Miniature Cache Simulations for Modeling and Optimization

CARL A. WALDSPURGER, TRAUSTI SAEMUNDSSON, IRFAN AHMAD, AND NOHHYUN PARK

Carl Waldspurger is an Independent Consultant and Technical Advisor, collaborating on research and development projects with several companies. Carl has a PhD in computer science from MIT and has served as program chair for USENIX ATC, FAST, and VEE. His research interests include resource management, virtualization, caching, computer architecture, and security. carl@waldspurger.org

Trausti Saemundsson is a Software Engineer at Google working on datacenter software. He has an MSc in computer science from Reykjavik University in Iceland. His research interests are systems, analytics, caching, machine learning, security, and optimizations. He has papers published at SoCC, USENIX ATC, and GLBIO. trauzti@gmail.com

Irfan Ahmad is the founder of CachePhysics. Irfan works on interdisciplinary endeavors in memory, storage, CPU, and distributed resource management. He has published at ACM, USENIX, and IEEE. He has served as program chair for HotCloud, HotStorage, and HotEdge. mr.irfan@gmail.com

Nohhyun Park is a Software Engineer at DatosIO working on data protection for distributed databases. He has a PhD in electrical and computer engineering from the University of Minnesota and is interested in workload characterization and performance modeling for large-scale systems. nohhyun.park@datos.io

We present a surprisingly simple technique that accurately models the behavior of a cache with *any* policy by simulating a scaled-down *miniature cache* with a small, spatially hashed sample of requests. We also demonstrate how to leverage such models to optimize caches dynamically, using scaled-down simulations to explore multiple cache configurations simultaneously.

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. A wide variety of caches have been implemented in hardware and software, clients and servers, storage arrays, key-value stores, and other system infrastructure.

By definition, a cache is a small, fast memory backed by larger, slower storage. As a result, cache space is inherently scarce, and methods that can better utilize this space are extremely valuable. Techniques for accurate and efficient cache modeling are especially important for informing cache allocation and partitioning decisions, optimizing cache parameters, and supporting goals including performance, isolation, and quality of service.

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. Although recent research has produced practical models for LRU caches, there has been no general, lightweight solution for more sophisticated policies, such as ARC [7], LIRS [4], and 2Q [5].

Modeling Caches with MRCs

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 and LIRS 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.

Mattson et al. introduced a method for constructing MRCs for *stack algorithms*—for example, LRU, LFU, etc.—that yields the entire MRC for all cache sizes in a single pass over a trace [6]. Efficient modern implementations of this algorithm have an asymptotic cost of $O(N \log M)$ time and $O(M)$ space for a trace of length N containing M unique blocks. Recent approximation techniques can construct accurate MRCs with dramatically lower costs than exact methods. In particular, SHARDS [9] and AET [3] require only $O(N)$ time and $O(1)$ space, with a tiny footprint of approximately 1 MB. 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.

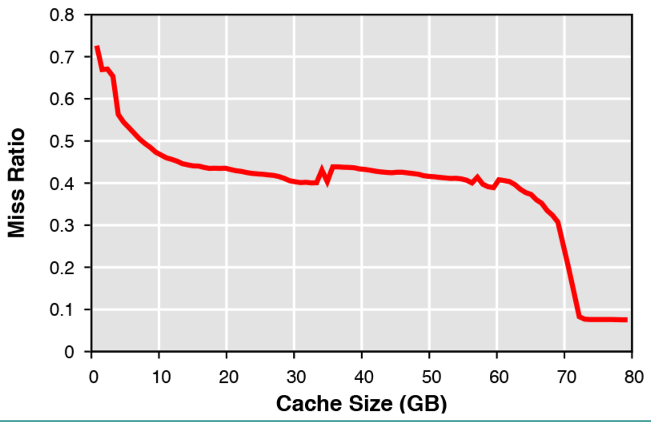

Figure 1: Example MRC. Miss-ratio curve for a production disk block trace using the ARC cache algorithm. The ratio of cache misses to total references is plotted as a function of cache size.

