Cache Modeling and Optimization using Miniature Simulations

Carl Waldspurger, Trausti Saemundson, and Irfan Ahmad, *CachePhysics, Inc.;* Nohhyun Park, *Datos IO, Inc.*

https://www.usenix.org/conference/atc17/technical-sessions/presentation/waldspurger

This paper is included in the Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

Open access to the Proceedings of the 2017 USENIX Annual Technical Conference is sponsored by USENIX.

Cache Modeling and Optimization using Miniature Simulations

Carl A. Waldspurger
CachePhysics, Inc.
carl@cachephysics.com

Trausti Saemundson
CachePhysics, Inc.
trauzti@gmail.com

Irfan Ahmad
CachePhysics, Inc.
irfan@cachephysics.com

Nohhyun Park
Datos IO, Inc.
nohhyun.park@datos.io

Abstract

Recent approximation algorithms (*e.g.*, CounterStacks, SHARDS and AET) make lightweight, continuously-updated miss ratio curves (MRCs) practical for online modeling and control of LRU caches. For more complex cache-replacement policies, *scaled-down simulation*, introduced with SHARDS, offers a general method for emulating a given cache size by using a miniature cache processing a small spatially-hashed sample of requests.

We present the first detailed study evaluating the effectiveness of this approach for modeling non-LRU algorithms, including ARC, LIRS and OPT. Experiments with over a hundred real-world traces demonstrate that scaled-down MRCs are extremely accurate while requiring dramatically less space and time than full simulation.

We propose an efficient, generic framework for dynamic optimization using multiple scaled-down simulations to explore candidate cache configurations simultaneously. Experiments demonstrate significant improvements from automatic adaptation of parameters including the stack size limit in LIRS, and queue sizes in 2Q.

Finally, we introduce *SLIDE*, a new approach inspired by Talus that uses scaled-down MRCs to remove performance cliffs automatically. SLIDE performs shadow partitioning transparently within a single unified cache, avoiding the problem of migrating state between distinct caches when partition boundaries change. Experiments demonstrate that SLIDE improves miss ratios for many cache policies, with large gains in the presence of cliffs.

1 Introduction

Caches are ubiquitous in modern computing systems, improving system performance by exploiting locality to reduce access latency and offload work from contended storage systems and interconnects. However, caches are notoriously difficult to model. It is well-known that performance is non-linear in cache size, due to complex effects that vary enormously by workload. Techniques for accurate and efficient cache modeling are especially valuable to inform cache allocation and partitioning decisions, optimize cache parameters, and support goals including performance, isolation, and quality of service.

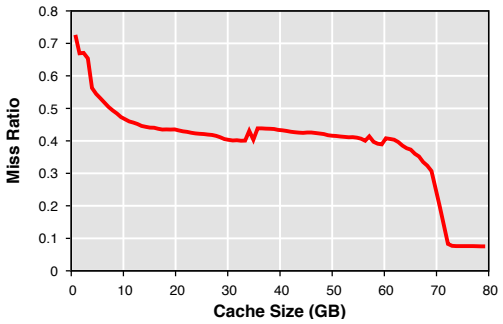

Figure 1: **Example MRC.** Miss-ratio curve for a production disk block trace using ARC cache algorithm. The ratio of cache misses to total references is plotted as a function of cache size.

1.1 Cache Modeling

Cache utility curves plot a performance metric as a function of cache size. Figure 1 shows an example miss-ratio curve (MRC), which plots the ratio of cache misses to total references for a workload (y-axis) as a function of cache size (x-axis). The miss ratio generally decreases as cache size increases, although complex algorithms such as ARC [14] and LIRS [9] can exhibit non-monotonic behavior due to imperfect dynamic adaptation.

MRCs are valuable for analyzing cache behavior. Assuming a workload exhibits reasonable stationarity at the time scale of interest, its MRC can also predict future performance. Thus, MRCs are powerful tools for optimizing cache allocations to improve performance and achieve service-level objectives [3, 11, 18, 22, 27, 28].

1.2 MRC Construction

Before the seminal paper by Mattson *et al.* [13], studies of memory and storage caching required running separate experiments for each cache size. Their key insight was that many replacement policies exhibit an inclusion property: given a cache C of size S, $C(S) \subseteq C(S+1)$. Such policies, which include LRU, LFU, and MRU, are referred to as *stack algorithms*. Mattson *et al.* introduced a method for such algorithms that constructs the entire MRC for all cache sizes in a single pass over a trace.

For a trace of length N containing M unique blocks, Mattson's algorithm takes $O(NM)$ time and $O(M)$ space. Efficient modern implementations of this algorithm have

an asymptotic cost of $O(N \log M)$ time and $O(M)$ space, employing a balanced tree to compute reuse distances and a hash table to accelerate lookups into this tree [16].

Recent advances [7, 20, 23, 26] have produced approximate methods that construct accurate MRCs with dramatically lower costs than exact methods. In particular, SHARDS [23] and AET [7] require only $O(N)$ time and $O(1)$ space, with a tiny footprint of approximately 1 MB.

Previously relegated to offline modeling, MRCs for stack algorithms can now be computed so inexpensively that they are practical for dynamic, online cache management, even in the most demanding environments. However, for more complex non-stack algorithms, such as ARC and LIRS, there are no known single-pass methods. As a result, separate runs are required for each cache size, similar to pre-Mattson modeling of LRU caches.

1.3 Cache Optimization

Low-cost online modeling of cache behavior using MRCs has many practical applications. Whereas a single cache instance runs with a single policy and a single set of configuration parameters, the ability to efficiently instantiate multiple concurrent models with different cache configurations offers a powerful generic framework for dynamic optimization. By simulating candidate cache configurations simultaneously, a system can quantify the impact of hypothetical parameter changes, so that the best settings can be applied to the actual cache.

This approach has the potential to overcome a key challenge in designing cache software today: policy and parameter tweaking is typically performed only at design time, considering a small number of benchmarks. Since no single configuration is best for all workloads, there is a significant optimization opportunity to instead adapt parameters automatically in live deployments.

A multi-model approach can help select the best general options, such as cache block size, write policy, or even replacement policy. The same method supports dynamic tuning of algorithm-specific parameters, such as queue sizes for 2Q [10] or LIRS [9].

Lightweight MRCs can be futher leveraged to guide efficient cache sizing, allocation, and partitioning for both individual workloads and complex multi-workload environments. For example, Talus shadow partitioning [3], which requires an MRC as input, can remove performance cliffs within a single workload, and improve cache partitioning across workloads.

1.4 Contributions

We make several key contributions over prior research in the areas of cache modeling and optimization:

Evaluate scaled-down simulation for complex policies To the best of our knowledge, scaled-down simulation is the *only* general approach capable of fast and accurate modeling of complex caching algorithms. We present the first detailed evaluation with non-LRU caching algorithms, including ARC, LIRS, and OPT. Our results indicate that sampling rates as low as 0.1% yield accurate MRCs with approximate miss ratio errors averaging much less than 0.01, at extremely low overhead.

New optimization framework We introduce a powerful new framework for optimizing cache performance dynamically by leveraging miniature cache simulations.

Transparent cliff removal We highlight challenges with Talus shadow partitioning for non-stack algorithms, and introduce SLIDE, a new approach that removes performance cliffs from such algorithms automatically and transparently – the first practical application of cliff removal techniques to complex cache algorithms.

New LIRS observations We describe previously-unreported parameter sensitivity and non-monotonic behavior with LIRS, and present a useful new optimization.

