locality needs. HALO achieves this by using a set of cascaded stride tables (CSTs) to capture the intra-region stride transitions. Each stride table tracks a longer stride history length and associates specific intra-region stride histories with the next possible strides to the same region. Figure 4a shows an example to demonstrate the working of the CSTs. We will first clarify several notations: CST_i refers to a stride table tracking i-length stride history, L_{MAX} refers to the maximum cascading degree ($L_{MAX} = 2$ in this example), (NS_0, f_0) refers to the first stride value and probability for the specific stride history pattern, etc. In the original stride sequence shown in Figure 4a, stride $\{1\}$ is followed by strides $\{2\}$ or $\{6\}$ with equal probability, which causes aliasing in the CST_1 table. Using only 1-history transitions for proxy synthesis can lead to a different stride interleaving in the proxy versus the original application because of such pattern aliasing. The aliasing effects can however, be eliminated in this example by capturing 2-history stride transitions in the CST_2 table.


Intra-region stride sequence : 0, 1, 2, 5, 1, 6, 3
Cascaded Stride Tables (CSTs)
1-History Local Stride
NS0
f0
NS1
f1
2-History Local Stride
Longer history can lead to accurate pattern capture
Nnext
Nnext
LMAX

(a) Intra-region stride profiling structures



(b) How CSTs are used during profiling and proxy generation

Figure 4: Intra-region locality profiling using cascaded stride tables (CSTs).

Using the CST_2 table can accurately model the stride following $\{1\}$ with 100% accuracy depending on its preceding stride ($\{0\}$ or $\{5\}$). Capturing other 2-history stride transitions in the CST_2 table (e.g., $\{2, 5\} \longrightarrow \{1\}$) is not necessary as the same patterns can be captured using 1-history transitions ($\{5\} \longrightarrow \{1\}$). Thus, using CSTs enables locality-specific access pattern capture, where, shorter history tables can efficiently capture simple/regular patterns, while more complex patterns are tracked using longer history-based stride transitions. Conceptually, using multiple cascaded tables to track histories of varying lengths is similar to the state-of-the-art TAGE branch predictor [56] or variable length delta prefetcher [58].

Figure 4b shows the profiling structures used for capturing multi-level stride transitions. During a profiling interval, HALO keeps track of accesses to different regions using the *region history table* (RHT). Each RHT entry tracks the number of region accesses, past L_{MAX} intra-region strides within the region, etc. To profile a memory access, RHT is indexed using the address's region *index* and a new stride is computed based on the region's last seen address. The CST tables are updated based on the new stride and the history of last L_{MAX} strides to the region. Unfortunately, during the profiling interval, it is not known as to which history length can capture the current stride transition with least aliasing (as the entire application reference stream has not been profiled yet). Thus, during a profiling interval, HALO updates all the stride tables using the accumulated stride history of the corresponding region, where each cascaded table tracking history length L_i is updated using the accumulated last L_i intra-region strides, $\forall i \leq L_{MAX}$. For example, when address *18* (region *R0*) is profiled, both CST tables are updated using the accumulated stride history and the new stride ($\{1\} \longrightarrow \{6\}$ *and* $\{5, 1\} \longrightarrow \{6\}$ respectively).

At the end of the profiling interval, HALO analyzes the complexity of stride transitions captured in the CSTs (starting from the longest-history one) and identifies the minimum history length (L'_{MAX}) that can capture the respective access patterns with least aliasing. In this example, at the end of the profiling interval, HALO post-processes the CST_2 table and invalidates the last three entries ($\{1, 2\} \longrightarrow \{5\}, \ldots, \{1, 6\} \longrightarrow \{3\}$) as the same patterns are captured in the CST_1 table. Also, the $\{1\} \longrightarrow \{2, 6\}$ entry in the CST_1 table is invalidated as 2-history pattern needs to be captured to remove aliasing effects. Figure 4a shows the final state of the CST tables after post-processing. ***The final post-processed state of the CST probability distribution tables is saved for proxy generation***. During post-processing, adjacent regions with similar intra-region stride patterns can be identified and merged to form larger regions. Since individual CST tables contain a maximum of a few tens to hundreds of entries for most applications, the time overhead to manage the cascaded tables is not significant.

3.3 Inter-region Reuse Locality

Capturing intra-region locality metrics alone is not sufficient to recreate the original memory access locality. HALO also captures how accesses to the individual regions are interleaved with respect to each other. To understand why, let us re-visit the program example in Figure 1. This program makes repeated accesses to the two arrays in an interleaved manner. However, during proxy generation, if the cloning framework generates accesses to the two arrays in a sequential manner (all accesses to $R0$ finish before $R1$ is accessed), the proxy program's locality will be very different from the original program. HALO captures the degree of interleaving between accesses to individual memory regions by monitoring coarse-grained temporal locality patterns using the ***region reuse distribution*** **(Π)** metric. The Π distribution captures the number of unique region accesses between successive accesses to the same region. Figure 5 shows an example of Π metric computation for the program discussed in Figure 1. The last row shows the computed Π metric (∞ represents a newly-accessed region). During proxy generation phase, the Π profile is used to reconstruct an ordered memory reference sequence from individual region access streams.

