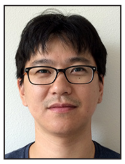
Beyond Working Sets
Online MRC Construction with SHARDS

CARL A. WALDSPURGER, NOHHYUN PARK, ALEXANDER GARTHWAITE, AND IRFAN AHMAD

Carl Waldspurger has been conducting research at CloudPhysics since its inception. Carl has a PhD in computer science from MIT. His research interests include resource management, virtualization, security, data analytics, and computer architecture.
carl@cloudphysics.com



Nohhyun Park is a Software Engineer at CloudPhysics working on data analytics and the supporting pipeline. 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@cloudphysics.com

Alexander Garthwaite is a Software Engineer at Twitter and an advisor to CloudPhysics. Alex has a PhD in computer and information science from the University of Pennsylvania. His interests include resource management, virtualization, programming language implementation, and computer architecture.
alex@cloudphysics.com

Irfan Ahmad is the CTO and cofounder of CloudPhysics. Irfan works on interdisciplinary endeavors in memory, storage, CPU, and distributed resource management. He has published at ACM SOCC (best paper), USENIX ATC, FAST, IISWC, etc. He has chaired HotCloud, HotStorage, and VMware's Innovation Conference.
irfan@cloudphysics.com

Estimating the performance impact of caching on storage workloads is an important and challenging problem. Miss ratio curves (MRCs) provide valuable information about cache utility, enabling efficient cache sizing and dynamic allocation decisions. Unfortunately, computing exact MRCs is too expensive for practical online applications. We introduce a novel approximation algorithm called SHARDS that leverages uniform randomized spatial sampling to construct surprisingly accurate MRCs using only modest computational resources. Operating in constant space and linear time, SHARDS makes online MRC generation practical for even the most constrained computing environments.

Caches designed to accelerate data access by exploiting locality are pervasive in modern systems. Operating systems and databases maintain in-memory buffer caches containing "hot" blocks considered likely to be reused. Server-side or networked storage caches using flash memory are popular as a cost-effective way to reduce application latency and offload work from rotating disks. Virtually all storage devices—ranging from individual disk drives to large storage arrays—include significant caches composed of RAM or flash memory.

Since cache space consists of relatively fast, expensive storage, it is inherently a scarce resource and is commonly shared among multiple clients. As a result, optimizing cache allocations is important. Today, administrators or automated systems seeking to optimize cache allocations are forced to resort to simple heuristics, or to engage in trial-and-error tests. Both approaches to performance estimation are problematic.

Heuristics simply don't work well for cache sizing, since they cannot capture the temporal locality profile of a workload. Without knowledge of marginal benefits, for example, doubling (or halving) the cache size for a given workload may change its performance only slightly, or by a dramatic amount.

Trial-and-error tests that vary the size of a cache and measure the effect are not only time-consuming and expensive, but also present significant risk to production systems. Correct sizing requires experimentation across a range of cache allocations; some might induce thrashing and cause a precipitous loss of performance. Long-running experiments required to warm up caches or to observe business cycles may exacerbate the negative effects. In practice, administrators rarely have time for this. Resigned to severe imbalances in cache utility, they often end up buying additional hardware.

The ideal approach is estimating workload performance as a function of cache size by modeling its inherent temporal locality; in other words, by incorporating information about the reuse of blocks. As the workload accesses each individual block, its *reuse distance*—the number of other unique intervening blocks referenced since its previous use—is captured and accumulated in a histogram. The complete *miss ratio curve* (MRC) for a workload is

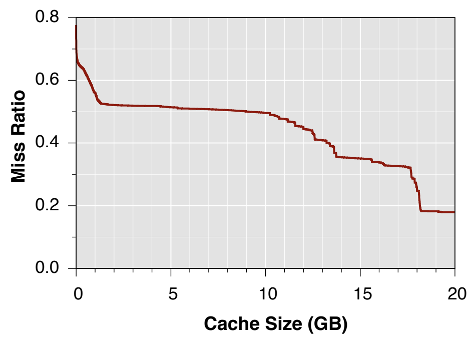


Figure 1: Example MRC. A miss ratio curve plots the ratio of cache misses to total references, as a function of cache size. Lower is better.

computed directly from its reuse-distance histogram. Unfortunately, even the most efficient exact implementations for MRC construction are too heavyweight for practical online use in production systems.