Although we focus on block-based storage systems, our techniques are broadly applicable to nearly any form of caching, including memory management in operating systems and hypervisors, application-level caches, key-value stores, and even hardware cache implementations.

The next section provides some background on non-stack caching algorithms. Section 3 describes our core scaled-down cache modeling technique, and presents a detailed evaluation of its accuracy and performance. Scaled-down caches are leveraged to optimize LIRS and 2Q by adapting algorithm parameters in Section 4. Section 5 introduces SLIDE, a new approach for removing performance cliffs, and demonstrates improvements with several cache policies. Related work is discussed in Section 6. Finally, we summarize our conclusions and highlight opportunities for future work in Section 7.

2 Non-stack Algorithms

Many modern caching algorithms outperform LRU on a wide range of workloads. Several, such as ARC, LIRS, and 2Q, treat blocks that have recently been seen only once differently from those that have been seen at least twice. Many policies employ *ghosts* – small metadata-only entries containing block identifiers, but not actual data. Some, like ARC, adapt to changes in workload patterns automatically. It is not surprising that such sophisticated policies are *non-stack algorithms* that violate the stack inclusion property. All caching algorithms aspire to close the gap with OPT, the unrealizable optimal policy.

2Q Inspired by LRU-K [17], Johnson and Shasha developed the 2Q algorithm [10]. As its name suggests, 2Q uses two queues: *A1* for blocks seen once and *Am* for blocks seen more than once. *A1* is split into *A1in*

and *A1out*, where *A1out* is a metadata-only ghost extension of *A1in*. 2Q promotes a block to *Am* only on a hit in *A1out*, so *A1in* behaves as a FIFO. The algorithm has two tunable parameters – the size of *A1in* relative to *Am*, and the size of *A1out* relative to the cache size.

ARC Megiddo and Modha introduced ARC, the adaptive replacement cache policy [14]. ARC is a self-tuning algorithm that manages both recently-used and frequently-used blocks in separate LRU lists: *T1* for blocks seen once, *T2* for blocks seen more than once, and their corresponding ghost extensions, *B1* and *B2*, which track metadata for recently-evicted blocks. Queue sizes change adaptively based on which gets more cache hits; there are no tunable parameters. ARC has been deployed widely in production systems, and is considered by many to be the "gold standard" for storage caching.

LIRS Jiang and Zhang developed LIRS, the low inter-reference recency set algorithm [9]. LIRS uses recency to estimate reuse distance when making replacement decisions. Blocks are categorized into high reuse-distance (HIR) and low reuse-distance (LIR) sets. All LIR blocks are resident but HIR blocks can be resident or ghosts. A block changes from HIR to LIR when its reuse distance is low compared to the current LIR set.

LIRS employs two LRU lists, called the S and Q stacks. Q contains all resident HIR blocks, and S contains LIR blocks as well as some resident HIR blocks and HIR ghosts. LIRS has two tunable parameters – the ratio of resident HIR and LIR blocks (the authors suggest 1% HIR), and the maximum size of S, which effectively bounds the number of ghosts. LIRS has been adopted by several production systems, including MySQL [25].

OPT Belady first described OPT, the theoretically optimal algorithm, also known as MIN [4, 1, 13]. OPT is a "clairvoyant" algorithm, since it relies on knowledge of future references to evict the block that will be reused the farthest in the future. Although OPT is actually a stack algorithm [21], it cannot be used to implement online eviction. Instead, OPT provides a bound on the performance of realizable algorithms.

3 Scaled-down Modeling

SHARDS [23] introduced single-pass techniques for constructing approximate MRCs based on randomized spatial sampling. References to representative locations are selected dynamically based on a function of their hash values. The "scaled down" reference stream is provided as input to a conventional single-pass MRC construction algorithm [13, 16] and the reuse distances it outputs are "scaled up" to adjust for the sampling rate.

While this approach works extremely well for stack algorithms such as LRU, there is no known single-pass method for non-stack caching algorithms. For such policies, a discretized MRC must be constructed by running separate simulations at many different cache sizes.

To support efficient modeling of *any* caching algorithm, the SHARDS authors proposed emulating a given cache size using a *miniature cache* running the full, unmodified algorithm over a small spatially-hashed sample of requests. Although a proof-of-concept experiment yielded promising results [23], there has been no detailed study of this approach. We present the first comprehensive evaluation of scaled-down simulation for modeling the sophisticated ARC, LIRS and OPT algorithms.

3.1 Miniature Simulations

A miniature simulation can emulate a cache with any specified size by scaling down both the actual cache size and its input reference stream. For example, consider modeling a cache with size S using a sampling rate R. A miniature simulation may emulate a cache of size S by scaling down the cache size to $R \cdot S$ and scaling down the reference stream using a hash-based spatial filter with sampling rate R. In practice, sampling rates on the order of $R = 0.01$ or $R = 0.001$ yield very accurate results, achieving huge reductions in space and time compared to a conventional full-size simulation.

More generally, scaled-down simulation need not use the same scaling factor for both the miniature cache size and its reference stream, although such configurations were not discussed when the technique was originally proposed [23]. The emulated cache size S_e, mini-cache size S_m, and input sampling rate R are related by $S_e = S_m/R$. Thus, S_e may be emulated by specifying a fixed rate R, and using a mini-cache with size $S_m = R \cdot S_e$, or by specifying a fixed mini-cache size S_m, and sampling its input with rate $R = S_m/S_e$. In practice, it is useful to enforce reasonable constraints on the minimum mini-cache size (*e.g.*, $S_m \geq 100$) and sampling rate (*e.g.*, $R \geq 0.001$) to ensure sufficient cache space and enough sampled references to simulate meaningful behavior.

3.1.1 Error Reduction

Like SHARDS, we apply a simple adjustment to reduce sampling error when computing the miss ratio for a miniature simulation. We have observed that when the number of sampled references, N_s, differs from the expected number, $E[N_s] = N \cdot R$, the sample set typically contains the wrong proportion of frequently-accessed blocks. To correct for this bias we divide the number of misses m by the *expected* number of references, instead of the actual number of references; *i.e.*, $m/E[N_s]$ is a better approximation of the true miss ratio than m/N_s.

3.1.2 Caches with Integrated Modeling

We have experimented with an alternative "unified" approach that integrates MRC construction into a live pro-

duction cache, without running separate simulations. Spatial hashing shards requests across a set of cache partitions, all serving actual requests. Several small partitions serve as *monitoring shards*, emulating multiple cache sizes within a small fraction of the overall cache.

An MRC can be generated on demand by simply accessing the miss ratios associated with each monitoring shard. Although integrated monitoring avoids additional simulation costs, we found that it typically degrades overall cache performance slightly, since most monitoring shards will not have efficient operating points.

3.2 Scaled-down MRCs

For non-stack algorithms, there are no known methods capable of constructing an entire MRC in a single pass over a trace. Instead, MRC construction requires a separate run for each point on the MRC, corresponding to multiple discrete cache sizes. Fortunately, we can leverage miniature caches to emulate each size efficiently.

We evaluate the accuracy and performance of our approach with three diverse non-LRU cache replacement policies: ARC, LIRS, and OPT. We developed efficient implementations of each in C, and validated their correctness against existing implementations [6, 8, 19].

We use a collection of 137 real-world storage block trace files, similar to those used in the SHARDS evaluation. These represent 120 week-long virtual disk traces from production VMware environments collected by CloudPhysics [23], 12 week-long enterprise server traces collected by Microsoft Research Cambridge [15], and 5 day-long server traces collected by FIU [12].