Modeling the Π metrics further help to accurately control the memory footprint of the generated proxy (based on the ∞ counts in the Π-profile). We observed that synthesizing proxies using only global stride transitions suffers from up to 195%, 91% and 55% error in replicating the memory footprint of benchmarks using a stride history length of 10, 30 and 60 respectively due to aliasing in the stride tables. Higher error in replicating the application memory footprint translates into higher TLB, cache and DRAM performance errors. On the other hand, by tracking stride transitions at a localized granularity, HALO reduces the error rates by reducing aliasing effects. Modeling inter-region reuse locality enables HALO proxies

Access	A[0]	B[N]	A[1]	B[N-1]	A[2]	B[N-2]	A[3]	B[N-3]	A[4]
Address	0	128	1	127	2	126	3	125	4
Region Address	0	1	0	1	0	1	0	1	0
∏ Metric	∞	∞	1	1	1	1	1	1	1

Figure 5: Inter-region reuse locality tracking.

Table 2: Profiled statistics

Statistic	Description
$CST = \{CST_1, \ldots, CST_{L_{MAX}}\}$	Set of cascaded stride tables with increasing history length
L_{MAX}	Maximum cascading degree
CST_i	Stride pattern table keeping stride transition counts from past i intra-region strides to next stride
N_{next}	No. of next intra-reg strides
$\Pi_{RD\|Count}$	Region reuse distance histogram
RD_{MAX}	Maximum region reuse distance bin
ρ_{rw}	Fraction of write accesses
$Rate_{mem}$	Rate of memory access generation

to achieve over 99% accuracy in replicating the desired memory footprint and TLB miss rate of the original applications.

3.4 Proxy Generation

Table 2 summarizes the key statistics that HALO captures to model application memory behavior. These statistical profiles are used to generate HALO's memory proxies. Algorithm 1 shows HALO's proxy generation algorithm. Table 2 statistics are provided as an input to the algorithm. The output is the memory proxy characterized by a tuple $\{(ADDR_i, RW_i)\}$, where $ADDR_i$ refers to the i^{th} proxy address and RW_i denotes the access type. Before proxy generation, miniaturization is applied by scaling down the collected statistical input profiles by the desired scaling factor, T_{min}. Care should be exercised when choosing an appropriate scaling factor because scaling beyond a certain limit will cause inaccuracies in modeling the memory reference patterns in the proxy due to the law of large numbers. We will evaluate sensitivity of cloning accuracy to the scaling factor in Section 5. In this algorithm, we assume the existence of a data-structure (*RegInfo*) to track the LRU history of distinct region references. The last n^{th} accessed region can be obtained by using the function *Get_Region(n)*, while the *RegInfo* data-structure can be updated as new regions get accessed using the *Update_Region()* function.

To generate the i^{th} memory address, the Π profile is sampled to obtain a region reuse distance value (line 4). The corresponding region is obtained by invoking the *Get_Region()* function. If chosen reuse distance is greater than RD_{MAX} (corresponding to ∞ reuse distance), a new region is chosen by sampling the address space. Then, the *RHT* table is looked up to find the last accessed address and stride history of the chosen region. The CSTs are looked up one-by-one (lines 12-19), starting from the longest history table, by using a partial hash of the accumulated stride history. A new stride is chosen based on the longest history match in the CSTs. This ensures that the next stride assignment is done using the most accurate profiled information. Finally, the i^{th} address is computed (line 21) using the last accessed address (*LAST_ADDR*) of the region and the chosen stride (S_i). Finally, the *RHT* entry and *RegInfo* data-structure are updated (lines 22-24). This process is repeated until the target number of references N is generated. In case the output is desired to be binary executable, HALO generates a C code with inline x86-64 assembly instructions for moving data from desired memory locations to specific registers for reading or writing. We ensure that all memory references access allowed memory locations within a large allocated memory region.

Algorithm 1 HALO's Proxy Generation Algorithm

```
1:  Input: Table 2 Statistical Profiles.
2:  Output: Trace $T[] = \{(ADDR_1, RW_1), \ldots, (ADDR_N, RW_N)\}$;
3:  for $i = 1, \ldots, N$ do
4:      Sample $\pi_i$ from $\Pi$
5:      if $\pi_i < RD_{MAX}$ then
6:          $Reg_i$ = RegInfo.Get_Region($\pi_i$);
7:      else
8:          Sample $Reg_i$ uniformly in the address space;
9:      end if
10:     $(R, LAST\_ADDR, LAST\_STR)$ = RHT[$Reg_i$];
11:     $j = L_{MAX}$;
12:     while $j > 0$ do
13:         $f = CST_j$.find($LAST\_STR[L_{MAX} - j : L_{MAX}]$)
14:         if $f == True$ then
15:             Sample stride $S_i$ from $CST_j$;
16:             break;
17:         end if
18:         j−;
19:     end while
20:     Sample $RW_i$ from $\rho_{rw}$;
21:     $ADDR_i = LAST\_ADDR + S_i$;
22:     $LAST\_STR$.push_back($S_i$); $LAST\_ADDR = ADDR_i$;
23:     RHT[$Reg_i$].UpdateState($LAST\_ADDR$, $LAST\_STR$);
24:     RegInfo.Update_Region($Reg_i$);
25: end for
26: return Trace[]
```

3.5 Execution Phase Consideration

We consider different execution phases of a program during proxy generation. To account for phase behavior, HALO divides the original access sequence into fixed size intervals and tracks an independent intra-region stride and inter-region reuse profile per profiled interval period. The *RegInfo* data-structure (for tracking region reuse) is not cleared between phases. HALO uses the per-interval stride and reuse profiles to generate a proxy sequence for the corresponding interval. In our experiments, we choose the interval length to be 100,000 memory references. However, we later show the sensitivity of cloning accuracy to changes in phase lengths.

3.6 Multi-programmed Workload Performance

When applications are co-scheduled on a CMP, memory access streams from different applications compete for the shared cache space. To model the cache-sharing behavior of co-scheduled workloads, HALO uses another statistic - the rate at which memory references are generated per application ($Rate_{mem}$). This metric accounts for the fraction of memory instructions over total instructions, instruction level parallelism and relative speed of the processor cores. As HALO proxies do not produce instruction streams other than memory references, HALO controls the distance between successive memory references based on this rate metric.

