Efficient MRC Construction with SHARDS

Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad, *CloudPhysics, Inc.*

https://www.usenix.org/conference/fast15/technical-sessions/presentation/waldspurger

This paper is included in the Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15).

February 16–19, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-201

Open access to the Proceedings of the 13th USENIX Conference on File and Storage Technologies is sponsored by USENIX

Efficient MRC Construction with SHARDS

Carl A. Waldspurger Nohhyun Park Alexander Garthwaite Irfan Ahmad
CloudPhysics, Inc.

Abstract

Reuse-distance analysis is a powerful technique for characterizing temporal locality of workloads, often visualized with miss ratio curves (MRCs). Unfortunately, even the most efficient exact implementations are too heavyweight for practical online use in production systems.

We introduce a new approximation algorithm that employs uniform randomized spatial sampling, implemented by tracking references to representative locations selected dynamically based on their hash values. A further refinement runs in constant space by lowering the sampling rate adaptively. Our approach, called *SHARDS* (*S*patially *H*ashed *A*pproximate *R*euse *D*istance *S*ampling), drastically reduces the space and time requirements of reuse-distance analysis, making continuous, online MRC generation practical to embed into production firmware or system software. SHARDS also enables the analysis of long traces that, due to memory constraints, were resistant to such analysis in the past.

We evaluate SHARDS using trace data collected from a commercial I/O caching analytics service. MRCs generated for more than a hundred traces demonstrate high accuracy with very low resource usage. MRCs constructed in a bounded 1 MB footprint, with effective sampling rates significantly lower than 1%, exhibit approximate miss ratio errors averaging less than 0.01. For large traces, this configuration reduces memory usage by a factor of up to 10,800 and run time by a factor of up to 204.

1 Introduction

Caches designed to accelerate data access by exploiting locality are pervasive in modern storage 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, and approaches for estimating workload performance as a function of cache size are particularly valuable.

1.1 Cache Utility Curves

Cache utility curves are effective tools for managing cache allocations. Such 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 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.

Assuming some level of stationarity in the workload pattern at the time scale of interest, its MRC can also 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.

1.2 Weaker Alternatives

The concept of a *working set* — the set of data accessed during the most recent sample interval [16] — is often used by online allocation algorithms in systems software [12, 54, 61]. While working-set estimation provides valuable information, it doesn't measure data reuse, nor does it predict changes in performance as cache allocations are varied. Without the type of information conveyed in a cache utility curve, 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 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 ex-

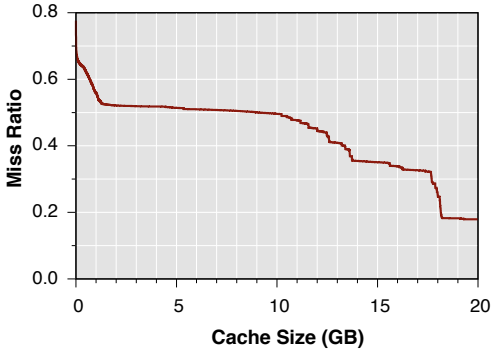

Figure 1: **Example MRC.** A miss ratio curve plots the ratio of cache misses to total references, as a function of cache size.

pensive, 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.

1.3 MRC Construction

Although cache utility curves are extremely useful for planning and optimization, the algorithms used to construct them are computationally expensive. To construct an exact MRC, it is necessary to observe data reuse over the access trace. *Every* accessed location must be tracked and stored in data structures during trace processing, resulting in large overheads in both time and space.

The technique due to Mattson *et al.* [34] scans the trace of references to collect a histogram of reuse distances. The *reuse distance* for an access to a block B is measured as the number of other intervening unique blocks referenced since the previous access to B. The number of times a particular reuse distance occurs is collected while processing the trace, over all possible reuse distances. Conceptually, for modeling LRU, accessed blocks are totally ordered in a stack from most recent to least recent access. On an access to block B, it:

- determines the reuse distance of B as:
  $D =$ stack depth of B (for first access to B, $D = \infty$),
- records D in a reuse-distance histogram, and
- moves B to the top of stack.

Standard implementations maintain a balanced tree to track the most recent references to each block and compute reuse distances efficiently, and employ a hash table for fast lookups into this tree. For a trace of length N containing M unique references, the most efficient implementations of this algorithm have an asymptotic cost of $O(N \log M)$ time and $O(M)$ space.

Given the non-linear 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.[1] 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.

1.4 Our Contributions

We introduce a new approach for reuse-distance analysis that constructs accurate miss ratio curves using only modest computational resources. We call our technique *SHARDS*, for *S*patially *H*ashed *A*pproximate *R*euse *D*istance *S*ampling. It employs randomized spatial sampling, implemented by tracking only references to representative locations, selected dynamically based on a function of their hash values. We further introduce an extended version of SHARDS which runs in *constant* space, by lowering the sampling rate adaptively.

The SHARDS approximation requires several orders of magnitude less space and time than exact methods, and is 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.

This low cost even enables concurrent evaluation of different cache configurations (*e.g.*, block size or write policy) using multiple SHARDS instances. We also present a related generalization to non-LRU policies.

We have implemented SHARDS in the context of a commercial I/O caching analytics service for virtualized environments. Our system streams compressed block I/O traces for VMware virtual disks from customer data centers 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.

For this paper, we analyzed both exact and approximate MRCs for more than a hundred virtual disks from our trace library, plus additional publicly-available block I/O traces. Averaged across all traces, the miss ratios of the approximated MRCs, constructed using a 0.1% sampling rate, deviate in absolute value from the exact MRCs by an average of less than 0.02; *i.e.*, the approximate sampled miss ratio is within 2 percentage points of the value calculated exactly using the full trace.

Moreover, approximate MRCs constructed using a fixed sample-set size, with only 8K samples in less than

[1] We have collected several single-VM I/O traces for which conventional MRC construction does not fit in 64 GB RAM.

1 MB memory, deviate by an average of less than 0.01 from the exact full trace values. This high accuracy is achieved despite dramatic memory savings by a factor of up to $10,800\times$ for large traces, and a median of $185\times$ across all traces. The computation cost is also reduced up to $204\times$ for large traces, with a median of $22\times$.

The next section presents the SHARDS algorithm, along with an extended version that runs in constant space. Details of our MRC construction implementation are examined in Section 3. Section 4 evaluates SHARDS through quantitative experiments on more than a hundred real-world I/O traces. Related work is discussed in Section 5. Finally, we summarize our conclusions and highlight opportunities for future work in Section 6.

2 SHARDS Sampling Algorithm

Our core idea is centered around a simple question: what if we compute reuse distances for a randomly sampled subset of the referenced blocks? The answer leads to SHARDS, a new algorithm based on spatially-hashed sampling. Despite the focus on storage MRCs, this approach can be applied more generally to approximate other cache utility curves, with any stream of references containing virtual or physical location identifiers.

2.1 Basic SHARDS

SHARDS is conceptually simple — for each referenced location L, the decision of whether or not to sample L is based on whether $hash(L)$ satisfies some condition. For example, the condition $hash(L) \bmod 100 < K$ samples approximately K percent of the entire location space. Assuming a reasonable hash function, this effectively implements uniform random spatial sampling.

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 pre-selecting 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 pre-selection especially inefficient.

More generally, using the sampling condition $hash(L) \bmod P < T$, 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. The sampling rate may be varied by changing the threshold

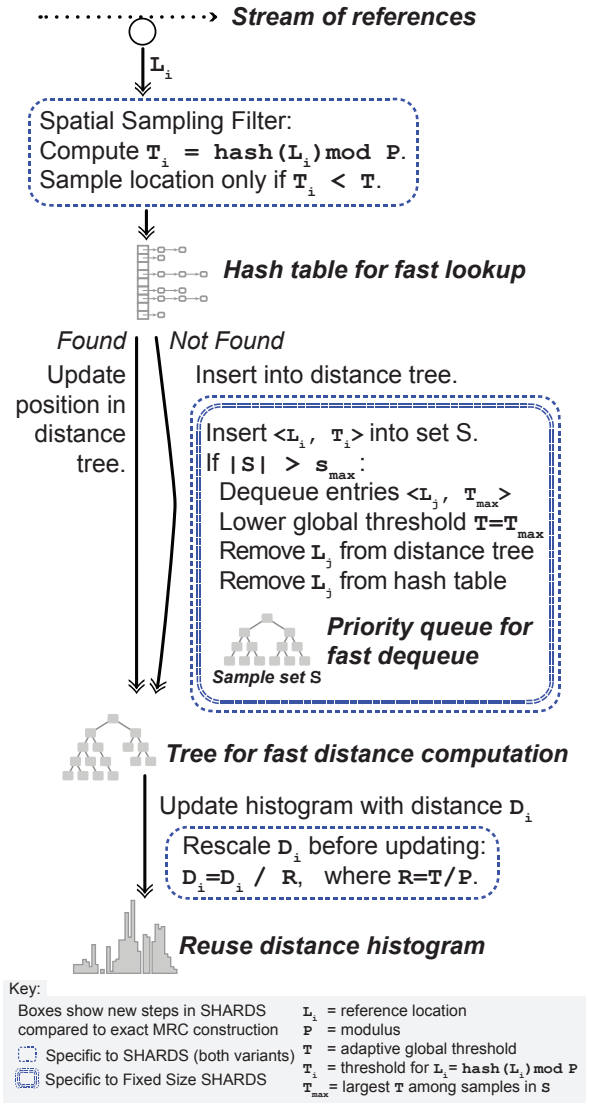

Figure 2: **Algorithm Overview.** New steps in SHARDS compared to a standard exact MRC construction algorithm.