For our experiments, we use a 16 KB cache block size, and misses are read from storage in aligned, fixed-size 16 KB units. Reads and writes are treated identically, effectively modeling a simple write-back caching policy. We have also experimented with 4 KB blocks, modeling different write policies, and processing only read requests, all with similar results.

3.2.1 Accuracy

For each trace, we compute MRCs at 100 discrete cache sizes, spaced uniformly between zero and a maximum cache size. To ensure these points are meaningful, the maximum cache size is calculated as the aggregate size of all unique blocks referenced by the trace. This value was estimated during a separate, one-time pre-processing step for each trace, using fixed-size SHARDS [23].

To quantify accuracy, we compute the difference between the approximate and exact miss ratios at each discrete point on the MRC, and aggregate these into a mean absolute error (MAE) metric, as in related work [26, 23, 7]. The box plots[1] in Figure 2 show the MAE distributions for ARC, LIRS, and OPT with sampling rates $R = 0.01$ and $R = 0.001$. The average error is surprisingly small in all cases. For $R = 0.001$, the median MAE for each algorithm is below 0.005, with a maximum of 0.033. With $R = 0.01$, the median MAE for each algorithm is below 0.002, with a maximum of 0.012.

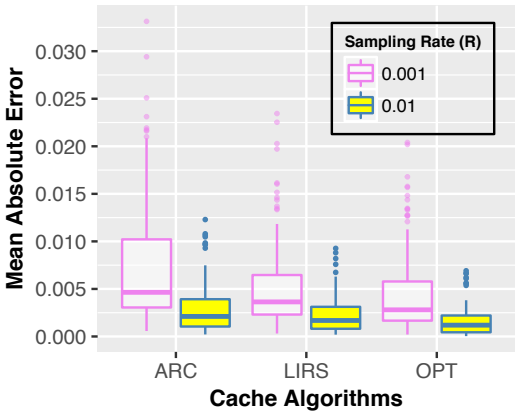

Figure 2: **Error Analysis.** Distribution of mean absolute error for all 137 traces with three algorithms (ARC, LIRS, OPT) at two different sampling rates ($R = 0.01$, $R = 0.001$).

Since the minimum cache size for LIRS is 200 blocks (to support the default 1% allocation to HIR), the LIRS MAE was calculated using this minimum size for some miss ratios, implying a higher sampling rate. Excluding these min-size runs, the median MAE for $R = 0.001$ is below 0.003, with a maximum of 0.025; for $R = 0.01$, the median is below 0.002, with a maximum of 0.009.[2]

Figure 3 contains 35 small plots that illustrate the accuracy of approximate MRCs with $R = 0.001$ on example traces with diverse MRC shapes and sizes. In most cases, the approximate and exact curves are nearly indistinguishable. The plots in Figure 4 show this accuracy with much greater detail for two example MSR traces. In all cases, miniature simulations model cache behavior accurately, including complex non-monotonic behavior by ARC and LIRS. These compelling results with such diverse algorithms and workloads suggest that scaled-down simulation is an extremely general technique capable of modeling nearly any caching algorithm.

3.2.2 Performance

For our performance evaluation, we used a platform configured with a six-core 3.3 GHz Intel Core i7-5820K processor and 32 GB RAM, running Ubuntu 14.04 (Linux kernel 4.4). Experiments compare traditional exact simulation with our lightweight scaled-down approach.

Resource consumption was measured using our five largest traces. We simulated three cache algorithms at

[1] The top and the bottom of each box represent the first and third quartiles. The thin whiskers show the min and max, excluding outliers.

[2] The reduced MAE may seem counter-intuitive. However, accuracy generally improves as the size of the scaled-down cache increases, and the excluded points were the smaller, less-accurate cache sizes.

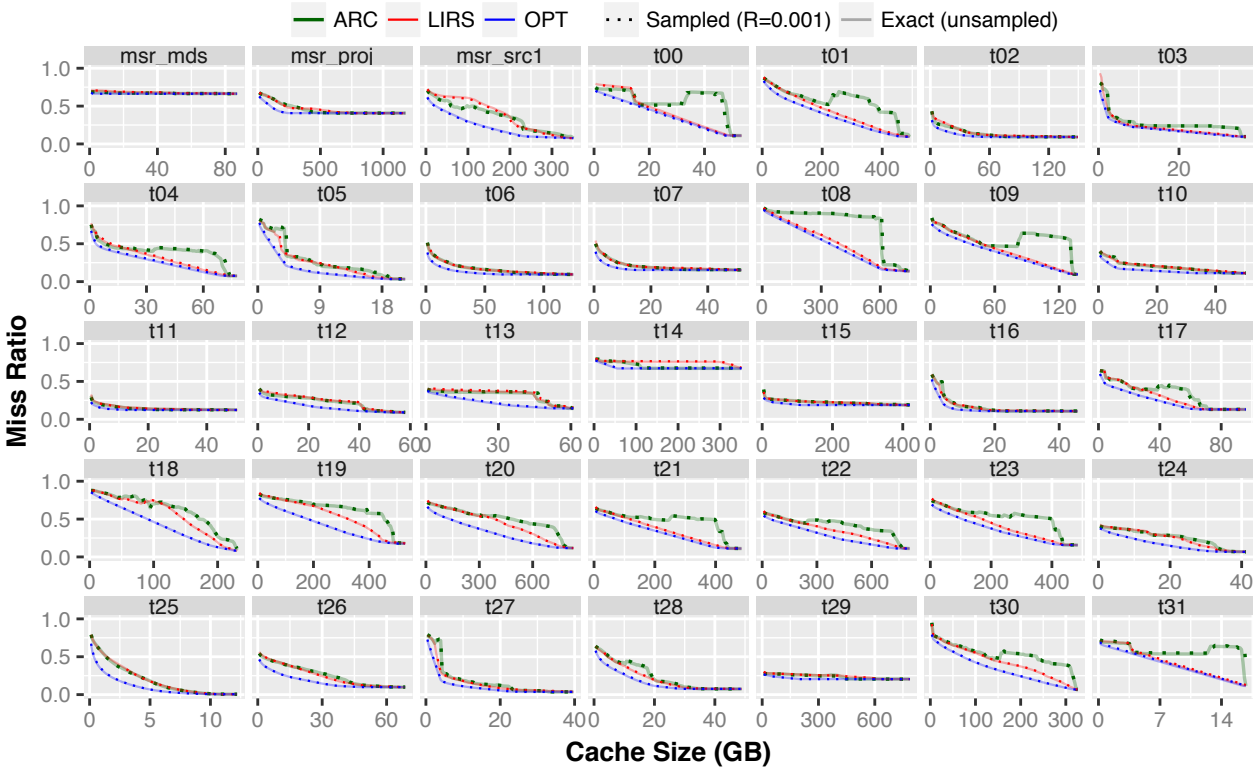

Figure 3: **Diverse MRCs: Exact vs. Miniature.** Exact and approximate MRCs for 35 representative traces: three named MSR workloads [15], and the CloudPhysics workloads labeled `t00`–`t31` in the SHARDS evaluation [23]. Approximate MRCs are constructed using scaled-down simulation with sampling rate $R = 0.001$. Each color represents a different cache algorithm.

five emulated sizes S_e (8 GB, 16 GB, 32 GB, 64 GB and 128 GB), using multiple sampling rates R (1, 0.1, 0.01 and 0.001) for a total of 60 experiments per trace. We repeated each run five times, and report average values.