4 EXPERIMENTAL FRAMEWORK

We evaluate HALO using 39 benchmarks from different application classes (see Table 3): (a) 26 SPEC CPU2006 benchmarks [5] using "ref" input set (all benchmarks except perl, sjeng and dealII due to compilation issues), (b) 6 benchmarks from the newly introduced SPEC CPU2017 [6, 51] suite (leela, exchange2, imagick, pop2, roms and nab of SPECspeed category), (c) 3 TPC-H [7] queries using

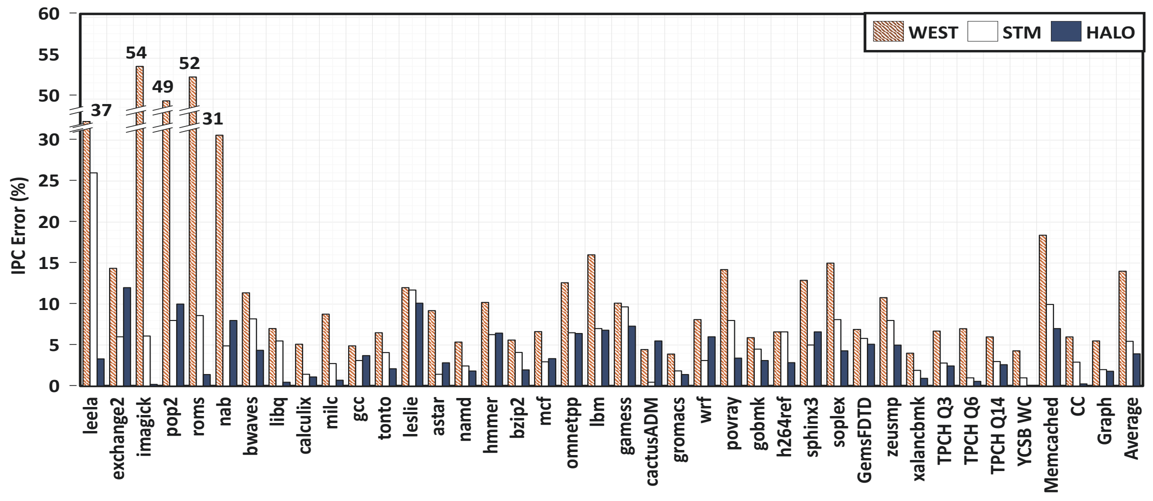


Figure 6: Instructions per cycle error of WEST, STM and HALO proxies versus the original applications.

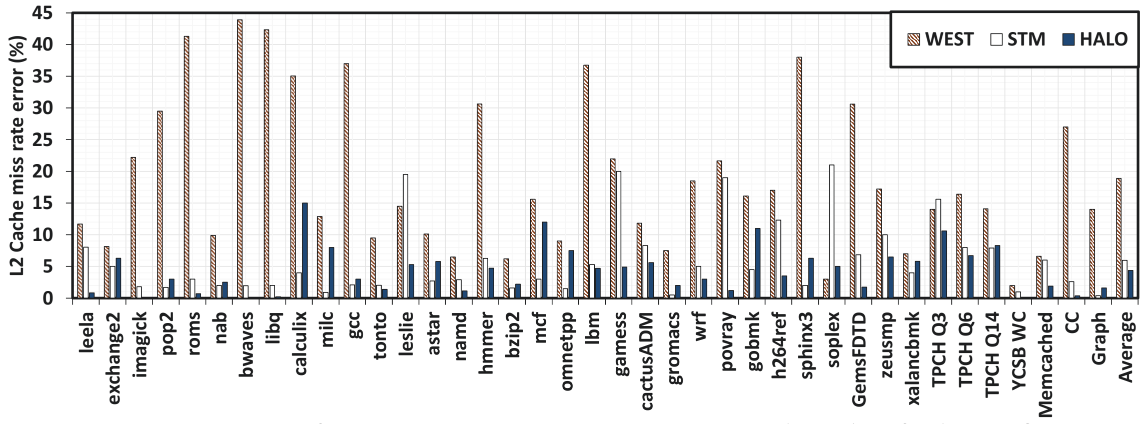


Figure 7: L2 miss rate errors of WEST, STM & HALO proxies across L2 cache and prefetcher configurations.

MySQL [4] database (10GB), (d) data-serving workload based on Yahoo! Cloud Serving Benchmark (YCSB) [18] framework, (e) graph analytics *tunkrank* application using Graphlab [2] framework from Cloudsuite and *connected components* application using GraphChi [1] framework, and (f) data-caching benchmark based on Memcached [3] from Cloudsuite.

Table 3: Benchmarks

Category	Benchmarks
General-purpose	26 SPEC CPU 2006 benchmarks 6 SPEC CPU 2017 benchmarks
Data Analytics	TPCH Q3, Q6, Q14
Graph Analytics	Connected Components (CC), TunkRank (Graph)
Data Caching	Memcahed
Data Serving	YCSB Workload C (WC)

Table 4: Profiled system configuration

Component	Configuration
CPU	X86_64 processor, atomic mode 4 GHz Single-core and multi-programmed runs
L1 cache	32KB, 2-way Icache; 64KB, 2-way Dcache 64B block size, LRU
L2 cache	4MB, 8-way, LRU, Shared
Main memory	4GB DDR3, 12.8 GB/sec
OS	Ubuntu 14.04

We profile the benchmarks using a Pin-based [39] detailed simulator. The system configuration used for profiling is shown in Table 4. For CPU2006 benchmarks, we profile a Simpoint [43, 57] of 250 million instructions. For the other benchmarks, we fast forward by 10 billion instructions, and then profile the execution of next 250 million instructions. We choose 250 million instructions to make the simulation runs for validation manageable. It should be noted that the HALO uses a statistical profile as input for proxy generation, which is independent of the execution length. For evaluation of different cache and prefetcher configurations, we use a validated trace-driven cache simulator. Our simulator is validated by comparing its miss rates with the standard cache modules provided with gem5 [15] simulator. For evaluation and testing of DRAM memory performance, we use the Ramulator [37] simulator. We compare our results against the state-of-the-art WEST and STM proposals.