Figure 1 shows an example 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 higher the miss ratio, the worse the performance; the miss ratio decreases as cache size increases. MRCs come in many shapes and sizes, and represent the historical cache behavior of a particular workload. This particular MRC reveals a staircase pattern representing knees in the working set: the first 2 GB of cache provide a large improvement, followed by a flat region for the next 8 GB, then another dropoff, and so on. Cache performance is highly nonlinear, so identifying such knees is critical for making efficient allocation and partitioning decisions.

Assuming some level of stationarity in the workload pattern at the time scale of interest, the workload's MRC can be used to predict its future cache performance. An administrator can use a system-wide miss ratio curve to help determine the aggregate amount of cache space to provision for a desired improvement in overall system performance. Similarly, an automated cache manager can utilize separate MRCs for multiple workloads of varying importance, optimizing cache allocations dynamically to achieve service-level objectives.

MRC Construction

In their seminal paper, Mattson, Gecsei, Slutz, and Traiger [1] proposed a technique to generate models of behavior for all cache sizes in a *single pass*. Since then, Mattson's technique has been applied widely. However, the computation and space required to generate such MRCs have been prohibitive. For a trace of length N containing M unique references, the most efficient exact implementations of this algorithm have an asymptotic cost of $O(N \log M)$ time and $O(M)$ space [4].

Given the nonlinear computation cost and unbounded memory requirements, it is impractical to perform real-time analysis in production systems. Even when processing can be delayed and performed offline from a trace file, memory requirements may still be excessive. For example, we have collected many traces for which conventional MRC construction does not fit in 64 GB RAM. This is especially important when modeling large storage caches; in contrast to RAM-based caches, affordable flash cache capacities often exceed 1 TB, requiring many gigabytes of RAM for traditional MRC construction.

The limitations of existing MRC algorithms led us to consider a very simple idea. What if we place a filter in front of a conventional MRC algorithm to randomly sample only a small subset of its input blocks, and run the full algorithm over these samples? The question was whether or not this would be sufficiently efficient and accurate for practical use.

Our answer to this question is a new algorithm based on spatially hashed sampling called *SHARDS* (*S*patially *H*ashed *A*pproximate *R*euse *D*istance *S*ampling) [7]. SHARDS runs in constant space and linear time by tracking only references to representative locations, selected dynamically based on a function of their hash values.

Randomized spatial sampling allows SHARDS to use several orders of magnitude less space and time than exact methods, making it inexpensive enough for practical online MRC construction in high-performance systems. The dramatic space reductions also enable analysis of long traces that is not feasible with exact methods. Traces that consume many gigabytes of RAM to construct exact MRCs require less than 1 MB for accurate approximations. The low cost even enables concurrent evaluation of different cache configurations (e.g., block size or write policy) using multiple SHARDS instances.

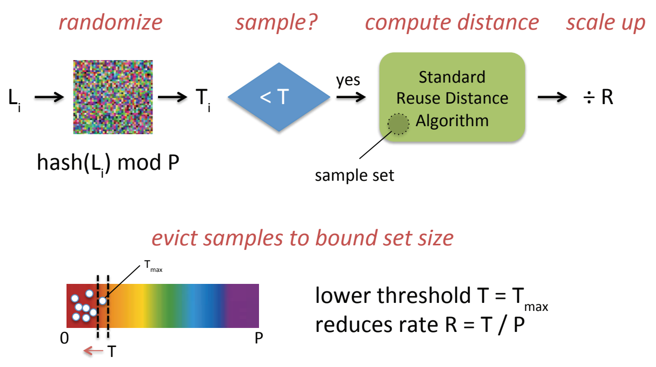


Figure 2: SHARDS algorithm overview. SHARDS filters the input to a standard reuse-distance algorithm using spatially hashed sampling. Each input location L_i is mapped to a hash value T_i, which is compared to a global threshold T that determines the sampling rate R. The threshold is lowered progressively as needed to maintain a fixed bound on the size of the sample set, s_{max}.