Unsurprisingly, the memory footprint[3] for cache simulation is a simple linear function consisting of fixed overhead (for policy code, libraries, etc.) plus variable space. For ARC and LIRS, the variable component is proportional to the cache size, $R \cdot S_e$. For OPT, which must track all future references, it is proportional to the number of sampled references, $R \cdot N$. Table 1 reports the fixed and variable components of the memory overhead determined by linear regression ($r^2 > 0.99$). As expected, accurate results with $R = 0.001$ require 1000× less space than full simulation, excluding the fixed overhead.

We also measured the CPU usage[4] consumed by our single-threaded cache implementations with both exact and scaled-down simulations for ARC, LIRS and OPT. As shown in Figure 5, runtime consists of two main components: cache simulation time, which is roughly linear in R, and the sampling overhead involving hashing and trace file I/O, which is roughly constant; MurmurHash [2] is invoked for each reference to determine if it should be sampled. The total runtime from each experiment was decomposed by running a corresponding experiment with a pre-sampled trace file, removing the overhead of hashing and most I/O costs.

	Linear Function		Example Trace (t22)	
Policy	**Fixed**	**Variable**	**R=0.001**	**R=1**
ARC	1.37 MB	71 B	1.57 MB	284 MB
LIRS	1.59 MB	75 B	1.80 MB	301 MB
OPT	7.10 MB	37 B	19.55 MB	18,519 MB

Table 1: **Memory Footprint.** Memory usage for ARC and LIRS is linear in the cache size, $R \cdot S_e$, while for OPT, it is linear in the number of sampled references, $R \cdot N$. Measured values are shown for CloudPhysics trace `t22` with S_e = 64 GB.

Overall, scaled-down simulation with $R = 0.001$ requires about 10× less CPU time than full simulation, and achieves throughput exceeding 53 million references per second for ARC and LIRS, and 39 million references per second for OPT. Fortunately, for multi-model optimization, hash-based sampling costs are incurred only once, not for each mini-cache. In an actual production cache, the cost of data copying would dwarf the hashing overhead, which represents a larger fraction of the fast cache simulation operations that manipulate only metadata.

[3]The peak resident set size was obtained from the Linux procfs node `/proc/<pid>/status` immediately before terminating.

[4]CPU time was obtained by adding the user and system time components reported by `/usr/bin/time`.

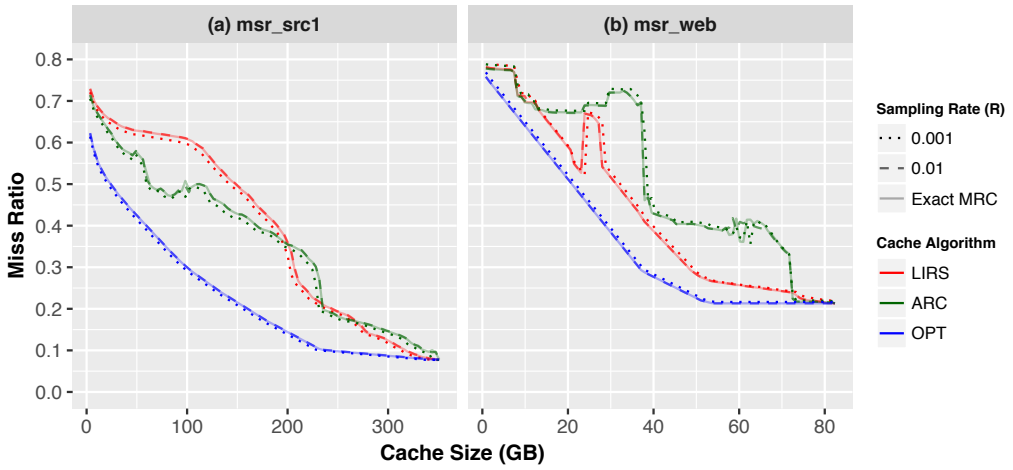

Figure 4: **Detailed MRCs.** Approximate MRCs for two enterprise storage traces [15] with three different algorithms. Miniature simulations capture cache behavior accurately, including complex non-monotonicity. All MAEs for $R = 0.001$ are less than 0.011.

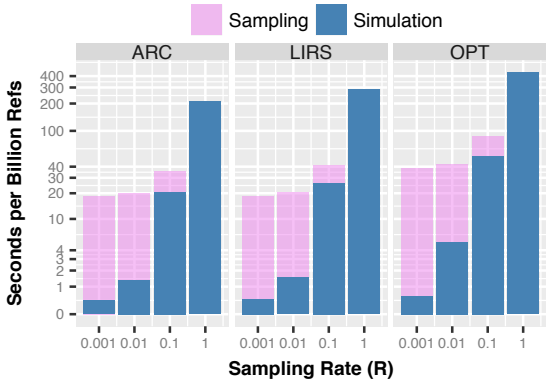

Figure 5: **Runtime.** Time to process one billion references as a function of R. *Sampling* represents time spent selecting references to simulate; *Simulation* is time spent actually executing the cache policy.

4 Adapting Cache Parameters

Our generic multi-model optimization framework leverages miniature simulations to adapt cache parameters dynamically. The impact of multiple candidate parameter values is evaluated periodically, and the best setting is applied to the actual cache. We present example optimizations that adapt cache parameters automatically for two well-known replacement algorithms: LIRS and 2Q.

While MRCs are typically stable over short time periods, they frequently vary over longer intervals. To adapt dynamically to changing workload behavior, we divide the input reference stream into a series of *epochs*. Our experiments use epochs consisting of one million references, although many alternative definitions based on wall-clock time, evictions, or other metrics are possible.

After each epoch, we calculate an exponentially-weighted moving average (EWMA) of the miss ratio for each mini-cache, to balance historical and current cache behavior. Our experiments use an EWMA weight of 0.2 for the current epoch. The parameter value associated with the mini-cache exhibiting the lowest smoothed miss ratio is applied to the actual cache for the next epoch.

4.1 LIRS Adaptation

As discussed in Section 2, the LIRS S stack is LRU-ordered and contains a mix of LIR blocks, resident HIR blocks and non-resident HIR blocks (ghosts). This queue tracks the internal stack-distance ordering between HIR and LIR blocks. A HIR block is reclassified as LIR if it is referenced when it has a stack distance lower than that of the oldest LIR block. During this "status switch", the oldest LIR block is changed to HIR, evicted from S, and inserted into Q. After the status switch, a pruning operation removes all HIR blocks from the tail of S.

The default LIRS algorithm allows S to grow without bound on a sequence of misses. To address this issue, the authors suggest limiting the size of S; to enforce that limit, the oldest HIR ghost is evicted from S once the size exceeds the limit. We denote[5] this bound by f, relative to the cache size c; the total size of S is limited to $c \cdot f$. The LIRS paper experimented with a few values of f and reported that even low values such as $f = 2$ work well. Our evaluation of scaled-down modeling accuracy in Section 3.2 uses $f = 2$, so that LIRS tracks roughly the same number of ghost entries as ARC.

Code Optimization We started our work on LIRS with a C implementation obtained from the authors [8]. However, this code enforced the S size limit by always searching for the oldest HIR ghost starting from the tail of S. Since this caused excessive overhead with our large traces, we developed a simple optimization that stores

[5]This limit was not explicitly named in the LIRS paper.