5 RESULTS AND ANALYSIS

In this section, we validate HALO's effectiveness in replicating cache, prefetcher, TLB and DRAM performance of applications over 20,000 different configurations. We use two metrics for validation: error between original and proxy performance metrics and Pearson's correlation coefficient. Pearson's correlation coefficient indicates how well the proxy benchmarks track the trends

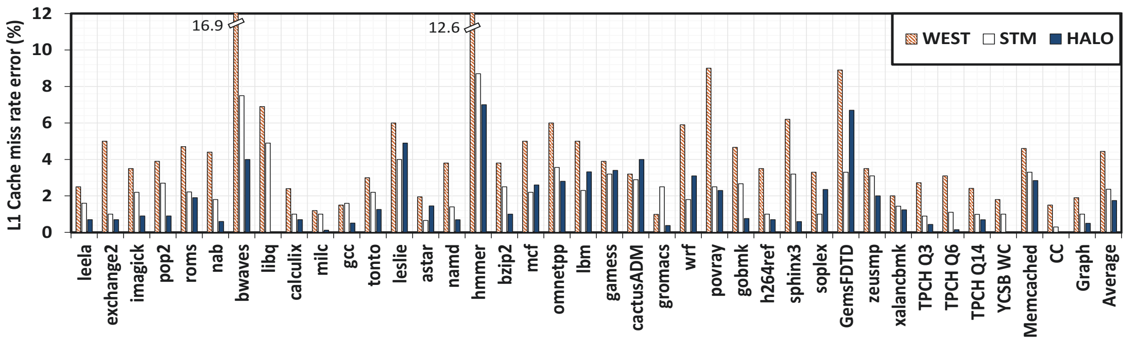


Figure 8: L1 miss rate errors of WEST, STM and HALO proxies across L1 cache and prefetcher configurations.

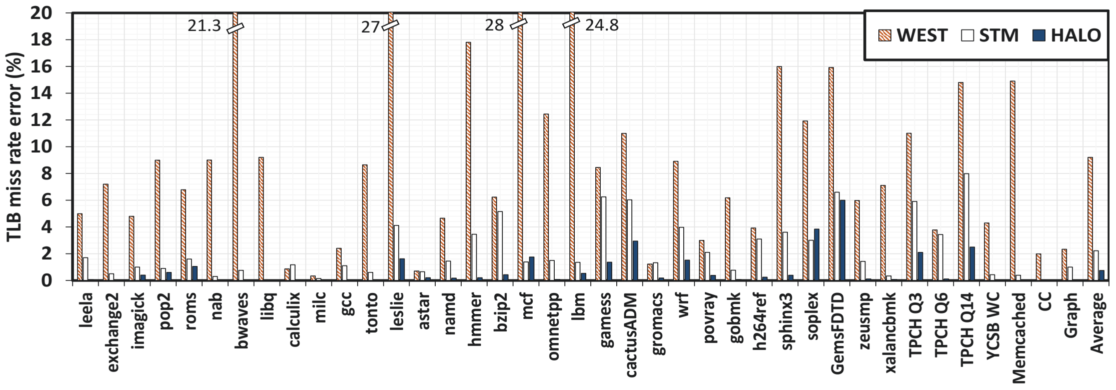


Figure 9: TLB miss rate errors of WEST, STM and HALO proxies across different TLB & page size configurations.

in the original applications, with 1 indicating perfect correlation, and 0 indicating no correlation. During design-space exploration, computer architects consider relative performance ranking (e.g. evaluate which configuration has a lower miss rate).

Instructions per cycle (IPC) - First, we evaluate HALO by measuring the performance of the original and proxy workloads across different configurations as we vary the size, associativity and line-size of the L1 and L2 caches. We also vary L2 stream prefetcher configurations, evaluating a total of over $\sim 6,600$ configurations. Figure 6 shows the error between IPC of the original and proxy workloads. Overall, the average error in replicating original workload IPC for WEST, STM and HALO proxies is 14%, 5.4% and 3.9% respectively. Higher cloning accuracy of HALO proxies over WEST and STM proxies is a result of more accurate modeling of cache, prefetcher and memory system performance. We will elaborate on performance implications of the individual metrics in the following paragraphs. Please note that IPC is used here as a metric to validate the proposed memory model across a range of memory hierarchy configurations, but is not indicative of processor-side performance (as HALO does not model non-memory instructions).

L2 cache and prefetcher configurations - Here, we will show HALO's effectiveness in replicating L2 cache performance by varying the L2 cache and prefetcher configurations. We evaluate 35 L2 cache configurations per benchmark (varying cache size between 1MB-16MB, associativity between 2-32 and line size between 32-128). For each cache configuration, we also vary L2 stream prefetcher configurations by changing number of stream buffers between 8-64 and prefetch degree between no-prefetching/1/2/4/8, leading to a total of 260 configurations per benchmark. Figure 7 shows the L2 miss rate error between the original and proxy benchmarks (averaged across different configurations). The average error in replicating L2 cache miss rates for WEST, STM and HALO proxies are 18.9%, 6% and 4.4% respectively. The correlation coefficients are 77.9%, 97.5% and 98.5% for WEST, STM and HALO respectively.