SHARDS Algorithm

The SHARDS algorithm, shown in Figure 2, is conceptually simple. A hash function takes each referenced location L_i, such as a logical block number (LBN), and maps it to a hash value T_i, that is uniformly distributed over the range [0, P), depicted as painting each location with a random color.

A global threshold T is used to divide the hash value space into two partitions, or "shards." Locations that hash to values below the threshold are sampled, and others are filtered out. The sampling rate R is simply the fraction of the hash value space that is sampled. In practice, typical sampling rates are significantly lower than 1%. More generally, using the sampling condition *hash(L)* mod $P < T$, with modulus P and threshold T, the effective sampling rate is $R = T/P$, and each sample represents $1/R$ locations, in a statistical sense. In practice, each sample typically represents hundreds or thousands of locations.

For the basic SHARDS algorithm, we simply take this spatial sampling filter, and place it in front of a standard reuse-distance algorithm, effectively scaling down its inputs by a factor of R. We then take the reuse distances output by the algorithm, and scale them back up, to reflect the sampling rate R.

This method has several desirable properties. As required for reuse distance computations, it ensures that all accesses to the same location will be sampled, since they will have the same hash value. It does not require any prior knowledge about the system, its workload, or the location address space. In particular, no information is needed about the set of locations that may be accessed by the workload, nor the distribution of accesses to these locations. As a result, SHARDS sampling is effectively stateless. In contrast, explicitly preselecting a random subset of locations may require significant storage, especially if the location address space is large. Often, only a small fraction of this space is accessed by the workload, making such preselection especially inefficient.

Although this basic approach can reduce the time and space required to generate an MRC by several orders of magnitude, it can still be improved. First, the required space grows slowly, but isn't bounded, making it hard to use in memory-constrained environments. Second, choosing an appropriate sampling rate can be challenging, since it implies an accuracy versus overhead tradeoff that can be difficult to evaluate, especially in an online system.

To address these issues, we developed a fixed-size version of SHARDS that operates in constant space. The basic idea is that instead of specifying the sampling rate R, we specify a maximum number of samples to track, s_{max}. Placing a hard bound on the sample set results in a constant-space algorithm. The basic spatial filtering step operates exactly the same as before. But now, if adding a new sample would exceed the space bound s_{max}, some existing sample must be evicted to make room.

We remove the sample with the maximum hash value, T_{max}, closest to T. The global threshold T is then lowered to T_{max} *since* any larger values cannot fit in the set, reducing the sampling rate R dynamically. When the threshold is lowered, a *subset-inclusion property* is maintained automatically; each location sampled *after* lowering the rate would also have been sampled *prior* to lowering the rate.

The subset-inclusion property is leveraged to lower the sampling rate adaptively as more unique locations are encountered, in order to maintain a fixed bound on the total number of samples that are tracked at any given point in time. The sampling rate is initialized to a high value; in practice $R_0 = 0.1$ is sufficiently high to achieve good results with nearly any workload.

As the rate is reduced, the counts associated with earlier updates to the reuse-distance histogram need to be adjusted. Ideally, the effects of all updates associated with an evicted sample should be rescaled exactly. Since this would incur significant space and processing costs, we opt for a simple approximation.

When the threshold is reduced, the count associated with each histogram bucket is scaled by the ratio of the new and old sampling rates, R_{new} / R_{old}, which is equivalent to the ratio of the new and old thresholds, T_{new} / T_{old}. Rescaling makes the simplifying assumption that previous references to an evicted sample contributed equally to all existing buckets—a reasonable statistical approximation when viewed over many sample evictions and rescaling operations. Rescaling is performed incrementally and inexpensively, and ensures that subsequent references to the remaining samples have the appropriate relative weight associated with their corresponding histogram bucket increments.