Miniature Simulation

The main idea behind miniature simulation is to approximate the behavior of a large cache by simulating a tiny one that processes only a tiny sample of its requests. Typically, the cache size and the input reference stream are both scaled down by several orders of magnitude.

A mini-simulation runs the full, unmodified cache replacement algorithm, making it possible to model *any* caching algorithm, including even ad hoc modifications often found in production systems. The miss ratio and other metrics are determined by simply extracting the usual statistics from the mini-cache, such as counts of misses and references. An adjustment that instead uses the *expected* number of references reduces bias due to sampling error significantly [8].

The reference stream is scaled down by using hashing to randomly sample the key space. A reference is sampled only when the hash value of its associated key is smaller than a threshold T that defines the sampling rate R. This approach is similar to our earlier work on SHARDS and is also related to sharding in distributed databases. Depending on the cache, a *key* may be a memory address, a logical block number for disk storage, or a string, as in a key-value store. The effectiveness of scaling depends on statistical self-similarity—that a randomized sample is fairly representative of the whole. As we will see, this is a good assumption that holds well in practice.

Figure 2 depicts a full-size cache and its input references, along with two scaled-down versions. To randomly sample the input, simple *temporal* sampling, such as flipping a coin for each reference, doesn't work. We must ensure that all references to the same key are always sampled or we will be blind to reuses that are central to caching behavior. Instead, randomized *spatial* sampling is implemented by selecting references based on deterministic hashes of their keys. In the figure, hash values are represented visually with shading. Scaling down by a factor of two results in a cache with half the size, and an input stream from half the key space, for example, by sampling a key only when the high-order bit of its hash is zero, shown as yielding half of the original shades. Similarly, scaling down by a larger factor of 128 shrinks both the cache size and the key space more dramatically.

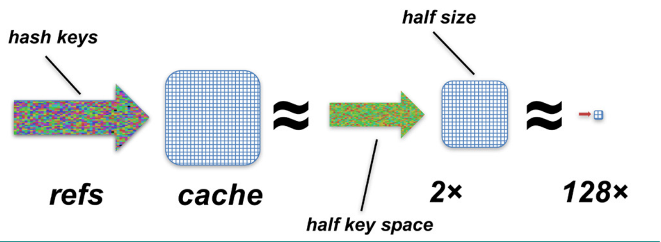

Figure 2: Scaling Down. Both the cache size and input reference stream are scaled down by factors of 2 and 128; each exhibits similar behavior. Only keys that fall within a subset of the hash space are sampled.

Scaling the key space and the cache size by the same amount maintains the same pressure on a mini-cache as the full-size cache, so it should exhibit approximately the same behavior. A cache of size S can be emulated 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. 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.

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 [7], LIRS [4], and the theoretically optimal OPT [2]. We use a collection of 137 real-world storage block trace files, similar to the