As WEST does not model spatial locality, it suffers from high errors especially when prefetchers are enabled for prefetch-friendly benchmarks (e.g., bwaves, libquantum). Also, WEST captures stack distance distributions at a cacheline granularity, and thus, suffers from high cloning errors when cache line-size changes, cache size increases. By modeling spatial locality patterns, STM outperforms WEST. However, for many benchmarks e.g., leela, h264ref, exchange2, povray, STM experiences high aliasing in its global stride tables, reducing cloning accuracy. Also, STM captures global stride transitions at a cacheline granularity. Thus, STM proxies do not capture spatial locality within cachelines and perform poorly when cacheline size is varied in some cases (e.g., zeusmp).

HALO outperforms both WEST and STM. HALO performs well even for benchmarks like leela, h264ref, povray, by using a local history depth of 8; dominant access patterns of these benchmarks cannot be captured even using 80-length global stride history transitions by STM. By leveraging multi-granularity stride transitions, HALO not only performs well for benchmarks like libquantum, which have regular strided patterns, but also for benchmarks like gcc and sphinx3, which make a lot of irregular data-structure accesses or bzip2, which has a significant fraction of control-flow dependent loads. HALO experiences high cloning errors with calculix and gobmk benchmarks (14.5% and 11% respectively), but the high L2 miss rate errors occurs systematically for configurations

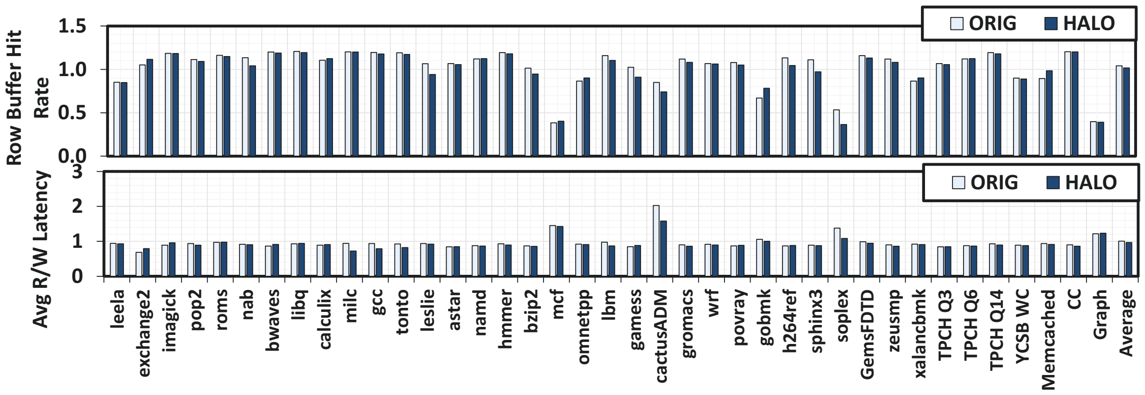


Figure 10: Comparing DRAM performance of HALO and original applications across different DRAM configurations.

with very few L1 cache misses; average L2 MPKI error is ≤0.01, which causes insignificant impact on IPC (≤ 1% and 3% for calculix and gobmk respectively) as shown in Figure 6. Benchmarks like TPC-H Q3 (complex *join* operation across three database tables) and mcf (operation on array of pointers) experience ∼10-11% error with a stride history of 8 due to aliasing; increasing local history depth to 14 reduces errors to ∼5.8% but increases profile sizes.

L1 cache and prefetcher configurations - We evaluate HALO's effectiveness across 40 different L1 cache configurations per benchmark (varying the cache size from 4KB-128KB, associativity from 2-16 and cache line-size between 32B-128B). For each cache configuration, we vary the L1 stream prefetcher configurations by changing the stream detection window between 8-32, prefetch degree between no-prefetching-8, resulting in 264 configurations per benchmark. Figure 8 shows the L1 miss rate errors. The average L1 cache miss rate error between original and proxy workloads is 4.5%, 2.4% and 1.8% for WEST, STM and HALO respectively.

STM captures temporal locality using per-set LRU stack distance distributions for a 16 KB, 2-way L1 cache, however it does not track any statistics related to access distribution or ordering across cache sets. Thus, for benchmarks such as bwaves, libquantum, STM proxies produce different conflict behavior across cache sets when the L1 test and profiled configurations differ significantly, resulting in higher cloning errors. Owing to LRU-stack based modeling of temporal locality behavior, WEST experiences higher errors when test configurations (especially, cacheline size) deviate from the baseline configuration. Overall, HALO outperforms both WEST and STM by exploiting higher predictability in localized memory access streams even with shorter history lengths. HALO experiences higher L1 performance modeling error for hmmer and GemsFDTD benchmarks (∼7%) as HALO does not model inter-region spatial locality which leads to cloning inaccuracies especially with prefetching.

TLB and page size configurations - Next, we will evaluate TLB performance of WEST, STM and HALO proxies (see Figure 9). We vary the number of TLB entries between 8-128 and page size between 1KB-16KB (total 25 configurations per benchmark). Overall, WEST, STM and HALO have 9.2%, 2.4% and 0.7% error in replicating TLB miss rates of original applications. WEST generates a random memory address for any references that miss in the L2 cache. This causes higher deviation in the memory footprint and fraction of active pages between WEST proxies and original applications. STM is more accurate in replicating TLB behavior than WEST, however, aliasing in STM's global stride tables also causes errors in replicating memory footprint and TLB performance. In contrast, by leveraging coarse-grained reuse locality to model inter-region interleaving, HALO can accurately model TLB performance across most benchmarks except GemsFDTD. HALO proxies are generated using a base region size of 4KB. In GemsFDTD, increasing the page size affects the inter-region access interleaving order, which results in higher TLB errors. For most other benchmarks, changing the TLB or page size configuration has minimal impact on HALO's accuracy. Overall, HALO outperforms both WEST and STM, achieving an average accuracy of 99.3%.