T dynamically. When the threshold is lowered from T to T', a *subset-inclusion property* is maintained automatically. Each location sampled *after* lowering the rate would also have been sampled *prior* to lowering the rate; since $T' < T$, the samples selected with T' are a proper subset of those selected with T.

2.2 Fixed-Rate MRC Construction

Conventional reuse-distance algorithms construct an exact MRC from a complete reference trace [34, 39]. Conveniently, as shown in Figure 2, existing MRC construction implementations can be run, essentially unmodified, by providing them a sampled reference trace as input. The only modification is that each reuse distance must be scaled appropriately by $1/R$, since each sampled location statistically represents a larger number of locations.

Standard MRC construction algorithms are computationally expensive. Consider a reference stream containing N total references to M unique locations. While an optimized implementation using efficient data structures requires only $O(N \log M)$ time, it still consumes $O(M)$ space for the hash table and balanced tree used to compute reuse distances. SHARDS can be used to construct an approximate MRC in dramatically less time and space. With a fixed sampling rate R, the expected number of unique sampled locations becomes $R \cdot M$. Assuming the sampled locations are fairly representative, the total number of sampled references is reduced to approximately $R \cdot N$. As we will see in Section 4, for most workloads, $R = 0.001$ yields very accurate MRCs, using memory and processing resources that are orders of magnitude smaller than conventional approaches.

2.3 Fixed-Size MRC Construction

Fixed-*rate* MRC construction achieves a radical reduction in computational resource requirements. Nevertheless, even with a low, constant sampling rate, space requirements may still grow without bound, along with the total number of unique locations that must be tracked. For memory-constrained environments, such as production cache controller firmware where MRCs could inform cache allocation decisions, it is desirable to place an upper bound on memory size.

An additional issue is the choice of an appropriate sampling rate, R, since the accuracy of MRC approximation using spatial sampling also depends on N and M. When these values are small, it is preferable to use a relatively large value for R (such as 0.1) to improve accuracy. When these values are large, it is preferable to use a relatively small value of R (such as 0.001), to avoid wasting or exhausting available resources. Weighing these tradeoffs is difficult, especially with incomplete information.

This suggests that accuracy may depend more on an adequate sample *size* than a particular sampling *rate*. This motivates an extended version of SHARDS that constructs an MRC in $O(1)$ space and $O(N)$ time, regardless of the size or other properties of its input trace.

2.3.1 Sampling Rate Adaptation

An appropriate sampling rate is determined automatically, and need not be specified. The basic idea is to lower the sampling rate adaptively, in order to maintain a fixed bound on the total number of sampled locations that are tracked at any given point in time. The sampling rate is initialized to a high value, and is lowered gradually as more unique locations are encountered. This approach leverages the subset-inclusion property maintained by SHARDS as the rate is reduced.

Initially, the sampling rate is set to a high value, such as $R_0 = 1.0$, the maximum possible value. This is implemented by using a sampling condition of the form $hash(L) \bmod P < T$, and setting the initial threshold $T = P$, so that every location L will be selected. In practice, $R_0 = 0.1$ is sufficiently high for nearly any workload.

The goal of operating in constant space implies that we cannot continue to track all sampled references. As shown in Figure 2, a new auxiliary data structure is introduced to maintain a fixed-size set S with cardinality $|S|$. Each element of S is a tuple $\langle L_i, T_i \rangle$, consisting of an actively-sampled location L_i, and its associated threshold value, $T_i = hash(L_i) \bmod P$. Let s_{max} denote the maximum desired size $|S|$ of set S; *i.e.*, s_{max} is a constant representing an upper bound on the number of actively-sampled locations. S can be implemented efficiently as a priority queue, ordered by the tuple's threshold value.

When the first reference to a location L that satisfies the current sampling condition is processed, it is a *cold miss*, since it has never been resident in the cache. In this case, L is not already in S, so it must be added to the set. If, after adding L, the bound on the set of active locations would be exceeded, such that $|S| > s_{max}$, then the size of S must be reduced. The element $\langle L_j, T_{max} \rangle$ with the largest threshold value T_{max} is removed from the set, using a priority-queue dequeue operation. The threshold T used in the current sampling condition is reduced to T_{max}, effectively reducing the sampling rate from $R_{old} = T/P$ to a new, strictly lower rate $R_{new} = T_{max}/P$, narrowing the criteria used for future sample selection.

The corresponding location L_j is also removed from all other data structures, such as the hash table and tree used in standard implementations. If any additional elements of S have the same threshold T_{max}, then they are also removed from S in the same manner.

2.3.2 Histogram Count Rescaling

As with fixed-rate sampling, reuse distances must be scaled by $1/R$ to reflect the sampling rate. An additional consideration for the fixed-size case is that R is adjusted dynamically. 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. While this is unlikely to be true for any individual sample, it is nonetheless a reasonable statistical approximation when viewed over many sample evictions and rescaling operations. Rescaling ensures

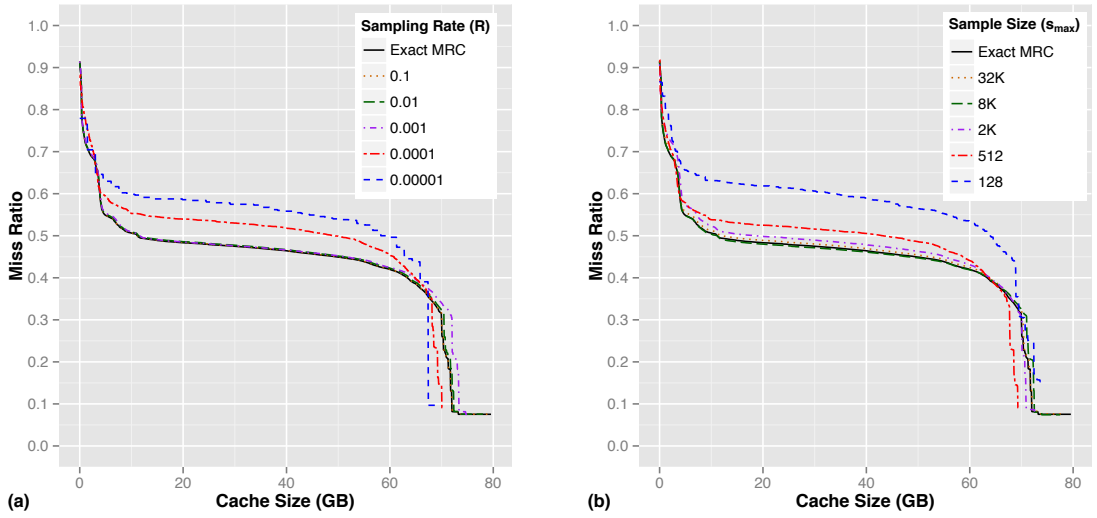

Figure 3: **Example SHARDS MRCs.** MRCs constructed for a block I/O trace containing 69.5M references to 5.2M unique blocks, using (a) fixed-rate SHARDS, varying R from 0.00001 to 0.1, and (b) fixed-size SHARDS, varying s_{max} from 128 to 32K.

that subsequent references to the remaining samples in S have the appropriate relative weight associated with their corresponding histogram bucket increments.

Conceptually, rescaling occurs immediately each time the current sampling threshold T is reduced. In practice, to avoid the expense of rescaling all histogram counts on every threshold change, it is instead performed incrementally. This is accomplished efficiently by storing T_{bucket} with each histogram bucket, representing the sampling threshold in effect when the bucket was last updated. When incrementing a bucket count, if $T_{bucket} \neq T$, then the existing count is first rescaled by T/T_{bucket}, the count is incremented, and T_{bucket} is set to T. During the final step in MRC construction, when histogram buckets are summed to generate miss ratios, any buckets for which $T_{bucket} \neq T$ need to be similarly rescaled.