Evaluating SHARDS

With a constant memory footprint, SHARDS is suitable for online use in memory-constrained systems, such as device drivers in embedded systems. To explore such applications, we developed a high-performance implementation, written in C, and optimized for space efficiency. With our default setting of $s_{max} = 8\text{K}$, the entire measured runtime footprint—including code size, stack space, and all other memory usage—is smaller than 1 MB, making this implementation practical even for extremely memory-constrained execution environments.

We have deployed SHARDS in the context of the commercial CloudPhysics I/O caching analytics service for virtualized environments. Our system streams compressed block I/O traces for VMware virtual disks from customer datacenters to a cloud-based backend that constructs approximate MRCs efficiently. A Web-based interface reports expected cache benefits, such as the cache size required to reduce average I/O latency by specified amounts. Running this service, we have accumulated a large number of production traces from customer environments.

We analyzed 106 week-long traces, collected from virtual disks in production customer environments with sizes ranging from 8 GB to 34 TB, with a median of 90 GB. The associated virtual machines were a mix of Windows and Linux, with up to 64 GB RAM (6 GB median) and up to 32 virtual CPUs (2 vCPUs median). In addition, we used 18 publicly available block I/O traces from the SNIA IOTTA repository [6], including a dozen week-long enterprise server traces collected by Microsoft Research Cambridge [3].

In total, we analyzed a diverse set of 124 real-world block I/O traces to evaluate the accuracy and performance of SHARDS compared to exact methods. For each experiment, we modeled a simple LRU cache replacement policy, with a 16 KB cache block size—typical for storage cache configurations in commercial virtualized systems.

To quantify the accuracy of SHARDS, we considered the difference between each approximate MRC, constructed using spatially hashed sampling, and its corresponding exact MRC, generated from a complete reference trace. An intuitive measure of this distance, also used to quantify error in related work, is the mean absolute difference or error (MAE) between the approximate and exact MRCs across several different cache sizes. This difference is between two values in the range [0, 1], so an absolute error of 0.01 represents 1% of that range.

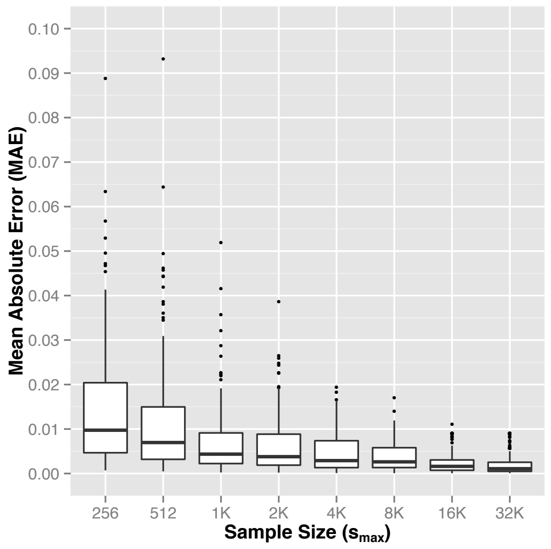


Figure 3: Error analysis. Mean absolute error calculated over all 124 traces for different SHARDS sample set sizes. The top and bottom of each box represents the first and third quartile values of the error; the thick black line is the median. The thin whiskers represent the min and max error, excluding outliers, which are represented by dots.

The box plots in Figure 3 show the MAE metric for a wide range of SHARDS sample set sizes (s_{max}). For each trace, this distance is computed over all discrete cache sizes, at 64 MB granularity, corresponding to all non-zero histogram buckets. Overall, the average error is extremely low. For s_{max} = 8K, the median MAE is 0.0027, with a worst case of 0.017. The error for tiny sample sizes is also surprisingly small. For example, with only 256 samples, the error for 75% of the traces is below 0.02, although there are many outliers.

Many statistical methods exhibit sampling error inversely proportional to $\sqrt{n}$, where n is the sample size. Our data is consistent; regressing the average absolute error for each s_{max} value shown in Figure 3 against $1/\sqrt{s_{max}}$ resulted in a high correlation coefficient of $r^2 = 0.98$. This explains the observed diminishing accuracy improvements with increasing s_{max}.

Why does SHARDS work so well, even with small sample sizes and correspondingly low sampling rates? Our intuition is that most workloads are composed of a fairly small number of basic underlying processes, each of which operates somewhat uniformly over relatively large amounts of data. As a result, a small number of representative samples is sufficient to model the main underlying processes. Additional samples are needed to properly capture the relative weights of these processes. Interestingly, the number of samples required to obtain accurate results for a given

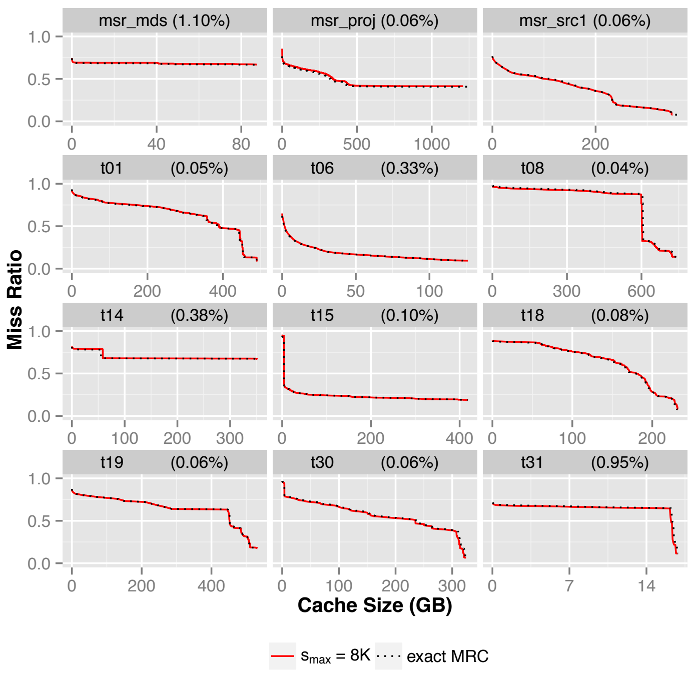


Figure 4: Example MRCs: exact vs. SHARDS. Exact and approximate MRCs for 12 representative traces. Approximate MRCs are constructed using SHARDS with s_{max} = 8K. Trace names are shown for three public MSR traces [3]; others are anonymized. The effective sampling rates appear in parentheses.

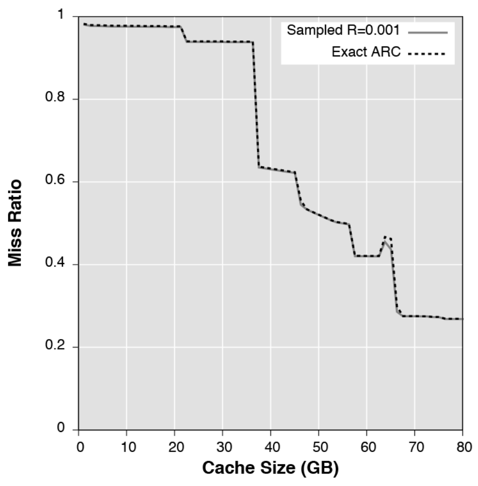


Figure 5: Scaled-down ARC simulation. Exact and approximate MRCs for the MSR-web disk trace [3]. Each curve plots 64 separate ARC simulations at different cache sizes.

workload may be indicative of its underlying dimensionality or intrinsic complexity.

Figure 4 provides further qualitative evidence of SHARDS accuracy for a dozen representative traces. In most cases, the approximate and exact MRCs are nearly indistinguishable. Each plot is annotated with the effective dynamic sampling rate, indicating the fraction of I/Os processed, including evicted samples. This rate reflects the amount of processing required to construct the MRC.