DRAM Performance - Next, we will evaluate HALO proxies to be used for memory subsystem design exploration in lieu of the original workloads. We evaluate over 25 DRAM configurations per benchmark by changing the DRAM bus width (4-16 bytes), bus frequency (800MHz-1GHz) and DRAM address mapping schemes (RoBaRaCoCh/ChRaBaRoCo) by swizzling the address decoding bits etc, while simultaneously varying the L2 cache size and associativity. We compare the original and proxy workloads across two key memory system performance metrics: DRAM row buffer hit rate and average read/write latency (see Figure 10). By accurately capturing the spatial and temporal locality of applications, HALO proxies perform closely with respect to the original application, achieving an average error rate of 2.3% and 4% for DRAM row buffer hit rate and average read/write latency respectively.

Phase-level cache performance modeling - Figure 11 compares the phase-wise L2 cache miss rates of the original and HALO workloads for the Graph-analytics and GemsFDTD benchmarks. Every corresponding phase of the original and proxy workload is aligned after accounting for miniaturization. We can observe that the cache miss rate of Graph analytics workload varies between ∼70-100% across the different phases and the HALO proxy follows

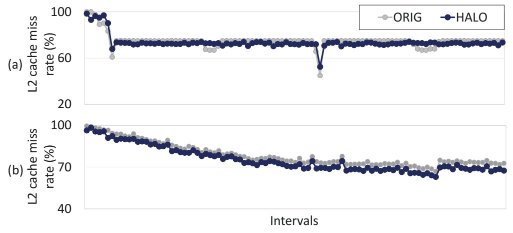


Figure 11: Example showing phase-level cache perf. modeling for (a) GemsFDTD and (b) Graph analytics

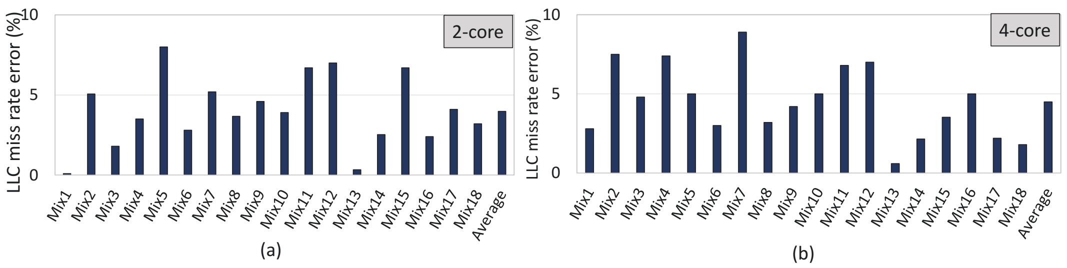


Figure 12: Multi-programmed performance error of HALO proxies for (a) 2-core and (b) 4-core workload mixes

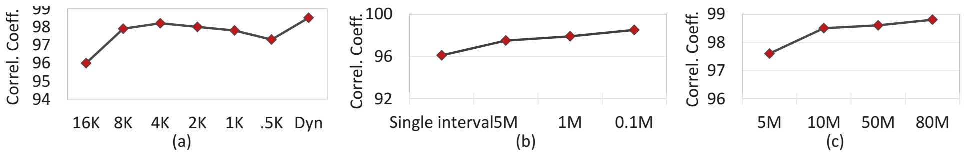


Figure 13: Impact of changing the (a) region size, (b) profiling interval period, (c) trace length on profiling accuracy.

the original application's trends very closely (average error = 1.6%). Similarly, although GemsFDTD experiences slightly higher average error, the proxy still captures the relative trends across different execution phases quite accurately. HALO has similar phase-level cloning accuracy across other benchmarks as well.

Multi-programmed workloads - Next, we will evaluate how accurately can HALO proxies, generated for benchmarks running in stand-alone mode, replicate shared cache behavior when co-scheduled with other applications. We categorize applications according to their L2 miss rates and randomly choose 18 benchmark mixes to co-schedule in a 2-core and 4-core setup. We evaluate 40 shared L2 cache configurations per mix by varying its cache-size, associativity, line-size and replacement policy. Overall, the average error in shared L2 cache miss rate between HALO and original multi-programmed workloads is 4% and 4.9% for 2-core and 4-core configurations respectively (see Figure 12).

Meta-data overhead - By exploiting higher predictability in localized memory access streams combined with an application-locality-specific multi-level stride capture mechanism, HALO achieves, on average, a ∼39X reduction in meta-data storage size as compared to STM, while also outperforming STM across the evaluated performance metrics. HALO's meta-data is also up to 29X smaller than gzip-compressed trace sizes. WEST does not capture spatial locality, as a result of which, it suffers from significantly high cloning errors. Also, WEST's statistics are directly proportional to the profiled cache configuration, as a result the size overhead of WEST's statistics becomes significantly high for larger caches (e.g., meta-data overhead exceeds 2.5GBs per application for modeling a modern-day 16GB DRAM cache).