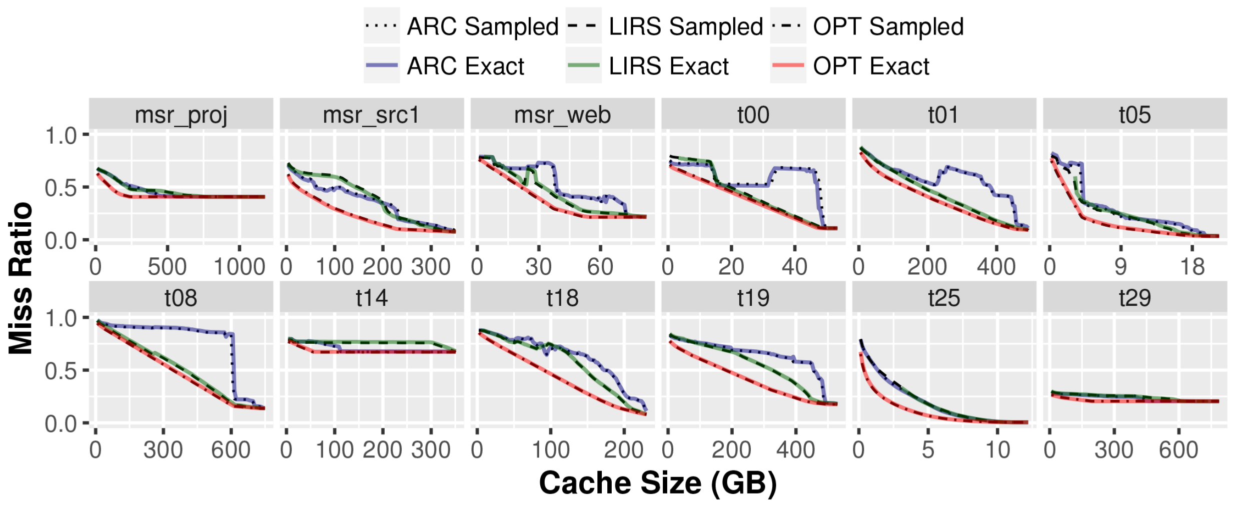

Figure 3: Example Mini-Sim MRCs. Exact and approximate MRCs for 12 representative traces. Approximate MRCs are constructed using scaled-down simulation with sampling rate $R = 0.001$. Each line type represents a different cache algorithm.

SHARDS evaluation [9]. These represent 120 week-long virtual disk traces from production VMware environments collected by CloudPhysics, 12 week-long enterprise server traces collected by Microsoft Research Cambridge, and five day-long server traces collected by FIU. 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.

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.

Figure 3 contains 12 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. 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 capable of modeling nearly any caching algorithm.

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 [9, 3]. The box plots in Figure 4 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.

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. Experiments compare traditional exact simulation with our lightweight scaled-down approach. In all cases, simulations track only metadata, and do not store data blocks.

Resource consumption was measured using our five largest traces. We simulated three cache algorithms at five emulated sizes S_e (8 GB, 16 GB, 32 GB, 64 GB, and 128 GB), using multiple

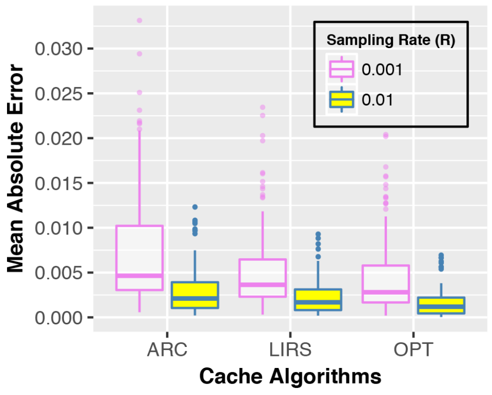

Figure 4: Error Analysis. Distribution of mean absolute error for all 137 traces with three algorithms (ARC, LIRS, OPT) at two different sampling rates (R = 0.01, R = 0.001).

	Linear Function		Example Trace (t22)	
Policy	**Fixed**	**Variable**	**R=.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.

sampling rates R (1, 0.1, 0.01, and 0.001) for a total of 60 experiments per trace.

Unsurprisingly, the memory footprint 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 1000x less space than full simulation, excluding the fixed overhead.

We also measured the CPU usage consumed by our single-threaded cache implementations with both exact and scaled-down simulations for ARC, LIRS, and OPT. The runtime consists of two main components: cache simulation, which is roughly linear in R, and sampling overhead, which is roughly constant; each reference must be hashed to determine if it should be sampled. The scaled-down simulation with $R = 0.001$ requires about 10x 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. Moreover, a separate hash for sampling isn't needed if one is already available; storage caches and key-value stores typically hash keys for performing lookups.

Cache Optimization

A single cache instance runs with a single policy and a single set of configuration parameters. Unfortunately, policy and parameter tweaking is typically performed only at design time, considering few benchmarks.