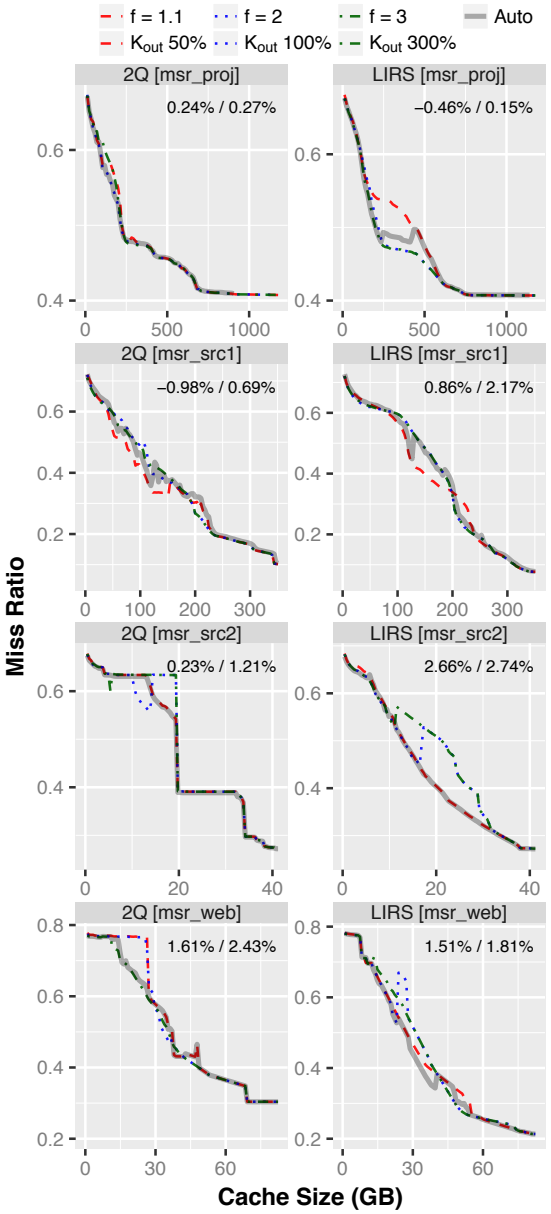

Figure 6: **Adaptive Parameter Tuning.** Dynamic multi-model optimization results for four example traces. Adaptation selects good values for 2Q (*Kout*) and LIRS (f) at most cache sizes with potential gains. The percentages (upper right) show the actual adaptation gain (*vs.* f=2, 50% *Kout*) and the potential gain (best per-cache-size values), averaged over all cache sizes.

a pointer to the entry previous to the last-removed HIR ghost.[6] It is guaranteed that no HIR ghost can appear after this element because entries are always added to the head of S. If the entry associated with this pointer is removed from the middle of S, such as on a hit, it is simply updated to the previous entry in the queue. We describe a similar optimization for SLIDE evictions in Section 5.5.

[6]This optimization met with approval from the LIRS authors [8].

Non-monotonicity Although the authors reported that LIRS does not suffer from non-monotonic behavior [9], we have observed it with several of our workloads when limiting the size of S. For example, Figure 4 reveals a prominent region for the `msr_web` trace where increasing the LIRS cache size results in a higher miss ratio. Interestingly, the degree of non-monotonicity varies with f, and there appear to be workload-dependent values that eliminate this behavior. For example, Figure 6 shows that `msr_src1`, `msr_src2` and `msr_web` perform well with $f = 1.1$, while $f = 3.0$ is best for `msr_proj`.

Automatic Tuning We use our multi-model optimization approach to adapt the LIRS f value dynamically for a subset of the traces described in Section 3.2. For each workload, five scaled-down simulations are performed with different values for f: 1.1, 1.5, 2.0, 2.5 and 3.0. Each simulation emulates the same cache size, equal to the size of the actual cache, with a fixed sampling rate $R = 0.005$. After each epoch consisting of 1M references, the miss ratios for each mini-cache are examined, and the best f value is applied to the actual cache.

Experiments The goal of automatic LIRS adaptation is to find the best value of f for each cache size. These ideal[7] static settings form an MRC that traces the lower envelope of the curves for different static f values. Actual and potential gains are computed as the mean signed difference across all cache sizes relative to the curve with the default fixed value $f = 2$. Potential gain is based on the best static f value for each cache size.

Figure 6 presents results for four representative MSR traces. Among all 12 MSR traces, `msr_src2` shows the best actual and potential gains; the worst case for adaptation is the net loss for `msr_proj`. For `msr_web` and `msr_src2`, adaptation converges on the best $f = 1.1$, and realizes 83–97% of the potential gain. The average actual *vs.* potential improvement across the MSR traces is 0.37% / 0.60%; adaptation captures the majority of possible gains. Results are mixed for traces like `msr_src1`, with some regressions, despite an overall gain. We are experimenting with automatic disabling of ineffective adaptation; early results show a small gain for `msr_proj`.

LIRS Observations For workloads with a single large working-set knee (*e.g.*, trace `t08` in Figure 3), the LIRS and OPT MRCs are often surprisingly close. LIRS appears to trace the convex hull of the LRU curve, slightly above OPT. This behavior is not intuitive, since LIRS has no knowledge of the knee, where the miss ratio drops suddenly once the working set fits. The explanation is that some blocks initially get stuck in the LIR set, and no later blocks have a low enough reuse distance to replace

[7]Although adaptation tends to converge on a single f value, selecting the best value for each individual epoch may yield a lower *dynamic* optimum. However, the combinatorics make this infeasible to simulate.

them. During another pass over the working set, accesses to these blocks will be hits. Thus, LIRS can effectively remove some cliffs by trapping blocks in the LIR set.

4.2 2Q Adaptation

The 2Q algorithm is not adaptive, so its queue sizes are specified manually. The authors suggest allocating 25% of the cache space to *A1in* and 75% to *Am*. They also suggest sizing the number of ghosts in *A1out* to be 50% of the elements in the cache. The 2Q paper defines the parameter *Kin* as the size of *A1in*, and *Kout* as the size of *A1out*, the ghost queue for blocks seen only once.

Comparing 2Q and LIRS, *Am* is similar to the subset of the LIRS *S* stack containing LIR blocks, *A1in* is comparable to the LIRS *Q* stack, and *A1out* is similar to the subset of the LIRS *S* stack containing HIR ghost blocks. While LIRS performs well allocating just 1% of its space to *Q*, 2Q needs a higher percentage for *A1in*. The sizing of *A1out* in 2Q is similar to f-adaptation in LIRS.

Since 2Q does not adapt its queue sizes dynamically, we again employ multi-model optimization, using eight scaled-down simulations with $R = 0.005$, 25% *Kin*, and *Kout* parameters between 50% and 300%. After each epoch consisting of 1M references, the best *Kout* value is applied to the actual cache. Automatic adaptation is designed to find the optimal *Kout* for each cache size. As in Section 4.1, we compute gain relative to the area between the default curve with fixed 50% *Kout* and the lower envelope of all the curves with static *Kout* values.

Figure 6 shows adaptation works well for `msr_web`, which has the best actual and potential gains over all 12 MSR traces; the auto-adapted curve tracks the lower envelope closely, capturing 66% of the possible static gain. For traces like `msr_proj` that are not very sensitive to *Kout*, adaptation shows modest absolute gains. The worst case is the significant loss for `msr_src1`, although the auto-disabling extension mentioned earlier results in a small gain. Averaged over all 12 MSR traces, the actual *vs.* potential improvement is 0.10% / 0.45%.

5 SLIDE

SLIDE is a new approach inspired by Talus [3] that leverages scaled-down MRCs to remove performance cliffs. We describe challenging issues with applying Talus to non-LRU policies, and explain how SLIDE resolves them. We then present efficient SLIDE implementation techniques that support transparent shadow partitioning within a single unified cache.

5.1 Talus Inspiration

Talus [3] is a technique that removes cache performance cliffs using hash-based partitioning. It divides the reference stream for a single workload into two shadow partitions, *alpha* and *beta*, steering a fraction ρ of references to the alpha partition. Each partition can be made to emulate the performance of a smaller or larger cache by controlling its size and input load.