3 Design and Implementation

We have developed several different implementations of SHARDS. Although designed for flexible experimentation, efficiency — especially space efficiency — was always a key goal. This section describes important aspects of both our fixed-rate and fixed-size MRC construction implementations, and discusses considerations for modeling various cache policies.

3.1 Fixed-Rate Implementation

To facilitate comparison with a known baseline, we start with the sequential version of the open-source C implementation of PARDA [39, 38]. PARDA takes a trace file as input, and performs offline reuse distance analysis, yielding an MRC. The implementation leverages two key data structures: a hash table that maps a location to the timestamp of its most recent reference, and a splay tree [48, 47] that is used to compute the number of distinct locations referenced since this timestamp.

Only a few simple modifications to the PARDA code were required to implement fixed-rate SHARDS, involving less than 50 lines of code. First, each referenced location read from the trace file is hashed, and processed only if it meets the specified sampling condition $hash(L) \bmod P < T$. For efficiency, the modulus P is set to a power of two[2] and "mod P" is replaced with the less expensive bitwise mask operation "& $(P-1)$". For a given sampling rate R, the threshold T is set to $round(R \cdot P)$. For the hash function, we used the public-domain C implementation of MurmurHash3 [3]. We also experimented with other hash functions, including a fast pseudo-random number generator [13], and found that they yielded nearly identical results.

Next, computed reuse distances are adjusted to reflect the sampling rate. Each raw distance D is simply divided by R to yield the appropriately scaled distance D/R. Since $R = T/P$, the scaled distance $(D \cdot P)/T$ is computed efficiently using an integer shift and division.

Figure 3(a) presents an example application of fixed-rate SHARDS, using a real-world storage block I/O trace[3]. The exact MRC is constructed using the unsampled, full-trace PARDA baseline. Five approximate MRCs are plotted for different fixed sampling rates, varying R between 0.00001 and 0.1, using powers of ten. The approximate curves for $R \geq 0.001$ are nearly indistinguishable from the exact MRC.

[2] We use $P = 2^{24}$, providing sufficient resolution to represent very low sampling rates, while still avoiding integer overflow when using 64-bit arithmetic for scaling operations.

[3] Customer VM disk trace $t04$, which also appears later in Figure 5.

Data structure element	$s_{max} < 64K$	$s_{max} < 4G$
hash table chain pointer	2	4
hash table entry	12	16
reference splay tree node	14	20
sample splay tree node	12	20
total per-sample size	40	60

Table 1: **Fixed-size SHARDS Data Structure Sizes.** Size (in bytes) used to represent elements of key data structures, for both 16-bit and 32-bit values of s_{max}.

3.2 Fixed-Size Implementation

With a constant memory footprint, fixed-size SHARDS is suitable for online use in memory-constrained systems, such as device drivers in embedded systems. To explore such applications, we developed a new implementation, written in C, optimized for for space efficiency.

Since all data structure sizes are known up-front, memory is allocated only during initialization. In contrast, other implementations perform a large number of dynamic allocations for individual tree nodes and hash table entries. A single, contiguous allocation is faster, and enables further space optimizations. For example, if the maximum number of samples s_{max} is bounded by 64K, "pointers" can be represented compactly as 16-bit indices instead of ordinary 64-bit addresses.

Like PARDA, our implementation leverages Sleator's public-domain splay tree code [47]. In addition to using a splay tree for computing reuse distances, we employ a second splay tree to maintain a priority queue representing the sample set S, ordered by hash threshold value. A conventional chained hash table maps locations to splay tree nodes. As an additional space optimization, references between data structures are encoded using small indices instead of general-purpose pointers.

The combined effect of these space-saving optimizations is summarized in Table 1, which reports the per-sample sizes for key data structures. Additional memory is needed for the output histogram; each bucket consumes 12 bytes to store a count and the update threshold T_{bucket} used for rescaling. For example, with s_{max} = 8K, the aggregate overhead for samples is only 320 KB. Using 10K histogram buckets, providing high resolution for evaluating cache allocation sizes, consumes another 120 KB. Even when code size, stack space, and all other memory usage is considered, the entire measured runtime footprint remains smaller than 1 MB, making this implementation practical even for extremely memory-constrained execution environments.

Figure 3(b) presents an example application of fixed-size SHARDS, using the same trace as Figure 3(a). Five approximate MRCs are plotted for different fixed sample sizes, varying s_{max} between 128 and 32K, using factors of four. The approximate curves for $s_{max} \geq 2K$ are nearly indistinguishable from the exact MRC.

3.3 Modeling Cache Policy

PARDA uses a simple binary trace format: a sequence of 64-bit references, with no additional metadata. Storage I/O traces typically contain richer information for each reference, including a timestamp, access type (read or write), and a location represented as an offset and length.

For the experiments in this paper, we converted I/O block traces to the simpler PARDA format, assumed a fixed cache block size, and ignored the distinction between reads and writes. This effectively models a simple LRU policy with fixed access granularity, where the first access to a block is counted as a miss.

We have also developed other SHARDS implementations to simulate diverse caching policies. For example, on a write miss to a partial cache block, a write-through cache may first read the entire enclosing cache-block-sized region from storage. The extra read overhead caused by partial writes can be modeled by maintaining separate histograms for ordinary reads and reads induced by partial writes. Other write-through caches manage partial writes at sub-block granularity, modeled using known techniques [57]. In all cases, we found hash-based spatial sampling to be extremely effective.

4 Experimental Evaluation

We conducted a series of experiments with over a hundred real-world I/O traces collected from our commercial caching analytics service for virtualized environments. We first describe our data collection system and characterize the trace files used in this paper. Next, we evaluate the accuracy of approximate MRCs. Finally, we present results of performance experiments that demonstrate the space and time efficiency of our implementations.

4.1 Data Collection

Our SaaS caching analytics service is designed to collect block I/O traces for VMware virtual disks in customer data centers running the VMware ESXi hypervisor [60]. A user-mode application, deployed on each ESXi host, coordinates with the standard VMware *vscsiStats* utility [1] to collect complete block I/O traces for VM virtual disks. A web-based interface allows particular virtual disks to be selected for tracing remotely.

Compressed traces are streamed to a cloud-based backend to perform various storage analyses, including offline MRC construction using SHARDS. If the trace is not needed for additional storage analysis, SHARDS sampling could be performed locally, obviating the need to stream full traces. Ideally, SHARDS should be integrated directly with the kernel-mode hypervisor component of *vscsiStats* for maximum efficiency, enabling continuous, online reuse-distance analysis.

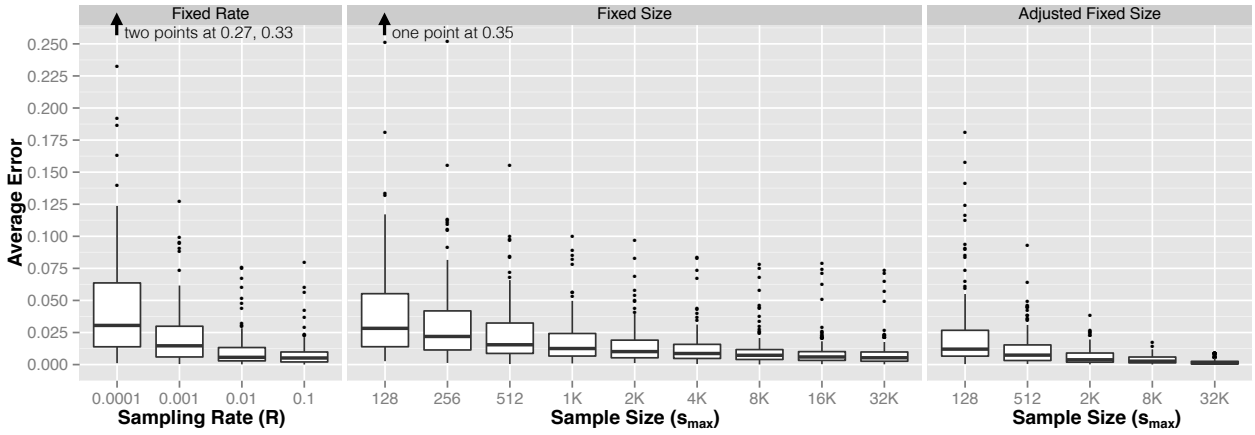

Figure 4: **Error Analysis.** Average absolute error calculated over all 124 traces for different SHARDS sampling parameters. The fixed-rate and fixed-size results are explained in Section 4.3.2, and the adjusted fixed-size results are discussed in Section 4.3.3.