Low-cost online modeling allows efficient instantiation of multiple concurrent models with different cache configurations, offering a powerful framework for dynamic optimization. Quantifying the impact of hypothetical parameter changes allows the

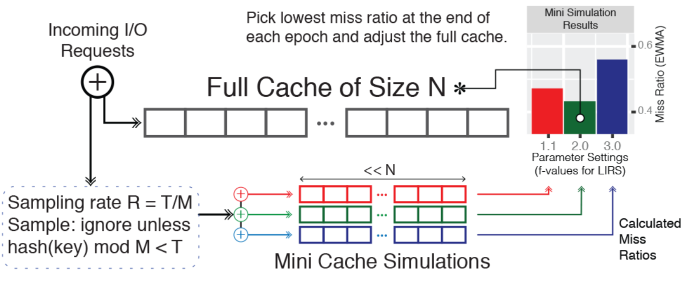

Figure 5: Online Optimization. Simultaneous miniature simulations enable automatic selection of the best parameter setting.

best settings to be applied to the actual cache. Such a multi-model approach can optimize cache block size, write policy, algorithm-specific tunables, or even replacement policy.

Lightweight MRCs can also guide efficient cache sizing, allocation, and partitioning for both individual workloads and complex multi-workload environments. For example, Talus [1], which requires an MRC as input, can remove performance cliffs within a single workload and improve cache partitioning across workloads.

Adapting Cache Parameters

As illustrated in Figure 5, our multi-model optimization framework leverages miniature simulations to evaluate the impact of different candidate parameter values. The best setting is applied to the actual cache periodically. We have implemented optimizations that adapt tunable parameters automatically for two well-known cache policies, LIRS [4] and 2Q [5], but we discuss only the LIRS results; the results for 2Q are similar [8].

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.

LIRS Adaptation

We adapt the size of the LIRS *S* stack, which controls the number of metadata-only ghost entries that are tracked [4], by setting *f*, a parameter that specifies the size of *S* as a fraction of the over-

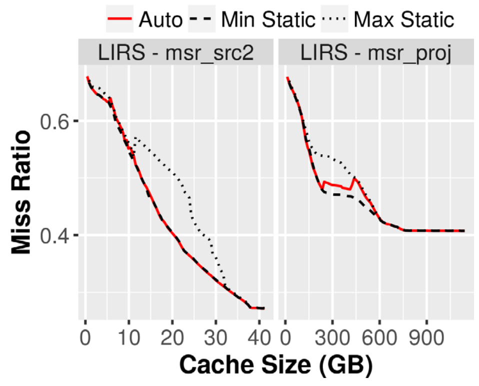

Figure 6: Adaptive Parameter Tuning. Dynamic optimization selects good values for the LIRS *f* parameter at most sizes with potential gains. The `msr_src2` and `msr_proj` workloads show the best- and worst-case results for the MSR traces.

all cache [8]. 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 one million references, the miss ratios for each mini-cache are examined, and the best *f* value is applied to the actual cache.

Figure 6 presents the best-case and worst-case results across the 12 MSR traces. The goal of automatic LIRS adaptation is to find the best value of *f* for each cache size. These ideal static settings form an MRC that traces the lower envelope of the curves for different static *f* values, plotted as the dashed curve. Similarly, the dotted curve shows the MRC with the pessimal static *f* setting at each cache size. The auto-adapted results for `msr_src2` hug the ideal lower envelope closely at nearly all cache sizes. In contrast, `msr_proj` deviates from the ideal for many cache sizes, but still does well for others. We are currently experimenting with techniques that automatically disable adaptation when it is ineffective.

SLIDE

SLIDE (*S*harded *L*ist with *I*nternal *D*ifferential *E*viction) is a completely different cache optimization technique that leverages scaled-down MRCs constructed by running miniature simulations. SLIDE was inspired by Talus [1], a powerful technique introduced in the computer architecture community for set-associative processor caches. Talus removes performance cliffs using interpolation to effectively operate at a point on the convex hull of an MRC—the shape formed by stretching a rubber band across the bottom of the curve. In the presence of cliffs, the large gap between an MRC and its convex hull represents a significant optimization opportunity.