Sensitivity study - We explore HALO's sensitivity to several parameters by measuring correlation between proxy and original workloads across different L1 cache and prefetcher configurations. First, we evaluate sensitivity to the region size by varying it from 0.5KB - 16KB and dynamic (see Figure 13a)). As region size increases, correlation drops slightly because of higher entropy in larger region patterns, which is difficult to capture using the same history depth without increasing aliasing. Smaller region sizes lead to accurate intra-region pattern capture, but reducing the region size below 0.5KB resulted in reduced performance correlation especially with prefetching because of not modeling inter-region spatial locality. Figure 13b shows HALO's performance sensitivity to the profiling interval size. As the interval size reduces, correlation improves because of accurate capture of phase-level performance patterns. However, having a very small profiling interval increases the profile size correspondingly. In our experiments, a profiling interval of 100,000 memory references provided the best balance of accuracy and meta-data overhead. Next, we evaluate the impact of trace miniaturization factor on cloning accuracy (see Figure 13c). As HALO relies on statistical convergence to generate the proxies, the scaling factor depends on the original number of accesses because of the law of large numbers. We can observe that the performance correlation holds good with 10 million memory references.

6 CONCLUSION

In this paper, we propose a novel memory locality modeling framework, HALO, that accurately models the spatial and temporal locality of applications. HALO isolates global memory references into individual localized regions and captures intra-region access patterns using fine-grained spatial locality patterns. To achieve greater meta-data storage efficiency, HALO captures multi-level stride patterns tailored to application's locality behavior. HALO synthesizes memory access streams from individual, localized stream accesses by modeling the degree of interleaving between accesses to different regions using coarse-grained temporal locality metrics. We evaluated HALO using ≥20,000 different cache, prefetcher, TLB, page-size and DRAM configurations and showed that HALO achieves over 98.3%, 95.6%, 99.3% and 96% accuracy in replicating prefetcher-enabled L1 & L2 caches, TLB and DRAM performance respectively. HALO outperforms the state-of-the-art workload cloning proposal in terms of cloning accuracy across all the evaluated metrics, while using ∼39X less meta-data storage.

7 ACKNOWLEDGEMENT

This research was supported in part by National Science Foundations under grants 1745813 and 1725743, and by the Texas Advanced Computing Center (TACC). Reena Panda was supported in part by a Graduate School Fellowship. Any opinions, findings, conclusions or recommendations are those of the authors and not of the National Science Foundation or other sponsors.