Talus requires the workload's MRC as an input. The partition sizes N_α and N_β, and their respective loads, ρ and $1-\rho$, are computed in a clever manner that ensures their combined aggregate miss ratio lies on the convex hull of the MRC. Although Talus was introduced in the context of hardware processor caches, a similar idea has also been applied to key-value web caches [5].

We view the hash-based partitioning employed by Talus for removing performance cliffs and the hash-based monitoring introduced with SHARDS for efficient MRC construction as two sides of the same coin. Both rely on the property that hash-based sampling produces a smaller reference stream that is statistically self-similar to the original stream. The ability to construct MRCs using hash-based sampling was not recognized by the Talus authors, who emphasized that no known methods could generate MRCs inexpensively for non-stack algorithms.

5.2 Challenges with Non-LRU MRCs

As noted by the Talus authors, a key challenge with applying Talus to non-stack algorithms is the need to construct MRCs efficiently in an online manner. This problem is solved by using the scaled-down modeling techniques described in Section 3. As with parameter adaptation described in Section 4, we divide the input reference stream into a series of epochs. After each epoch, we construct a discretized MRC from multiple scaled-down simulations with different cache sizes, smoothing each miss ratio using an EWMA. We then identify the subset of these miss ratios that form the convex hull for the MRC, and compute the optimal partition sizes and loads using the same inexpensive method as Talus.

In theory, the combination of scaled-down MRC construction and Talus shadow partitioning promises to improve the performance of *any* caching policy by interpolating efficient operating points on the convex hulls of workload MRCs. In practice, we encountered several additional challenges while trying to implement Talus for caching algorithms such as ARC and LIRS.

5.3 Challenges with Non-LRU Partitioning

Talus requires distinct cache instances for its separate alpha and beta partitions, which together have a fixed total size. This hard division becomes problematic in a system where the partition boundaries change dynamically in response to an MRC that evolves over time. Similarly, when ρ changes dynamically, some cache entries may reside in the "wrong" partition based on their hash values.

These issues were not discussed in the Talus paper.

We initially tested simple strategies to address these issues. For example, removing cache entries eagerly when decreasing the size of a partition, and reallocating the reclaimed space to the other partition. Or migrating entries across partitions eagerly to ensure that each resides in the correct partition associated with its hash. Such eager strategies performed poorly, as migration checks and actions are expensive, and data may be evicted from one partition before the other needs the space. Moreover, it's not clear how migrated state should be integrated into its new partition, even for a simple policy like LRU, since list positions are not ordered across partitions.

A lazy strategy for reallocation and migration generally fares better. Cache entries can be reclaimed from an over-quota partition on demand, and entries residing in incorrect partitions migrated only on hits. However, this approach adds non-trivial complexity to the core caching logic. More importantly, while migrating to the MRU position on a hit seems reasonable for an LRU policy, it's not clear how to merge state appropriately for more general algorithms. Some policies do not even specify how to transform state to support dynamic resizing.

5.4 Transparent Shadow Partitioning

We developed a new approach called *SLIDE* (*S*harded *L*ist with *I*nternal *D*ifferential *E*viction) to address these challenges. In contrast to Talus, SLIDE maintains a single unified cache, and defers partitioning decisions until eviction time. SLIDE conveniently avoids the resizing, migration, and complexity issues discussed above.

A *SLIDE list* is a new abstraction that serves as a drop-in replacement for the standard LRU list used as a common building block by many sophisticated algorithms, including ARC, LIRS, and 2Q. Since SLIDE interposes on primitive LRU operations that add (insert-at-head), reference (move-to-head), and evict (remove-from-tail) entries, it is completely transparent to cache-replacement decisions. An unmodified cache algorithm can support Talus-like partitioning by simply relinking to substitute SLIDE lists for LRU[8] lists. We have successfully optimized ARC (*T1*, *T2*, *B1* and *B2*), LIRS (*S* and *Q*), 2Q (*Am*, *A1in* and *A1out*), and LRU in this manner.

5.5 SLIDE Lists

A SLIDE list is implemented by extending a conventional doubly-linked LRU list. All list entries remain ordered from MRU (head) to LRU (tail). Each entry is augmented with a compact hash[9] of its associated location identifier (*e.g.*, block number or memory address).

[8]SLIDE also works with FIFO lists, such as the 2Q *A1in* queue; a referenced entry simply isn't moved to the head of the list.

[9]Our default implementation uses small 8-bit hashes, providing better than 0.4% resolution for performing hash-based partitioning.

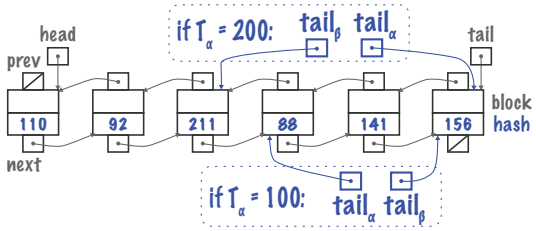

Figure 7: **SLIDE List.** Extensions (highlighted) to a doubly-linked list include hash values plus $tail_\alpha$ and $tail_\beta$ pointers used to find the LRU blocks in the alpha and beta partitions. Different dynamic thresholds T_α illustrate flexible partitioning.

This hash value is compared to the current threshold T_α to classify the entry as belonging to either the alpha or beta "partition". This makes the SLIDE partition boundary more flexible than the hard partitions in Talus.

As depicted in Figure 7, in addition to the usual head and tail pointers, SLIDE maintains two new tail pointers, $tail_\alpha$ and $tail_\beta$. To evict from alpha, the LRU alpha entry is located by walking the list backwards from $tail_\alpha$ until an entry with a hash value below T_α is found. Similarly, an eviction from beta starts with $tail_\beta$ and continues until an entry with a hash value at or above T_α is found.

The tail-search pointers $tail_\alpha$ and $tail_\beta$ are initialized to `NULL`, which indicates that searches should begin from the absolute tail of the combined list. They are updated lazily during evictions, and to skip over entries that are moved to the MRU position on a hit. Since entries are added only to the head of the LRU-ordered list, the amortized cost for these updates is $O(1)$, as each tail pointer traverses a given entry only once.

Many LRU implementations maintain a count of the number of entries in the list. A SLIDE list also tracks N_α, the number of entries that currently belong to the alpha partition. The SLIDE configuration operation specifies both ρ and a target size for alpha, expressed as a fraction F_α of the total number of entries, N_{tot}. During an eviction, an entry is preferentially removed from the alpha partition if it is over quota (*i.e.*, $N_\alpha > F_\alpha \cdot N_{tot}$), or otherwise from the beta partition. If the preferred victim partition is empty, then the absolute LRU entry is evicted.

It is not obvious that substituting SLIDE lists for the internal lists within non-stack algorithms will approximate hard Talus partitions. The basic intuition is that configuring each SLIDE list with identical values of F_α and ρ will effectively divide the occupancy of each individual list – and therefore divide the entire aggregate algorithm state – to achieve the desired split between alpha and beta. As with Talus, this depends on the statistical self-similarity property of hash-based spatial sampling. While SLIDE may differ from strict Talus partitioning, it empirically works well for ARC, LIRS, 2Q, and LRU.