Talus uses hash-based partitioning to divide the reference stream for a single workload into two shadow partitions, *alpha* and *beta*, which operate as separate sub-caches. Each partition is made to emulate the performance of a smaller or larger cache by controlling its size and its input load, represented by the fraction of the reference stream it receives. Talus requires the workload's MRC as an input and computes the partition sizes and their respective loads in a clever manner that ensures their combined aggregate miss ratio lies on the convex hull of the MRC. We view the hash-based partitioning employed by Talus for optimization and our hash-based monitoring for efficient modeling 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.

One key challenge with applying Talus to non-stack algorithms is constructing MRCs efficiently at runtime. Fortunately, scaled-down models provide a convenient solution. As with parameter adaptation, 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 that forms the convex hull for the MRC, and compute the optimal partition sizes and loads using the same inexpensive method as Talus.

Non-LRU Shadow Partitioning Challenges

In theory, combining scaled-down MRCs with Talus shadow partitioning can improve the performance of *any* caching policy by interpolating efficient operating points on the convex hulls of workload MRCs. In practice, it was much more difficult than we expected to apply Talus to caching algorithms such as ARC and LIRS.

Talus requires distinct cache instances for its alpha and beta partitions, which have a fixed aggregate size. This hard division becomes problematic in systems where partition boundaries change dynamically as MRCs evolve over time. Similarly, when per-partition input loads change dynamically, some cache entries may reside in the "wrong" partition based on their hash values.

Eager strategies, such as removing cache entries when decreasing the size of a partition or migrating entries across partitions to ensure each resides in the correct partition, perform poorly since migration is 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, since list positions are not ordered across partitions.

Lazy strategies for reallocation and migration fare better but complicate the core caching logic. More importantly, while migrating to the MRU position on a hit seems reasonable for an

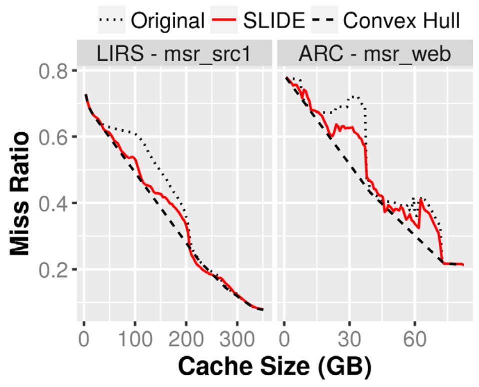

Figure 7: SLIDE Cliff Reduction. Scaled-down MRCs are constructed dynamically from seven mini-cache simulations. SLIDE improves miss ratios for LIRS and ARC at most sizes with potential gains but does exhibit some regressions.

LRU policy, it's not clear how to merge state appropriately for more general algorithms.

Transparent Shadow Partitioning

Faced with these challenges, we developed SLIDE. In contrast to Talus, SLIDE maintains a single unified cache and defers partitioning decisions until eviction time, conveniently avoiding 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. Since SLIDE interposes on primitive LRU operations that add, reference, and evict entries, it is transparent to cache replacement decisions. An unmodified algorithm can support Talus-like partitioning by simply relinking to substitute SLIDE lists for existing ones. We have optimized ARC (*T1*, *T2*, *B1*, and *B2*), LIRS (*S* and *Q*), 2Q (*Am*, *A1in*, and *A1out*), and LRU in this manner.

SLIDE extends a conventional doubly linked LRU list, which remains totally ordered from MRU (head) to LRU (tail). Each entry is augmented with a compact hash of its key, which is compared to a current threshold that dynamically classifies it as belonging to either the alpha or beta "partition." Additional state supports efficient SLIDE versions of all list operations [8]. SLIDE preferentially evicts from the tail of the over-quota partition.

It is not obvious that substituting SLIDE lists for internal lists will approximate Talus partitions. The basic intuition is that configuring each internal list with identical SLIDE partition sizes and input loads effectively divides the occupancy of each individual list—and therefore the entire aggregate algorithm state—to achieve the desired split between alpha and beta. While SLIDE may differ from strict Talus partitioning, it empirically works well for ARC, LIRS, 2Q, and LRU.