REFERENCES

[1] GraphChi. https://github.com/GraphChi/graphchi-cpp.
[2] GraphLab. www.graphlab.org.
[3] Memcached. www.memcached.org.
[4] MySQL. http://www.mysql.com.
[5] SPEC CPU 2006 Benchmarks. www.spec.org/cpu2006.
[6] SPEC CPU 2017 Benchmarks. www.spec.org/cpu2017.
[7] TPC-H Benchmark Suite. http://www.tpc.org/tpch.
[8] Jedec standard jesd235. High Bandwidth Memory (HBM) DRAM, 2013.
[9] Jedec standard jesd235a. High Bandwidth Memory (HBM) 2 DRAM, 2016.
[10] A. Awad and Y. Solihin. Stm: Cloning the spatial and temporal memory access behavior. *HPCA*, pages 237–247, 2014.
[11] G. Balakrishnan and Y. Solihin. West: Cloning data cache behavior using stochastic traces. *HPCA*, pages 387–398, 2012.
[12] G. Balakrishnan and Y. Solihin. Memst: Cloning memory behavior using stochastic traces. In *MEMSYS*, pages 146–157, 2015.
[13] R. Bell, R. Bhatia, and L. K. John. Automatic testcase synthesis and performance model validation for high-performance powerpc processors. In *International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 154-165, March*, 2006.
[14] R. H. Bell, Jr. and L. K. John. Improved automatic testcase synthesis for performance model validation. In *ICS*, 2005.
[15] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.
[16] J. Chen, D. Kaseridis, and L. K. John. Modeling program resource demand using inherent program characteristics. In *Proceedings of ACM SIGMETRICS*, 2011.
[17] C. Chou, A. Jaleel, and M. K. Qureshi. Bear: Techniques for mitigating bandwidth bloat in gigascale dram caches. In *ISCA*, pages 198–210, 2015.
[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, pages 143–154, 2010.
[19] E. Deniz, A. Sen, B. Kahne, and J. Holt. Minime: Pattern-aware multicore benchmark synthesizer. *IEEE Transactions on Computers*, 64(8):2239–2252, 2015.
[20] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. *SIGPLAN Not.*, 38(5):245–257, May 2003.
[21] L. Eeckhout, R. J. Bell, B. Stougie, K. D. Bosschere, and L. K. John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In *ISCA, Munich, Germany, pp. 350-361*, 2004.
[22] L. Eeckhout, K. de Bosschere, and H. Neefs. Performance analysis through synthetic trace generation. In *ISPASS*, pages 1–6, 2000.
[23] L. Eeckhout, Y. Luo, K. Bosschere, and L. K. John. BLRL: Accurate and efficient warmup for sampled processor simulation. In *The Computer Journal. Vol. 48. No. 4, May*, 2005.
[24] L. V. Ertvelde and L. Eeckhout. Benchmark synthesis for architecture and compiler exploration. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–11, 2010.
[25] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: A study of scale-out workloads on modern hardware. *SIGPLAN Not.*, 47(4), Mar. 2012.
[26] K. Ganesan, J. Jo, W. L. Bircher, D. Kaseridis, Z. Yu, and L. K. John. Sympo: A systematic approach for escalating system-level power consumption using synthetic benchmarks. In *19th IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT), Vienna, Austria*, 2010.
[27] K. Ganesan, J. Jo, and L. John. Synthesizing memory-level parallelism aware miniature clones for spec cpu2006 and implantbench workloads. In *ISPASS*, 2010.
[28] K. Ganesan and L. K. John. Maximum multicore power (mampo) - an automatic multithreaded synthetic power virus generation framework for multicore systems. In *ACM SuperComputing Conference (SC 2011)*, 2011.
[29] W. Gao, Y. Zhu, Z. Jia, C. Luo, L. Wang, Z. Li, J. Zhan, Y. Qi, Y. He, S. Gong, X. Li, S. Zhang, and B. Qiu. Bigdatabench: a big data benchmark suite from web search engines. *CoRR*, abs/1307.0320, 2013.
[30] N. Gulur, M. Mehandale, R. Manikantan, and R. Govindarajan. ANATOMY: An analytical model of memory system performance. In *Proceedings of ACM SIGMETRICS*, 2014.
[31] L. K. John. Vawiram: A variable width random access memory module. In *9th International Conference on VLSI Design, pp. 219-224, January*, 1996.
[32] A. Joshi, L. Eeckhout, R. H. Bell, and L. John. Performance cloning: A technique for disseminating proprietary applications as benchmarks. In *IEEE IISWC*, pages 105–115, Oct 2006.
[33] A. Joshi, L. Eeckhout, and L. John. The return of synthetic benchmarks. In *2008 SPEC Benchmark Workshop*, pages 1–11, 1 2008.
[34] A. Joshi, L. Eeckhout, L. K. John, and C. Isen. Automated microprocessor stressmark generation. In *IEEE International High Performance Computer Architecture (HPCA) Symposium*, 2008.
[35] A. Joshi, L. John, J. Yi, R. H. J. Bell, L. Eeckhout, and D. Lilja. Evaluating the efficacy of statistical simulation for design space exploration. In *International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 70-79, March*, 2006.
[36] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John. Measuring benchmark similarity using inherent program characteristics. *IEEE Transactions on Computers*, 55(6):769–782, 2006.
[37] Y. Kim, W. Yang, and O. Mutlu. Ramulator: A fast and extensible dram simulator. In *IEEE Computer Architecture Letters*, 2015.
[38] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, pages 357–368, 1998.
[39] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.
[40] R. K. V. Maeda, Q. Cai, and J. Xu. Fast and accurate exploration of multi-level caches using hierarchical reuse distance. In *HPCA*, 2017.
[41] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, June 1970.
[42] O. Mutlu. Memory scaling: A systems architecture perspective. In *MemCon*, 2013.
[43] A. Nair and L. K. John. Simulation points for SPEC CPU 2006. In *26th International Conference on Computer Design, ICCD*, pages 397–403, 2008.
[44] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *HPCA*, 2004.
[45] M. Oskin, F. T. Chong, and M. Farrens. Hls: Combining statistical and symbolic simulation to guide microprocessor designs. In *ISCA*, pages 71–82, 2000.
[46] R. Panda, C. Erb, M. Lebeane, J. Ryoo, and L. K. John. Performance characterization of modern databases on out-of-order cpus. In *IEEE SBAC-PAD*, 2015.
[47] R. Panda, D. Jimenez, and P. V. Gratz. B-fetch: Branch prediction directed prefetching for in-order processors. *IEEE Computer Architecture Letters*, 11:41–44, 2012.
[48] R. Panda and L. John. Proxy benchmarks for emerging big-data workloads. In *The 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.
[49] R. Panda and L. K. John. Data analytics workloads: Characterization and similarity analysis. In *International Performance, Computers, and Communications Conference (IPCCC)*, pages 1–9. IEEE Computer Society, 2014.
[50] R. Panda and L. K. John. Proxy benchmarks for emerging big-data workloads. In *IEEE ISPASS*, 2017.
[51] R. Panda, S. Song, J. Dean, and L. K. John. Wait of a decade: Did SPEC CPU2017 benchmarks broaden the performance spectrum? In *HPCA*, 2018.
[52] R. Panda, X. Zheng, A. Gerstlauer, and L. K. John. CAMP: Accurate modeling of core and memory locality for proxy generation of big-data applications. In *DATE*, 2018.
[53] R. Panda, X. Zheng, and L. K. John. Accurate address streams for llc and beyond (slab): A methodology to enable system exploration. In *IEEE ISPASS*, 2017.
[54] R. Panda, X. Zheng, J. Wang, A. Gerstlauer, and L. John. Statistical pattern based modeling of GPU memory access streams. In *ACM Design Automation Conference (DAC)*, 2017.
[55] J. H. Ryoo, M. R. Meswani, R. Panda, and L. K. John. (POSTER) SILC-FM: subblocked interleaved cache-like flat memory. In *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2016.
[56] A. Seznec. A new case for the TAGE branch predictor. In *IEEE MICRO*, 2011.
[57] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst.*, 36(5):45–57, Oct. 2002.
[58] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti. Efficiently prefetching complex address patterns. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, 2015.
[59] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 252–263, Washington, DC, USA, 2006. IEEE Computer Society.
[60] L. Van Ertvelde and L. Eeckhout. Dispersing proprietary applications as benchmarks through code mutation. *ASPLOS*, pages 201–210, 3 2008.
[61] J. Wang, R. Panda, and L. K. John. Selsmap: A selective stride masking prefetching scheme. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 369–372, 2017.
[62] Y. Wang, G. Balakrishnan, and Y. Solihin. Metoo: Stochastic modeling of memory traffic timing behavior. *PACT*, pages 457–467, 2015.
[63] J. Weinberg and A. E. Snavely. Accurate memory signatures and synthetic address traces for hpc applications. In *ICS*. ACM, 2008.