Overall, quantitative experiments confirm that, for all workloads, SHARDS yields accurate MRCs, in radically less time and space than conventional exact algorithms. Compared to the sequential implementation of PARDA [4], a modern high-performance reuse-distance algorithm, SHARDS requires dramatically less memory and processing resources. For our trace set, we measured memory reductions by a factor of up to 10,800χ for large traces, and a median of 185χ across all traces. The computation cost was also reduced up to 204χ for large traces, with a median of 22χ. For large traces, SHARDS throughput exceeds 17 million references per second.

Renewed Interest in MRCs

Recently, there has been renewed interest in algorithms for efficient MRC construction, using a variety of different techniques, which has been very exciting to see. For example, Saemundsson et al. [5] grouped references into variable-sized buckets. Their ROUNDER aging algorithm with 128 buckets yields MAEs up to 0.04 with a median MAE of 0.006 for partial MRCs, but the space complexity remains *O(M)*.

Wires et al. recently created an alternate way of computing MRCs using a *counter stack* [8]. In the closest matching test case using the same large trace and an identical cache configuration, Counter Stacks is more than 7χ slower and needs 62χ as much memory as SHARDS with s_{max} = 8K. In this case, Counter Stacks is more accurate, with an MAE of only 0.0025, compared to 0.0061 for SHARDS. Using s_{max} = 32K, with a 2 MB memory footprint, SHARDS yields a comparable MAE of 0.0026, still approximately 7χ faster, with a 40χ smaller footprint. While Counter Stacks uses $O(\log M)$ space, SHARDS computes MRCs in small *constant* space. As a result, it is practical to use separate, potentially concurrent SHARDS instances to efficiently compute multiple MRCs tracking different properties or time-scales for a given reference stream.

Scaled-Down Simulation

Like other algorithms based on Mattson's single-pass method [1], SHARDS constructs MRCs for caches that use a stack-algorithm replacement policy, such as LRU. Significantly, the same underlying spatial sampling approach can be used to simulate more sophisticated policies, such as ARC [2], for which there are no known single-pass methods to speed up analysis.

Our approach is to simulate each cache size separately, while scaling down the simulations to regain efficiency. As with basic SHARDS, input references are filtered using a hash-based sampling condition, corresponding to the sampling rate *R*. A series of separate simulations is run, each using a different cache size, which is also scaled down by *R*. Figure 5 presents both exact and scaled-down sampled MRCs for the public MSR web block trace [3], for 64 simulated ARC cache sizes. With $R = 0.001$, the simulated cache is only 0.1% of the desired cache size, achieving huge reductions in space and time, while exhibiting excellent accuracy, with an MAE of 0.002.

Encouraged by our results from generalizing hash-based spatial sampling to model sophisticated cache replacement policies, we are exploring similar techniques for other complex systems. We are also examining the rich temporal dynamics of MRCs at different time scales.

References

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

[2] 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),* Berkeley, CA, 2003, USENIX Association, pp. 115–130.

[3] D. Narayanan, A. Donnelly, and A. Rowstron, "Write Off-Loading: Practical Power Management for Enterprise Storage," *Trans. Storage,* vol. 4, no. 3 (Nov. 2008), 10:1–10:23.

[4] Q. Niu, J. Dinan, Q. Lu, and P. Sadayappan, "PARDA: A Fast Parallel Reuse Distance Analysis Algorithm," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS '12),* Washington, DC, 2012, IEEE Computer Society, pp. 1284–1294.

[5] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson, "Dynamic Performance Profiling of Cloud Caches," in *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14),* New York, NY, 2014, ACM, pp. 28:1–28:14.

[6] SNIA. SNIA IOTTA Repository Block I/O Traces: http://iotta.snia.org/tracetypes/3.

[7] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad, "Efficient MRC Construction with SHARDS," in *13th USENIX Conference on File and Storage Technologies (FAST '15),* Santa Clara, CA, 2015, USENIX Association, pp. 95–110.

[8] J. Wires, S. Ingram, Z. Drudi, N. J. A. Harvey, and A. Warfield, "Characterizing Storage Workloads with Counter Stacks," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14),* Berkeley, CA, 2014, USENIX Association, pp. 335–349.