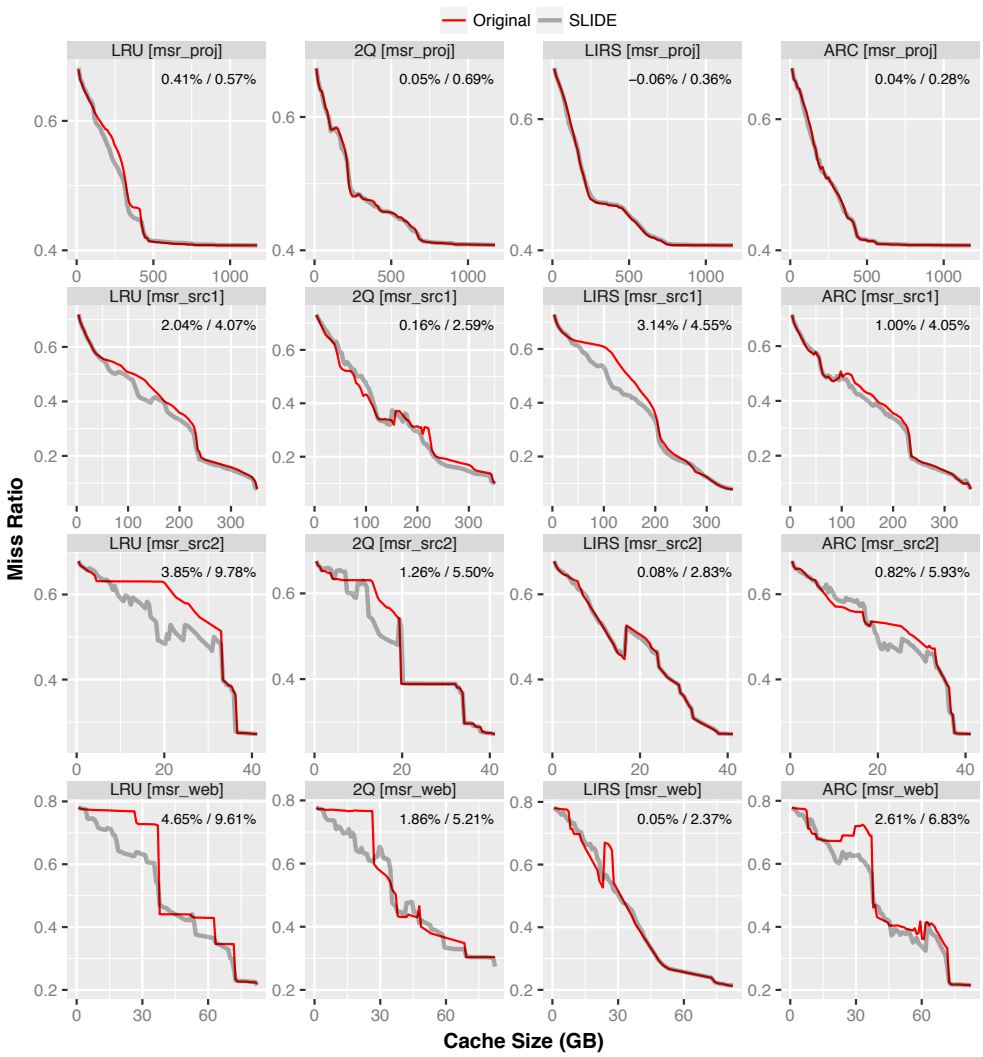

Figure 8: **SLIDE Cliff Removal.** Results for four traces using scaled-down MRCs from seven mini-cache simulations. SLIDE improves the miss ratio for LRU, 2Q, LIRS and ARC caches at most sizes with potential gains, but does exhibit some regressions. The percentages (upper right) show the actual SLIDE gain and the potential gain (ideal convex hull) averaged over all cache sizes.

5.6 SLIDE Reconfiguration

Periodic reconfiguration may move partition boundaries dynamically, changing the threshold T_α. To support constant-time recomputation of N_α, SLIDE optionally maintains an array of counts tracking the number of entries associated with each hash value.[10]

A change to T_α must also reset the $tail_\alpha$ and $tail_\beta$ search pointers, as later entries may have been reclassified to different partitions. Although not guaranteed to be $O(1)$, one pointer must be the same as the global tail, and the expected number of entries the other must re-traverse is $1/F_\alpha$, assuming a uniform hash distribution. This will typically be small compared to the epoch length, even for heavily-skewed partitions; F_α could also be bounded.

[10] An array of 256 counts for our implementation with 8-bit hashes.

5.7 Experiments

We evaluate the effectiveness of SLIDE using a subset of the traces described in Section 3.2. For each workload, a separate experiment is performed at 100 cache sizes. For each size, a discrete MRC is constructed via multiple scaled-down simulations with sampling rate $R = 0.005$. SLIDE is reconfigured after each 1M-reference epoch, using 0.2 as the EWMA weight for the current epoch.

Seven emulated cache sizes are positioned exponentially around the actual size, using relative scaling factors of 1/8, 1/4, 1/2, 1, 2, 4, and 8. For $R = 0.005$, the mini-cache metadata is approximately 8% of the actual metadata size (R times the sum of the scaling factors). For a 16 KB cache block size and 64 B metadata entries, this represents less than 0.04% of total memory consumption.

Many alternative configurations can provide differ-

ent time-space tradeoffs, *e.g.*, fixed-size variable-*R* mini-caches, as described in Section 3.1. Similarly, increasing the number of emulated cache sizes generally yields more accurate MRCs and improves SLIDE results, at the cost of additional mini-cache resource consumption.

Figure 8 plots the results of SLIDE performance cliff removal for four representative MSR traces with LRU[11], 2Q, LIRS and ARC policies. Ideally, SLIDE would trace the convex hull of the original MRC. In practice, this is not attainable, since the MRC evolves dynamically, and its few discrete points yield a crude convex hull. For each plot, we show both the actual SLIDE gain and the potential gain on the convex hull, each computed as the mean signed difference across all cache sizes from the original curve. We also characterize the larger set of all 12 MSR traces, although this metric often averages out more significant differences visible in the plots.

As expected, gains are largest for workloads with non-trivial cliffs, such as `msr_src2` and `msr_web`; SLIDE reduces their miss ratios by more than 10% in many cases. For the larger set of MSR traces, the best-case actual *vs.* potential gains are 4.65% / 9.78% (LRU), 1.86% / 5.50% (2Q), 3.14% / 4.55% (LIRS) and 2.61% / 6.84% (ARC). The average actual *vs.* potential improvements are 0.88% / 2.09% (LRU), 0.26% / 1.30% (2Q), 0.23% / 0.89% (LIRS) and 0.36% / 1.47% (ARC). Overall, SLIDE captures a reasonable fraction of possible gains.

For traces such as `msr_proj`, where the original MRC is nearly convex, SLIDE provides little improvement. For a few traces like `msr_src1`, results are mixed, with SLIDE improving many policies and cache sizes, but degrading others slightly. Across all 12 MSR traces and all four policies, the worst-case gain is -0.14%. As future work, we are extending SLIDE to disable itself dynamically to prevent losses and yield Pareto improvements.

6 Related Work

Research on cache modeling and MRC construction has focused on LRU and stack algorithms [13, 16, 26, 23, 7]. Modeling non-stack algorithms requires offline cache simulations with extremely high resource consumption, making online analysis and control impractical.

As explained in Section 3, basic scaled-down simulation was first introduced with our prior work on SHARDS [23], but there has been no detailed study of this approach. To the best of our knowledge, scaled-down simulation with miniature caches is the *only* approach that can model complex algorithms efficiently.

Our automatic adaptation is motivated by the observation that no single set of cache parameters performs well across all workloads. SOPA [24] is a cache framework for *inter-policy* adaptation. It collects a full trace during an evaluation period, replays it into simulators for multiple candidate policies, and adopts the best one. To facilitate policy switches, SOPA maintains a separate LRU-ordered list of all cached blocks. Blocks are replayed in LRU-to-MRU order, helping the new algorithm reconstruct recency metadata, but any frequency or ghost state is lost in translation. Our techniques are complementary, and could reduce SOPA analysis overhead significantly.

SLIDE, inspired by Talus [3], uses MRCs to remove cache performance cliffs. Section 5 presents a detailed comparison, and explains how SLIDE overcomes the challenges of applying Talus to non-LRU policies.

Cliffhanger [5] removes performance cliffs from web memory caches without an explicit miss-ratio curve. Assuming a full MRC is too expensive, limited-size shadow queues instead estimate its gradient, with the sign of the second derivative identifying a cliff. A significant limitation is that Cliffhanger can only scale a single cliff, which must be located within the limited visibility of its shadow queues. Although the authors state that Cliffhanger could work with any eviction policy, their algorithms and experiments are specific to LRU. It is not clear how to apply their shadow-queue technique to more complex caching policies, especially given the challenges identified in Section 5.3. Non-monotonicity may also present problems; even a small local bump in the MRC could be misinterpreted as the single cliff to be removed.

7 Conclusions

We have explored using miniature caches for modeling and optimizing cache performance. Compelling experimental results demonstrate that scaled-down simulation works extremely well for a diverse collection of complex caching algorithms – including ARC, LIRS, 2Q and OPT – across a wide range of real-world traces. This suggests our technique is a robust method capable of modeling nearly any cache policy accurately and efficiently.

Lightweight modeling of non-stack algorithms has many practical applications, including online analysis and control. We presented a general method that runs multiple scaled-down simulations to evaluate hypothetical configurations, and applied it to optimize LIRS and 2Q parameters automatically. We also introduced SLIDE, a new technique that performs Talus-like performance cliff removal transparently for complex policies.

Miniature caches offer the tantalizing possibility of improving performance for most caching algorithms on most workloads automatically. We hope to make additional progress in this direction by exploring opportunities to refine and extend our optimization techniques.

[11] SLIDE is very effective for LRU, and could use SHARDS or AET to construct MRCs more efficiently for a pure LRU policy.

Acknowledgments Thanks to CloudPhysics for helping to make this work possible, and to John Blumenthal, Jeff Hausman, Jim Kleckner, Xiaojun Liu, and Richard Sexton for their encouragement and support. Thanks also to the anonymous reviewers and our shepherd Timothy Wood for their valuable feedback and suggestions.

References

[1] AHO, A. V., DENNING, P. J., AND ULLMAN, J. D. Principles of Optimal Page Replacement. *J. ACM 18*, 1 (Jan. 1971), 80–93.

[2] APPLEBY, A. SMHasher and MurmurHash. `https://code.google.com/p/smhasher/`.

[3] BECKMANN, N., AND SANCHEZ, D. Talus: A Simple Way to Remove Cliffs in Cache Performance. In *Proceedings of the 21st international symposium on High Performance Computer Architecture (HPCA-21)* (February 2015).

[4] BELADY, L. A. A Study of Replacement Algorithms for Virtual Storage Computers. *IBM Systems Journal 5*, 2 (1966), 78–101.

[5] CIDON, A., EISENMAN, A., ALIZADEH, M., AND KATTI, S. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 379–392.

[6] GRYSKI, D. go-arc git repository. `https://github.com/dgryski/go-arc/`.

[7] HU, X., WANG, X., ZHOU, L., LUO, Y., DING, C., AND WANG, Z. Kinetic Modeling of Data Eviction in Cache. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (Berkeley, CA, USA, 2016), USENIX ATC '16, USENIX Association, pp. 351–364.

[8] JIANG, S. LIRS source code. Private communication, Oct 2016.

[9] JIANG, S., AND ZHANG, X. LIRS: An Efficient Low Interreference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2002), SIGMETRICS '02, ACM, pp. 31–42.

[10] JOHNSON, T., AND SHASHA, D. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1994), VLDB '94, Morgan Kaufmann Publishers Inc., pp. 439–450.

[11] KOLLER, R., MASHTIZADEH, A. J., AND RANGASWAMI, R. Centaur: Host-Side SSD Caching for Storage Performance Control. *2015 IEEE International Conference on Autonomic Computing (ICAC)* (2015), 51–60.

[12] KOLLER, R., AND RANGASWAMI, R. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. *Trans. Storage 6*, 3 (Sept. 2010), 13:1–13:26.

[13] MATTSON, R. L., GECSEI, J., SLUTZ, D. R., AND TRAIGER, I. L. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal 9*, 2 (June 1970), 78–117.

[14] MEGIDDO, N., AND MODHA, D. S. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2003), FAST '03, USENIX Association, pp. 115–130.

[15] NARAYANAN, D., DONNELLY, A., AND ROWSTRON, A. Write Off-loading: Practical Power Management for Enterprise Storage. *Trans. Storage 4*, 3 (Nov. 2008), 10:1–10:23.

[16] NIU, Q., DINAN, J., LU, Q., AND SADAYAPPAN, P. PARDA: A Fast Parallel Reuse Distance Analysis Algorithm. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium* (Washington, DC, USA, 2012), IPDPS '12, IEEE Computer Society, pp. 1284–1294.

[17] O'NEIL, E. J., O'NEIL, P. E., AND WEIKUM, G. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1993), SIGMOD '93, ACM, pp. 297–306.

[18] QURESHI, M. K., AND PATT, Y. N. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2006), MICRO 39, IEEE Computer Society, pp. 423–432.

[19] SAEMUNDSSON, T. cache algorithm git repository. `https://github.com/trauzti/cache/`.

[20] SAEMUNDSSON, T., BJORNSSON, H., CHOCKLER, G., AND VIGFUSSON, Y. Dynamic Performance Profiling of Cloud Caches. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, ACM, pp. 28:1–28:14.

[21] SALTZER, J. H., AND KAASHOEK, M. F. *Principles of Computer System Design: An Introduction*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009.

[22] STEFANOVICI, I., THERESKA, E., O'SHEA, G., SCHROEDER, B., BALLANI, H., KARAGIANNIS, T., ROWSTRON, A., AND TALPEY, T. Software-defined Caching: Managing Caches in Multi-tenant Data Centers. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, ACM, pp. 174–181.

[23] WALDSPURGER, C. A., PARK, N., GARTHWAITE, A., AND AHMAD, I. Efficient MRC Construction with SHARDS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2015), FAST '15, USENIX Association, pp. 95–110.

[24] WANG, Y., SHU, J., ZHANG, G., XUE, W., AND ZHENG, W. SOPA: Selecting the Optimal Caching Policy Adaptively. *Trans. Storage 6*, 2 (July 2010), 7:1–7:18.

[25] WIKIPEDIA. LIRS caching algorithm — Wikipedia, the free encyclopedia, 2017. [Online; accessed 22-Jan-2017].

[26] WIRES, J., INGRAM, S., DRUDI, Z., HARVEY, N. J. A., AND WARFIELD, A. Characterizing Storage Workloads with Counter Stacks. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI '14, USENIX Association, pp. 335–349.

[27] ZHAO, W., AND WANG, Z. Dynamic Memory Balancing for Virtual Machines. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2009), VEE '09, ACM, pp. 21–30.

[28] ZHOU, P., PANDEY, V., SUNDARESAN, J., RAGHURAMAN, A., ZHOU, Y., AND KUMAR, S. Dynamic Tracking of Page Miss Ratio Curve for Memory Management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2004), ASPLOS XI, ACM, pp. 177–188.