4.2 Trace Files

We use 106 week-long vscsiStats traces, collected by our caching analytics service from virtual disks in production customer environments. These traces represent VMware virtual disks with sizes ranging from 8 GB to 34 TB, with a median of 90 GB. The associated VMs are 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 included several publicly-available block I/O traces from the SNIA IOTTA repository [51]. We used a dozen week-long enterprise server traces collected by Microsoft Research Cambridge [37], as well as six day-long server traces collected by FIU [31]. In total, this gives us a diverse set of 124 real-world block I/O traces to evaluate the accuracy and performance of SHARDS compared to exact methods.

4.3 Accuracy

We analyze the accuracy of MRCs constructed using SHARDS by comparing them to corresponding exact MRCs without sampling. Differences between the approximate and exact curves are measured over a wide range of sampling parameters. Numerous MRC plots are displayed as visual examples of SHARDS' accuracy.

4.3.1 Parameters

Our system supports many configuration parameters. We specify a 16 KB cache block size, so that a cache miss reads from primary storage in aligned, fixed-size 16 KB units; typical storage caches in commercial virtualized systems employ values between 4 KB and 64 KB. As discussed in Section 3.3, reads and writes are treated identically, effectively modeling a simple LRU cache policy. By default, we specify a histogram bucket size of 4K cache blocks, so that each bucket represents 64 MB.

Fixed-rate sampling is characterized by a single parameter, the sampling rate R, which we vary between 0.0001 and 0.1 using powers of ten. Fixed-size sampling has two parameters: the sample set size, s_{max}, and the initial sampling rate, R_0. We vary s_{max} using powers of two between 128 and 32K, and use $R_0 = 0.1$, since this rate is sufficiently high to work well with even small traces.

4.3.2 Error Metric

To analyze the accuracy of SHARDS, we consider the difference between each approximate MRC, constructed using hash-based spatial 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 [53, 43, 65], 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.

The box plots[4] in Figure 4 show the MAE metric for a wide range of fixed-rate and fixed-size sampling parameters. 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 small, even for low sampling rates and small sample sizes. Fixed-rate sampling with $R = 0.001$ results in approximate MRCs with a median MAE of less than 0.02; most exhibit an MAE bounded by 0.05. The error for fixed-rate SHARDS typically has larger variance than fixed-size SHARDS, indicating that accuracy is better controlled via sample count than sampling rate.

For fixed-size SHARDS with s_{max} = 8K, the median MAE is 0.0072, with a worst-case of 0.078. Aside from a few outliers (13 traces), the error is bounded by 0.021.

[4]The top and the bottom of each box represent the first and third quartile values of the error. The thin whiskers represents the min and max error, excluding outliers. Outliers, represented by dots, are the values larger than $Q_3 + 1.5 \times IQR$, where $IQR = Q_3 - Q_1$.

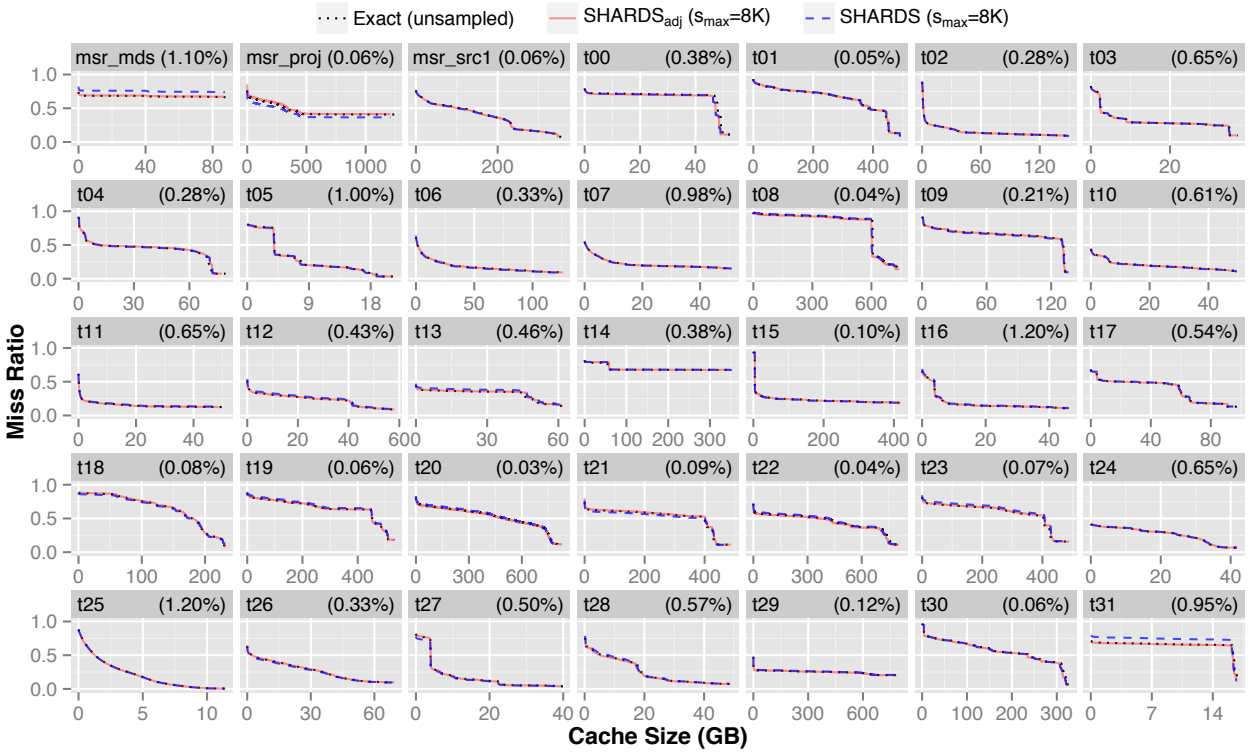

Figure 5: **Example MRCs: Exact vs. Fixed-Size SHARDS.** Exact and approximate MRCs for 35 representative traces. Approximate MRCs are constructed using fixed-size SHARDS and SHARDS$_{adj}$ with s_{max} = 8K. Trace names are shown for three public MSR traces [37]; others are anonymized as $t00$ through $t31$. The effective sampling rates appear in parentheses.

4.3.3 Using Reference Estimates to Reduce Error

In cases where SHARDS exhibits non-trivial error, we find that a coarse "vertical shift" accounts for most of the difference, while finer features are modeled accurately. This effect is seen in Figure 3. Recently, this observation led us to develop SHARDS$_{adj}$, a simple adjustment that improves accuracy significantly at virtually no cost.

Spatial sampling selects a static set of blocks. If the dynamic behavior of the sample set differs too much from that of the complete trace, the weights of the sums of buckets and the total count of accesses from the reuse histogram will be off, skewing the resulting MRC. For example, excluding too many or too few very hot blocks biases dynamic access counts.

Ideally, the average number of repetitions per block should be the same for both the sample set and the complete trace. This happens when the actual number of sampled references, N_s, matches the expected number, $E[N_s] = N \cdot R$. When this does not occur, we find that it is because the sample set contains the wrong proportion of frequently accessed blocks. Our correction simply adds the difference, $E[N_s] - N_s$, to the first histogram bucket before computing final miss ratios.

We now consider this adjustment to be best practice. Although there was insufficient time to update all of our earlier experiments, SHARDS$_{adj}$ results appear in Figures 4 and 5. Figure 4 reveals that the error with SHARDS$_{adj}$ is significantly lower. Across all 124 traces, this adjustment reduces the median fixed-size SHARDS error with s_{max} = 8K to 0.0027, and the worst-case to 0.017, factors of nearly 3× and 5×, respectively. Excluding the two outliers, MAE is bounded at 0.012. Even with just 128 samples, the median MAE is only 0.012.

4.3.4 Example MRCs

The quantitative error measurements reveal that, for nearly all traces, with fixed-size sampling at s_{max} = 8K, the miss ratios in the approximate MRCs deviate only slightly from the corresponding exact MRCs. Although space limitations prevent us from showing MRCs for *all* of the traces described in Section 4.2, we present a large number of small plots for this practical configuration.

Figure 5 plots 35 approximate MRCs, together with the matching exact curves; in most cases, the curves are nearly indistinguishable. In all cases, the location of prominent features, such as steep descents, appear faithful. Each plot is annotated with the effective dynamic sampling rate, indicating the fraction of IOs processed, including evicted samples. This rate reflects the amount of processing required to construct the MRC.

SHARDS$_{adj}$ effectively corrects all cases with visible error. For trace $t31$, the worst case over all 124 traces for SHARDS, error is reduced from 0.078 to 0.008.

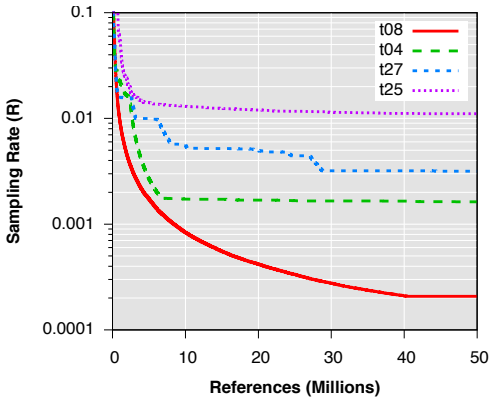

Figure 6: **Dynamic Rate Adaptation.** Sampling rate R (on log scale) for four traces over time. Each starts at $R_0 = 0.1$, and is lowered dynamically as more unique references are sampled.

4.3.5 Sampling Rate Adaptation

Choosing a sampling rate that achieves high accuracy with good efficiency is challenging. The automatic rate adaptation of fixed-size SHARDS is advantageous because it eliminates the need to specify R. Figure 6 plots R as a function of reference count for four diverse traces: *t08*, *t04*, *t27*, and *t25* from Figure 5. For each, the sampling rate starts at a high initial value of $R_0 = 0.1$, and is lowered progressively as more unique locations are encountered. The figure shows that SHARDS adapts automatically for each of the traces, which contain significantly different numbers of unique references. After 50 million references, the values of R for these traces are 0.0002, 0.0016, 0.0032, and 0.0111. The total number of samples processed, including evictions from the fixed-size sample set S, is given by the area under each curve.

4.3.6 Discussion

Quantitative experiments confirm that, for nearly all workloads, SHARDS yields accurate MRCs, in radically less time and space than conventional exact algorithms. While the accuracy achieved with high sampling rates may not be surprising, success with very low rates, such as $R = 0.001$, was unexpected. Even more extraordinary is the ability to construct accurate MRCs for a broad range of workloads, using only a small constant number of samples, such as s_{max} = 8K, or even s_{max} = 256. The constant-space and rate-adaptation properties of fixed-size SHARDS make it our preferred approach.

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

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 4 against $1/\sqrt{s_{max}}$ resulted in a high correlation coefficient of $r^2 = 0.97$. This explains the observed diminishing accuracy improvements with increasing s_{max}.

4.4 Performance

We conducted performance experiments in a VMware virtual machine, using a 64-bit Ubuntu 12.04 guest running Linux kernel version 3.2.0. The VM was configured with 64 GB RAM, and 8 virtual cores, and executed on an under-committed physical host running VMware ESXi 5.5, configured with 128 GB RAM and 32 AMD Opteron x86-64 cores running at 2 GHz.

To quantify the performance advantages of SHARDS over exact MRC construction, we use a modern high-performance reuse-distance algorithm from the open-source PARDA implementation [39, 38] as our baseline. Although the main innovation of PARDA is a parallel reuse distance algorithm, we use the same sequential "classical tree-based stack distance" baseline as in their paper. The PARDA parallelization technique would also result in further performance gains for SHARDS.

4.4.1 Space

To enable a fair comparison of memory consumption with SHARDS, we implemented minor extensions to PARDA, adding command-line options to specify the number of output histogram buckets and the histogram bucket width.[5] We also added code to both PARDA and SHARDS to obtain accurate runtime memory usage[6].

All experiments were run over the full set of traces described in Section 4.2. Each run was configured with 10 thousand histogram buckets, each 64 MB wide (4K cache blocks of size 16 KB), resulting in an MRC for cache allocations up to 640 GB.

Sequential PARDA serves as a baseline, representing an efficient, exact MRC construction algorithm without sampling. Fixed-rate SHARDS, implemented via the simple code modifications described in Section 3.1, is configured with $R = 0.01$ and $R = 0.001$. Finally, the new space-efficient fixed-size SHARDS implementation, presented in Section 3.2, is run with s_{max} = 8K and $R_0 = 0.1$.

Figure 7 shows the memory usage for each algorithm over the full set of traces, ordered by baseline memory consumption. The drastic reductions with SHARDS required the use of a log scale. As expected, for traces

[5] By default, PARDA is configured with hard-coded values – 1M buckets, each a single cache block wide.

[6] We obtain the peak resident set size directly from the Linux procfs node `/proc/<pid>/status` immediately before terminating; the `VmHWM` line reports the "high water mark" [32].

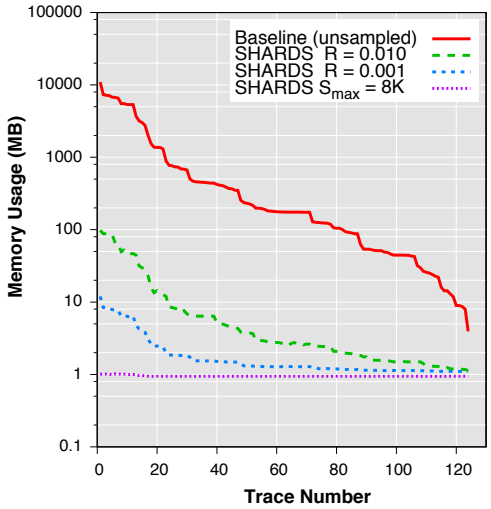

Figure 7: **Memory Usage.** Measured memory consumption (in MB, log scale) for unsampled baseline, fixed-rate SHARDS ($R = 0.01, 0.001$), and fixed-size SHARDS ($s_{max} = 8K$).

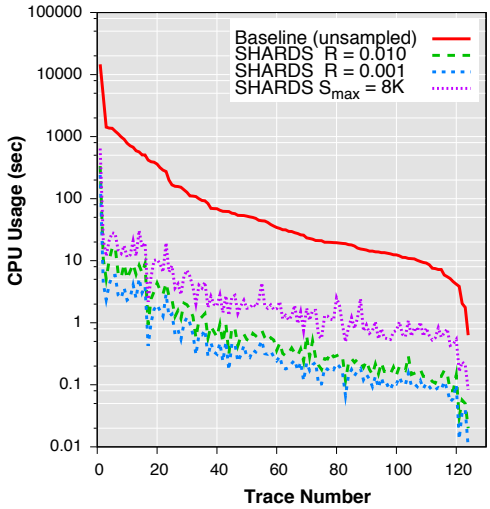

Figure 8: **CPU Usage.** Measured run time (in seconds, log scale) for unsampled baseline, fixed-rate SHARDS ($R = 0.01$, 0.001), and fixed-size SHARDS ($s_{max} = 8K$).

with large numbers of unique references, the memory required for fixed-rate SHARDS is approximately R times as big as the baseline. With much smaller traces, fixed overheads dominate. For fixed-size SHARDS, the runtime footprint remained approximately 1 MB for all runs, ranging from 964 KB to 1,044 KB, with an average of 974 KB, yielding a savings of up to $10,800\times$ for large traces and a median of $185\times$ across all traces.

4.4.2 Time

Figure 8 plots the CPU usage measured[7] for the same runs described above, ordered by baseline CPU consumption. The significant processing time reductions with SHARDS prompted the use of a log scale.

Fixed-rate SHARDS with $R = 0.01$ results in speedups over the baseline ranging from $29\times$ to $449\times$, with a median of $75\times$. For $R = 0.001$, the improvement ranges from $41\times$ to $1,029\times$, with a median of $128\times$. For short traces with relatively small numbers of unique references, fixed overheads dominate, limiting speedups to values lower than implied by R.

Fixed-size SHARDS with $s_{max} = 8K$ and $R_0 = 0.1$ incurs more overhead than fixed-rate SHARDS with $R = 0.01$. This is due to the non-trivial work associated with evicted samples as the sampling rate adapts dynamically, as well as the cost of updating the sample set priority queue. Nonetheless, fixed-size SHARDS achieves significant speedups over the baseline, ranging from $6\times$ to $204\times$, with a median of $22\times$. In terms of throughput, for the top three traces ordered by CPU consumption in Figure 8, fixed-size SHARDS processes an average of 15.4 million references per second.

[7]For each run, CPU time was obtained by adding the user and system time components reported by `/usr/bin/time`.

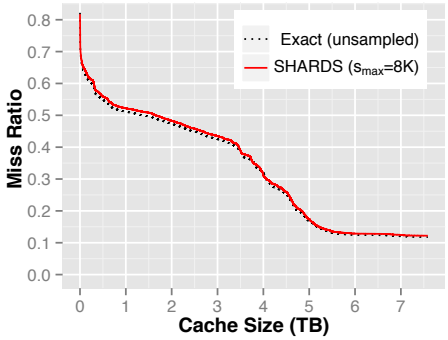

Figure 9: **Mixed Workloads.** Exact and approximate MRCs for merged trace interleaving 4.3G IOs to 509M unique blocks from 32 separate virtual disks. Fixed-size SHARDS with $s_{max} = 8K$ exhibits an average absolute error of only 0.008.

4.5 MRCs for Mixed Workloads

Our VM-based traces represent single-machine workloads, while the IOs received by storage arrays are typically an undistinguished, blended mix of numerous independent workloads. Figure 9 demonstrates the accuracy of fixed-size SHARDS using a relative-time-interleaved reference stream combining all 32 virtual disk traces (*t00*...*t31*) shown in Figure 5. With $s_{max} = 8K$, SHARDS exhibits a small MAE of 0.008. The high accuracy and extremely low overhead provide additional confidence that continuous, online MRC construction and analysis is finally practical for production storage arrays.

4.6 Non-LRU Replacement Policies

SHARDS constructs MRCs for a cache using an LRU replacement policy. Significantly, the same underlying hash-based spatial sampling approach appears promising for simulating more sophisticated policies, such as LIRS [27], ARC [35], CAR [5], or Clock-Pro [26].

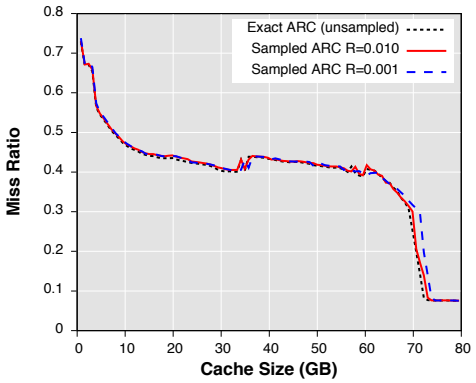

Figure 10: **Scaled-Down ARC Simulation.** Exact and approximate MRCs for VM disk trace $t04$. Each curve plots 100 separate ARC simulations at different cache sizes.

As with fixed-rate SHARDS, the input trace is filtered to select blocks that satisfy 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 10 presents results for the same trace as in Figure 3, leveraging an open-source ARC implementation [21]. For $R = 0.001$, 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 0.01.

5 Related Work

Before the seminal paper of Mattson, Gecsei, Slutz, and Traiger [34], studies of memory and storage systems required running separate experiments for each size of a given level of the hierarchy. Their key insight is that many replacement policies have an inclusion property: given a cache C of size z, $C(z) \subseteq C(z+1)$. Such policies, referred to as *stack algorithms*, include LRU, LFU, and MRU.[8] Mattson *et al.* also model set associativity, deletion, no-write-allocate policies, and the handling of the entire set of memory and storage hierarchies as a list of such stacks. Others extended the model for caches with write-back or subblocking policies [56, 57], variable-size pages [58], set associativity [24, 30, 52], and modeling groups of these behaviors in a single analysis [25, 59].

Because it generates models of behavior for all cache sizes in a single pass over a trace, Mattson's technique has been applied widely. Application areas include the modeling of caches [24, 30, 52]; of multicore caches including the effects of invalidation [45, 44]; guidance of mechanisms to apportion shared caches amongst processes [41, 53, 18]; the scheduling of memory within an operating system [49, 72, 4]; the sizing and management of unified buffer caches [28]; secondary exclusive (victim I/O) caches [33], and memory caches [43]; the sizing of garbage-collected heaps [68, 67]; the impact of memory systems and caches on Java performance [29]; the transparent borrowing of memory for low-priority computation [14]; the balancing of memory across sets of virtual machines [70, 69]; and the analysis of program behavior and compilation for data layout [2, 11, 17].

[8] Policies other than LRU require one more step: after a block is moved to the top, the remaining blocks are reordered in a single pass.

5.1 Optimizations

Mattson's algorithm takes $O(NM)$ time and $O(M)$ space for a trace of length N containing M unique blocks. Given its broad applicability, much effort has been spent improving its performance in both space and time.

5.1.1 Management of LRU Stacks

Early improvements added hash tables either to detect cold accesses or to map references to their previous entries [6, 40, 56]. Bennett and Kruskal [6] used a balanced tree over the trace tracking which references were the most recent instances and keeping counts in the subtrees, allowing each distance to be computed in $O(\log N)$ steps. Olken [40] improved on this by tracking only the most recent references in the tree, further reducing the distance computation to $O(\log M)$. Use of a balanced tree is now common for computing distances. By managing the stack of references with a doubly-linked or chunked list and mapping references to nodes in the stack, the algorithm takes $O(N \log M)$ time and $O(M)$ space.

Another key advance, by Niu *et al.*, is processing a trace in parallel [39]. Their technique, PARDA, achieves impressive speedups by splitting a trace into P partitions, processing each independently, and patching up missing information between partitions.

5.1.2 Compression and Grouping of References

Much effort has been spent reducing trace sizes [50, 36]. Smith [50] compressed virtual-memory traces by ignoring references whose reuse distance is less than some K, and periodically scanning access-bits for pages.

Another optimization is to coarsen the distances that are tracked [52, 72, 68, 67, 4, 70, 43]. Kim *et al.* [30] track groups of references in the stack where the sizes of the groups are powers of two. By tracking the boundary of each group of references, updates to the LRU stack simply adjust the distances for the pages pushed across these boundaries. This can reduce costs of tracking accesses to $O(G)$ where G is the number of groups tracked. Recently, Saemundsson *et al.* [43] 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 [42], but the space complexity remains $O(M)$. Zhao *et al.* [70] report an error rate of 10% for their level of coarseness, 32 pages. None of the others report error rates.

Ding and Zhong [17] apply clustering in the context of the splay tree they use to track reuse-distances for pro-

gram analysis. By dynamically compressing the tree, they bound the overall size and cost of their analysis, achieving a time bound of $O(N \log\log M)$ and a space bound of $O(\log M)$. The relative error is bounded by how the nodes in the tree are merged and compressed and, so, factors in as the base of the log. For an error of 50%, the base is 2; for smaller ones, the base quickly approaches 1. For their purpose, they can tolerate large error bounds.

5.1.3 Temporal Sampling

Temporal sampling — complementary to SHARDS — reduces reference-tracking costs by only doing so some of the time. Berg *et al.* [7, 8] sample every *N*th reference (1 in every 10K) to derive MRCs for caches. Bryan and Conte's cluster sampling [10], RapidMRC [53], and work on low-cost tracking for VMs [69], by contrast, divide the execution into periods in which references are either sampled or are not. They also tackle how to detect phase changes that require regeneration of the reuse-distances. RapidMRC reports a mean average error rate of 1.02 misses per thousand instructions (MPKI) with a maximum of 6.57 MPKI observed. Zhao *et al.* [69] report mean relative errors of 3.9% to 12.6%. These errors are significantly larger than what SHARDS achieves.

Use of sampling periods allows for accurate measurements of reuse distances within a sample period. However, Zhong and Chang [71] and Schuff *et al.* [45, 44] observe that naively sampling every *N*th reference as Berg *et al.* do or using simple sampling phases causes a bias against the tracking of longer reuse distances. Both efforts address this bias by sampling references during a sampling period and then following their next accesses across subsequent sampling and non-sampling phases.

5.1.4 Spatial and Content-Based Sampling

A challenge when sampling references is that reuse-distance is a recurrent behavior. One solution is to extract a sample from the trace based on an identifying characteristic of its references. Spatial sampling uses addresses to select a sample set. Content-based sampling does so by using data contents. Both techniques can capture all events for a set of references, even those that occur rarely.

Many analyses for set-associative caches have used set-sampling [23, 41, 46]. For example, UMON-DSS [41] reduces the cost of collecting reuse-distances by sampling the behavior of a subset of the sets in a processor cache. Kessler *et al.* [23] compare temporal sampling, set-sampling and constant-bit sampling of references and find that the last technique is most useful when studying set-associative caches of different dimensions.

Many techniques targeting hardware implementations use grouping or spatial sampling to constrain their use of space [72, 41, 4, 59, 46]. However, these tend to focus on narrow problems such as limited set associativity [41] or limited cache size ranges [4] for each MRC. Like these approaches, SHARDS reduces and bounds space use, but unlike them, it models the full range of cache sizes. In addition, these techniques do not report error rates.

Inspired by processor hardware for cache sampling, Waldspurger *et al.* propose constructing an MRC by sampling a fixed set of pages from the guest-physical memory of a VM [62]. Unfortunately, practical sampling requires using small (4 KB) pages, increasing the overhead of memory virtualization [9]. Choosing sampled locations up-front is also inefficient, especially for workloads with large, sparse address spaces. In contrast, SHARDS does not require any information about the address space.

Xie *et al.* address a different problem: estimation of duplication among blocks in a storage system [66]. Their system hashes the contents of blocks producing fingerprints. These are partitioned into sets with one set chosen as the sample. Their model has error proportional to the sample-set size. This property is used to dynamically repartition the sample so that the sample size is bounded. Like Xie *et al.*, the SHARDS sampling rate can be adjusted to ensure an upper bound on the space used. But, how the sample set is chosen, how the sampling rate is adjusted, and how the sampling ratio is used to adjust the summary information are different. Most importantly, where their work looks at individual blocks' hash values and how these collide, our technique accurately captures the relationship *between* pairs of *accesses* to the blocks.

5.2 Analytical Models

Many analytical models have been proposed to approximate MRCs with reduced effort. By constraining the block replacement policy, Tay and Zou [55] derive a universal equation that models cache behavior from a small set of sampled data points. He *et al.* propose modeling MRCs as fractals and claim error rates of 7-10% in many cases with low overhead [22]. Berg *et al.* [7, 8, 19, 18] use a closed-form equation of the miss rate. Through a sequence of sampling, deriving local miss rates and combining these separate curves, they model caches with random or LRU replacement. Others model cache behavior by tracking hardware performance counters [15, 63, 46].

Unlike the analytical approaches, SHARDS estimates the MRC directly from the sampled trace. We have shown that SHARDS can be implemented using constant space and with high accuracy. Where the error of SHARDS is small, the analytic techniques report errors of a few percent to 50% with some outliers at 100-200%. Berg *et al.* simply offer graphs for comparison.

5.3 Counter Stacks

Mattson *et al.* track distances as counts of unique references between reuses. Wires *et al.* extend this in three ways in their recent MRC approximation work, using a *counter stack* [65].

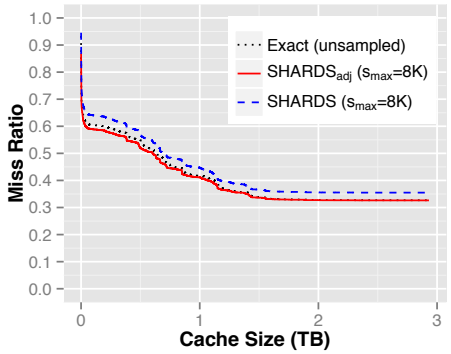

Figure 11: **Merged MSR Trace.** Exact, SHARDS and SHARDS$_{adj}$ MRCs for the merged "master" MSR trace used in the Counter Stacks evaluation [65], with s_{max} = 8K.

First, the counts of repetitions, themselves, can be computed by comparing changes in the number of unique references seen from different starting points in the stream. The sequence of locations observed by a newer counter is a proper suffix of the sequence recorded by an older one. So, if the newer counter increases but the older does not, then the older location must have repeated, and its reuse-distance is the older counter's value.

Second, the repetitions and reuse-distances can be approximated efficiently using a bounded set of counters. Instead of starting a new counter with every reference, one may *downsample* the set of counters, creating and tracking a new one periodically. The set can be further *pruned* since, over time, adjacent counters converge as they observe the same set of elements. Using probabilistic counters based on the HyperLogLog algorithm [20] together with downsampling and pruning, the counter stack algorithm only uses $O(\log M)$ space.

Third, columns of counts in the counter stack can be periodically written to a checkpoint together with timestamps for subsequent analysis. Checkpointed counter-stack sequences can be spliced, shifted temporally, and combined to model the behavior of combinations of workloads. Because the checkpoint only captures stacks of counts at each timestamp, such modeling assumes that different checkpoints access disjoint sets of blocks.

To provide a direct quantitative comparison with SHARDS, we generated the same merged "master" MSR trace used by Wires *et al.* [65], configured identically with only read requests and a 4 KB cache block size. Figure 11 shows MRCs constructed using fixed-size SHARDS, with 48K histogram buckets of size 64 MB, supporting cache sizes up to 3 TB. For s_{max} = 8K, the MAE is 0.006 with SHARDS$_{adj}$ (0.029 unadjusted). The MRC is computed using only 1.3 MB of memory in 137 seconds, processing 17.6M blocks/sec. Wires *et al.* report that Counter Stacks requires 80 MB of memory, and 1,034 seconds to process this trace at a rate of 2.3M blocks/sec. In this case, Counter Stacks is approximately 7× slower and needs 62× as much memory as SHARDS$_{adj}$, but is more accurate, with an MAE of only 0.0025 [64]. Using s_{max} = 32K, with 2 MB of memory in 142 seconds, yields a comparable MAE of 0.0026.

While Counter Stacks uses $O(\log M)$ space, fixed-size SHARDS computes MRCs in small *constant* space. As a result, separate SHARDS instances can efficiently compute multiple MRCs tracking different properties or time-scales for a given reference stream, something Wires *et al.* claim is not practical.

One advantage of Counter Stacks is that every reference affects the probabilistic counters and contributes to the resulting MRC. By contrast, SHARDS assumes that hashing generates a uniformly distributed set of values for a reference stream. While an adversarial trace could yield an inaccurate MRC, we have not encountered one.

Unlike Counter Stacks, SHARDS maintains the identity of each block in its sample set. This enables tracking additional information, including access frequency, making it possible to directly implement other policies such as LFU, LIRS [27], ARC [35], CAR [5], or Clock-Pro [26], as discussed in Section 4.6.

6 Conclusions

We have introduced SHARDS, a new hash-based spatial sampling technique for reuse-distance analysis that computes approximate miss ratio curves accurately using only modest computational resources. The approach is so lightweight — operating in constant space, and requiring several orders of magnitude less processing than conventional algorithms — that online MRC construction becomes practical. Furthermore, SHARDS enables offline analysis for long traces that, due to memory constraints, could not be studied using exact techniques.

Our experimental evaluation of SHARDS demonstrates its accuracy, robustness, and performance advantages, over a large collection of I/O traces from real-world production storage systems. Quantitative results show that, for most workloads, an approximate sampled MRC that differs only slightly from an exact MRC can be constructed in 1 MB of memory. Performance analysis highlights dramatic reductions in resource consumption, up to 10,800× in memory and up to 204× in CPU.

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

Acknowledgments Thanks to the anonymous reviewers, our shepherd Arif Merchant, Jim Kleckner, Xiaojun Liu, Guang Yang, John Blumenthal, and Jeff Hausman for their valuable feedback and support.

References

[1] AHMAD, I. Easy and efficient disk I/O workload characterization in VMware ESX Server. In *Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization* (Washington, DC, USA, 2007), IISWC '07, IEEE Computer Society, pp. 149–158.

[2] ALMÁSI, G., CAŞCAVAL, C., AND PADUA, D. A. Calculating stack distances efficiently. *SIGPLAN Not. 38*, 2 supplement (June 2002), 37–43.

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

[4] AZIMI, R., SOARES, L., STUMM, M., WALSH, T., AND BROWN, A. D. Path: Page access tracking to improve memory management. In *Proceedings of the 6th International Symposium on Memory Management* (New York, NY, USA, 2007), ISMM '07, ACM, pp. 31–42.

[5] BANSAL, S., AND MODHA, D. S. CAR: Clock with adaptive replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2004), FAST '04, USENIX Association, pp. 187–200.

[6] BENNETT, B. T., AND KRUSKAL, V. J. LRU stack processing. *IBM Journal of Research and Development 19* (1975), 353–357.

[7] BERG, E., AND HAGERSTEN, E. StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2004)* (Austin, Texas, USA, Mar. 2004).

[8] BERG, E., AND HAGERSTEN, E. Fast Data-Locality Profiling of Native Execution. In *Proceedings of ACM SIGMETRICS 2005* (Banff, Canada, June 2005).

[9] BHARGAVA, R., SEREBRIN, B., SPADINI, F., AND MANNE, S. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2008), ASPLOS XIII, ACM, pp. 26–35.

[10] BRYAN, P. D., AND CONTE, T. M. Combining cluster sampling with single pass methods for efficient sampling regimen design. In *25th International Conference on Computer Design, ICCD 2007, 7-10 October 2007, Lake Tahoe, CA, USA, Proceedings* (2007), IEEE, pp. 472–479.

[11] CAŞCAVAL, C., AND PADUA, D. A. Estimating cache misses and locality using stack distances. In *Proceedings of the 17th Annual International Conference on Supercomputing* (New York, NY, USA, 2003), ICS '03, ACM, pp. 150–159.

[12] CARR, R. W., AND HENNESSY, J. L. WSCLOCK–a simple and effective algorithm for virtual memory management. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1981), SOSP '81, ACM, pp. 87–95.

[13] CARTA, D. F. Two fast implementations of the minimal standard random number generator. *CACM 33*, 1 (Jan. 1990), 87–88.

[14] CIPAR, J., CORNER, M. D., AND BERGER, E. D. Transparent contribution of memory. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference* (Berkeley, CA, USA, 2006), ATEC '06, USENIX Association, pp. 11–11.

[15] CONTE, T. M., HIRSCH, M. A., AND HWU, W.-M. W. Combining trace sampling with single pass methods for efficient cache simulation. *IEEE Trans. Comput. 47*, 6 (June 1998), 714–720.

[16] DENNING, P. J. The working set model for program behavior. *Commun. ACM 11*, 5 (May 1968), 323–333.

[17] DING, C., AND ZHONG, Y. Predicting whole-program locality through reuse distance analysis. *SIGPLAN Not. 38*, 5 (May 2003), 245–257.

[18] EKLOV, D., BLACK-SCHAFFER, D., AND HAGERSTEN, E. Fast modeling of shared caches in multicore systems. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers* (New York, NY, USA, 2011), HiPEAC '11, ACM, pp. 147–157.

[19] EKLOV, D., AND HAGERSTEN, E. StatStack: Efficient modeling of LRU caches. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on* (March 2010), pp. 55–65.

[20] FLAJOLET, P., FUSY, E., GANDOUET, O., AND MEUNIER, F. HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm. In *Proceedings of the 2007 International Conference on Analysis of Algorithms (AOFA '07)* (2007), pp. 127–146.

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

[22] HE, L., YU, Z., AND JIN, H. FractalMRC: Online cache miss rate curve prediction on commodity systems. In *IPDPS'12* (2012), pp. 1341–1351.

[23] HILL, K. M., KESSLER, R. E., HILL, M. D., AND WOOD, D. A. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Transactions on Computers 43* (1994), 664–675.

[24] HILL, M. D., AND SMITH, A. J. Evaluating associativity in CPU caches. *IEEE Trans. Comput. 38*, 12 (Dec. 1989), 1612–1630.

[25] JANAPSATYA, A., IGNJATOVIĆ, A., AND PARAMESWARAN, S. Finding optimal L1 cache configuration for embedded systems. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference* (Piscataway, NJ, USA, 2006), ASP-DAC '06, IEEE Press, pp. 796–801.

[26] JIANG, S., CHEN, F., AND ZHANG, X. Clock-pro: An effective improvement of the clock replacement. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), ATEC '05, USENIX Association, pp. 35–35.

[27] 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.

[28] KIM, J. M., CHOI, J., KIM, J., NOH, S. H., MIN, S. L., CHO, Y., AND KIM, C. S. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Proceedings of the 4th Symposium on Operating System Design and Implementation – Volume 4* (Berkeley, CA, USA, 2000), OSDI'00, USENIX Association, pp. 9–9.

[29] KIM, J.-S., AND HSU, Y. Memory system behavior of Java programs: Methodology and analysis. In *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2000), SIGMETRICS '00, ACM, pp. 264–274.

[30] KIM, Y. H., HILL, M. D., AND WOOD, D. A. Implementing stack simulation for highly-associative memories. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 1991), SIGMETRICS '91, ACM, pp. 212–213.

[31] 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.

[32] LINUX PROGRAMMER'S MANUAL. proc(5) Linux manual page. http://man7.org/linux/man-pages/man5/proc.5.html.

[33] LU, P., AND SHEN, K. Multi-layer event trace analysis for parallel I/O performance tuning. *2013 42nd International Conference on Parallel Processing* (2007), 12.

[34] MATTSON, R. L., GECSEI, J., SLUTZ, D. R., AND TRAIGER, I. L. Evaluation techniques for storage hierarchies. *IBM Syst. J. 9*, 2 (June 1970), 78–117.

[35] 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.

[36] MICHAUD, P. Online compression of cache-filtered address traces. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on* (April 2009), pp. 185–194.

[37] 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.

[38] NIU, Q. PARDA git repository. https://bitbucket.org/niuqingpeng/file_parda/.

[39] 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.

[40] OLKEN, F. Efficient methods for calculating the success function of fixed space replacement policies. *Perform. Eval. 3*, 2 (1983), 153–154.

[41] 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.

[42] SAEMUNDSSON, T. Private communication, Jan 2015.

[43] 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.

[44] SCHUFF, D. L., KULKARNI, M., AND PAI, V. S. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2010), PACT '10, ACM, pp. 53–64.

[45] SCHUFF, D. L., PARSONS, B. S., AND PAI, V. S. Multicore-aware reuse distance analysis. Tech. Rep. ECE-TR-388, Purdue University, Sep 2009.

[46] SEN, R., AND WOOD, D. A. Reuse-based online models for caches. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2013), SIGMETRICS '13, ACM, pp. 279–292.

[47] SLEATOR, D. An implementation of top-down splaying with sizes. ftp://ftp.cs.cmu.edu/usr/ftp/usr/sleator/splaying.

[48] SLEATOR, D. D., AND TARJAN, R. E. Self-adjusting binary search trees. *J. ACM 32*, 3 (July 1985), 652–686.

[49] SMARAGDAKIS, Y., KAPLAN, S. F., AND WILSON, P. R. The EELRU adaptive replacement algorithm. *Perform. Eval. 53*, 2 (2003), 93–123.

[50] SMITH, A. Two methods for the efficient analysis of memory address trace data. *Software Engineering, IEEE Transactions on SE-3*, 1 (Jan 1977), 94–101.

[51] SNIA. SNIA iotta repository block I/O traces. http://iotta.snia.org/tracetypes/3.

[52] SUGUMAR, R. A., AND ABRAHAM, S. G. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 1993), SIGMETRICS '93, ACM, pp. 24–35.

[53] TAM, D. K., AZIMI, R., SOARES, L. B., AND STUMM, M. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2009), ASPLOS XIV, ACM, pp. 121–132.

[54] TANENBAUM, A. S. *Modern Operating Systems*, 3rd ed. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[55] TAY, Y. C., AND ZOU, M. A page fault equation for modeling the effect of memory size. *Perform. Eval. 63*, 2 (Feb. 2006), 99–130.

[56] THOMPSON, J. G. *Efficient Analysis of Caching Systems*. PhD thesis, EECS Department, University of California, Berkeley, Sep 1987.

[57] THOMPSON, J. G., AND SMITH, A. J. Efficient (stack) algorithms for analysis of writeback and sector memories. *ACM Trans. Comput. Syst. 7*, 1 (Jan. 1989), 78–117.

[58] TRAIGER, I., AND SLUTZ, D. *One-pass Techniques for the Evaluation of Memory Hierarchies*. IBM research report. IBM Research Division, 1971.

[59] VIANA, P., GORDON-ROSS, A., BARROS, E., AND VAHID, F. A table-based method for single-pass cache optimization. In *Proceedings of the 18th ACM Great Lakes Symposium on VLSI* (New York, NY, USA, 2008), GLSVLSI '08, ACM, pp. 71–76.

[60] VMWARE, INC. VMware vSphere Hypervisor (ESXi). http://www.vmware.com/products/vsphere-hypervisor/overview.html.

[61] WALDSPURGER, C. A. Memory resource management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (New York, NY, USA, 2002), OSDI '02, ACM, pp. 181–194.

[62] WALDSPURGER, C. A., VENKATASUBRAMANIAN, R., GARTHWAITE, A. T., AND BASKAKOV, Y. Efficient online construction of miss rate curves, Apr. 2014. U.S. Patent #8,694,728. Filed Nov. 2010. Assigned to VMware, Inc.

[63] WEST, R., ZAROO, P., WALDSPURGER, C. A., AND ZHANG, X. Online cache modeling for commodity multicore processors. *SIGOPS Oper. Syst. Rev. 44*, 4 (Dec. 2010), 19–29.

[64] WIRES, J. Private communication, Jan 2015.

[65] 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.

[66] XIE, F., CONDICT, M., AND SHETE, S. Estimating duplication by content-based sampling. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, 2013), USENIX, pp. 181–186.

[67] YANG, T., BERGER, E. D., KAPLAN, S. F., AND MOSS, J. E. B. CRAMM: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 103–116.

[68] YANG, T., HERTZ, M., BERGER, E. D., KAPLAN, S. F., AND MOSS, J. E. B. Automatic heap sizing: Taking real memory into account. In *Proceedings of the 4th International Symposium on Memory Management* (New York, NY, USA, 2004), ISMM '04, ACM, pp. 61–72.

[69] ZHAO, W., JIN, X., WANG, Z., WANG, X., LUO, Y., AND LI, X. Low cost working set size tracking. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2011), USENIXATC'11, USENIX Association, pp. 17–17.

[70] 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.

[71] ZHONG, Y., AND CHANG, W. Sampling-based program locality approximation. In *Proceedings of the 7th International Symposium on Memory Management* (New York, NY, USA, 2008), ISMM '08, ACM, pp. 91–100.

[72] ZHOU, P., PANDEY, V., SUNDARESAN, J., RAGHURAMAN, A., ZHOU, Y., AND KUMAR, S. Dynamic tracking of page miss ratio curve for memory management. *SIGOPS Oper. Syst. Rev. 38*, 5 (Oct. 2004), 177–188.