Experiments

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 one million-reference epoch, using an EWMA weight of 0.2.

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), representing less than 0.04% of total memory consumption for an actual cache. Alternative configurations provide different tradeoffs between time, space, and accuracy.

Figure 7 plots some example results of SLIDE performance cliff reduction for LIRS and ARC policies with workloads that exhibit cliffs. 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. Nevertheless, for these examples, SLIDE captures a significant fraction of the potential gain, represented by the area between the MRC and its convex hull: 69% for LIRS and 38% for ARC. For workloads with MRCs that are already mostly convex, there is little opportunity for improvement, so SLIDE typically yields marginal benefits.

Conclusion

We have explored the use of 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, 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 has many applications, including online analysis and control. We presented a general method that runs scaled-down simulations to evaluate hypothetical configurations, and applied it to optimize tunable cache policy parameters automatically. We also introduced SLIDE, a new transparent technique that performs Talus-like performance cliff removal.

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.

References

[1] N. Beckmann and D. Sanchez, "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): https://people.csail.mit.edu/sanchez/papers/2015.talus.hpca.pdf.

[2] L. A. Belady, "A Study of Replacement Algorithms for Virtual Storage Computers," *IBM Systems Journal,* vol. 5, no. 2 (1966), pp. 78–101: http://users.informatik.uni-halle.de/~hinnebur/Lehre/Web_DBIIb/uebung3_belady_opt_buffer.pdf.

[3] X. Hu, X. Wang, L. Zhou, Y. Luo, C. Ding, and Z. Wang, "Kinetic Modeling of Data Eviction in Cache," in *Proceedings of the 2016 USENIX Annual Technical Conference (ATC '16)*, pp. 351–364: https://www.usenix.org/conference/atc16/technical-sessions/presentation/hu.

[4] S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-Reference 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 (SIGMETRICS '02)*, pp. 31–42: http://web.cse.ohio-state.edu/hpcs/WWW/HTML/publications/papers/TR-02-6.pdf.

[5] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," in *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*, pp. 439–450: http://www.vldb.org/conf/1994/P439.PDF.

[6] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal*, vol. 9, no. 2 (June 1970), pp. 78–117.

[7] N. Megiddo and D. S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, pp. 115–130: http://www2.cs.uh.edu/~paris/7360/PAPERS03/arcfast.pdf.

[8] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park, "Cache Modeling and Optimization Using Miniature Simulations," in *2017 USENIX Annual Technical Conference (ATC '17)*, pp. 487–498: https://www.usenix.org/system/files/conference/atc17/atc17-waldspurger.pdf.

[9] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad, "Efficient MRC Construction with SHARDS," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, pp. 95–110: https://www.usenix.org/system/files/conference/fast15/fast15-paper-waldspurger.pdf.

Register Now!

nsdi '18 **15th USENIX Symposium on Networked Systems Design and Implementation**

April 9–11, 2018 • Renton, WA, USA

NSDI focuses on the design principles, implementation, and practical evaluation of networked and distributed systems. The symposium provides a high-quality, single-track forum for presenting results and discussing ideas that further the knowledge and understanding of the networked systems community as a whole, continue a significant research dialog, or push the architectural boundaries of network services.

The program includes three days of technical sessions, a poster session, and evening Birds-of-a-Feather sessions (BoFs).

Register by March 19 and save!

www.usenix.org/nsdi18

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Save the Date!

OSDI 18 **13th USENIX Symposium on Operating Systems Design and Implementation**

October 8–10, 2018 • Carlsbad, CA, USA

OSDI brings together professionals from academic and industrial backgrounds in what has become a premier forum for discussing the design, implementation, and implications of systems software. The OSDI Symposium emphasizes innovative research as well as quantified or insightful experiences in systems design and implementation.

Abstract registrations are due April 26, 2018.

Program Co-Chairs:
Andrea Arpaci-Dusseau, *University of Wisconsin—Madison*
and Geoff Voelker, *University of California, San Diego*

www.usenix.org/osdi18

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION