Characterizing Storage Workloads with Counter Stacks

Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew Warfield
Coho Data

Abstract

Existing techniques for identifying working set sizes based on miss ratio curves (MRCs) have large memory overheads which make them impractical for storage workloads. We present a novel data structure, the *counter stack*, which can produce approximate MRCs while using sublinear space. We show how counter stacks can be checkpointed to produce workload representations that are many orders of magnitude smaller than full traces, and we describe techniques for estimating MRCs of arbitrary workload combinations over arbitrary windows in time. Finally, we show how online analysis using counter stacks can provide valuable insight into live workloads.

1 Introduction

Caching is poorly understood. Despite being a pervasive element of computer system design – one that spans processor, storage system, operating system, and even application architecture – the effective sizing of memory tiers and the design of algorithms that place data within them remains an art of characterizing and approximating common case behaviors.

The design of hierarchical memories is complicated by two factors: First, the collection of live workload-specific data that might be analyzed to make "application aware" decisions is generally too expensive to be worthwhile. Approaches that model workloads to make placement decisions risk consuming the computational and memory resources that they are trying to preserve. As a result, systems in many domains have tended to use simple, general purpose algorithms such as LRU to manage cache placement. Second, attempting to perform offline analysis of access patterns suffers from the performance overheads imposed in trace collection, and the practical challenges of both privacy and sheer volume, in sharing and analyzing access traces.

Today, these problems are especially pronounced in designing enterprise storage systems. Flash memories are now available in three considerably different form factors: as SAS or SATA-attached solid state disks, as NVMe devices connected over the PCIe bus, and finally as flash-backed nonvolatile RAM, accessible over a DIMM interface. These three connectivity models all use the same underlying flash memory, but present performance and pricing that are pairwise 1-2 orders of magnitude apart. Further, in addition to solid-state memories, spinning disks remain an economical option for the storage of cold data.

This paper describes an approach to modeling, analyzing, and reasoning about memory access patterns that has been motivated through our experience in designing a hierarchical storage system [10] that combines these varying classes of storage media. The system is a scalable, network-attached storage system that can benefit from workload awareness in two ways: First, the system can manage allocation of the memory hierarchy in response to workload characteristics. Second, the capacity at each level of the hierarchy can be independently expanded to satisfy application demands, by adding additional hardware. Both of these properties require a more precise ability to understand and characterize individual storage workloads, and in particular their working set sizes over time.

Miss ratio curves (MRCs) are an effective tool for assessing working set sizes, but the space and time required to generate them make them impractical for large-scale storage workloads. We present a new data structure, the *counter stack*, which can generate approximate MRCs in sublinear space, for the first time making this type of analysis feasible in the storage domain.

Counter stacks use probabilistic counters [18] to estimate MRCs. The original approach to generating MRCs is based on the observation that a block's 'stack distance' (also known as its 'reuse distance') gives the capacity needed to cache it, and this distance is exactly the number of unique blocks accessed since the previous request for the block. The key idea behind counter stacks is that probabilistic counters can be used to efficiently estimate stack distances, allowing us to compute approximate MRCs at a fraction of the cost of traditional techniques.

Counter stacks are fast. Our Java implementation can process a week-long trace of 13 enterprise servers in 17 minutes using just 80 MB of RAM; at a rate of 2.3 million requests per second, the approach is practical for on-

line analysis in production systems. By comparison, a recent C implementation of a tree-based optimization [27] of Mattson's original stack algorithm [23] takes roughly an hour and 92 GB of RAM to process the same trace.

Our contributions in this paper are threefold. First, we introduce a novel technique for estimating miss ratio curves using counter stacks, and we evaluate the performance and accuracy of this technique. Second, we show how counter stacks can be periodically checkpointed and streamed to disk to provide a highly compressed representation of storage workloads. Counter stack *streams* capture important details that are discarded by statistical aggregation while at the same time requiring orders of magnitude less storage and processing overhead than full request traces; a counter stack stream of the compressed 2.9 GB trace mentioned above consumes just 11 MB. Third, we present techniques for working with multiple independent counter stacks to estimate miss ratio curves for new workload combinations. Our library implements *slice*, *shift*, and *join* operations, enabling the nearly-instantaneous computation of MRCs for arbitrary workload combinations over arbitrary windows in time. These capabilities extend the functionality of MRC analysis and provide valuable insight into live workloads, as we demonstrate with a number of case studies.

2 Background

The many reporting facilities embedded in the modern Linux storage stack [5, 7, 19, 25] are testament to the importance of being able to accurately characterize live workloads. Common characterizations typically fall into one of two categories: coarse-grain aggregate statistics and full request traces. While these representations have their uses, they can be problematic for a number of reasons: averages and histograms discard key temporal information; sampling is vulnerable to the often bursty and irregular nature of storage workloads; and full traces impose impractical storage and processing overheads. New representations are needed which preserve the important features of full traces while remaining manageable to collect, store, and query.

Working set theory [12] provides a useful abstraction for describing workloads more concisely, particularly with respect to how they will behave in hierarchical memory systems. In the original formulation, working sets were defined as the set of all pages accessed by a process over a given epoch. This was later refined by using LRU modelling to derive an MRC for a given workload and restricting the working set to only those pages that exhibit strong locality. Characterizing workloads in terms of the unique, 'hot' pages they access makes it easier to understand their individual hardware requirements, and has proven useful in CPU cache management for many years [21, 28, 35]. These concepts hold for storage workloads as well, but their application in this domain is challenging for two reasons.

First, until now it has been prohibitively expensive to calculate the working set of storage workloads due to their large sizes. Mattson's original stack algorithm [23] required $O(NM)$ time and $O(M)$ space for a trace of N requests and M unique elements. An optimization using a balanced tree to maintain stack distances [1] reduces the time complexity to $O(N \log M)$, and recent approximation techniques [14, 38] reduce the time complexity even further, but they still have $O(M)$ space overheads, making them impractical for storage workloads that may contain billions of unique blocks.

Second, the extended duration of storage workloads leads to subtleties when reasoning about their working sets. CPU workloads are relatively short-lived, and in many cases it is sufficient to consider their working sets over small time intervals (e.g., a scheduling quantum) [42]. Storage workloads, on the other hand, can span weeks or months and can change dramatically over time. MRCs at this scale can be tricky: if they include too little history they may fail to capture important recurring patterns, but if they include too much history they can significantly misrepresent recent behavior.

This phenomenon is further exacerbated by the fact that storage workloads already sit behind a file system cache and thus typically exhibit longer reuse distances than CPU workloads [43]. Consequently, cache misses in storage workloads may have a more pronounced effect on miss ratios than CPU cache misses, because subsequent reaccesses are likely to be absorbed by the file system cache rather than contributing to hits at the storage layer.

One implication of this is that MRC analysis needs to be performed over various time intervals to be effective in the storage domain. A workload's MRC over the past hour may differ dramatically from its MRC over the past day; both data points are useful, but neither provides a complete picture on its own.

This leads naturally to the notion of a *history of locality*: a workload representation which characterizes working sets as they change over time. Ideally, this representation contains enough information to produce MRCs over arbitrary ranges in time, in much the same way that full traces support statistical aggregation over arbitrary intervals. A naïve implementation could produce this representation by periodically instantiating new Mattson stacks at fixed intervals of a trace, thereby modelling independent LRU caches with various amounts of history, but such an ap-

proach would be impractical for real-world workloads.

In the following section we describe a novel technique for computing stack distances (and by extension, MRCs), from an inefficient, idealized form of counter stacks. Section 4 explains several optimizations which allow a practical counter stack implementation that requires sublinear space, and Section 5 presents the additional operations that counter stacks support, such as slicing and joining.

3 Counter Stacks

Counter stacks capture locality properties of a sequence of accesses within an address space. In the context of a storage system, accesses are typically read or write requests to physical disks, logical volumes, or individual files. A counter stack can process a sequence of requests as they occur in a live storage system, or it can process, in a single pass, a trace of a storage workload. The purpose of a counter stack is to represent specific characteristics of the stream of requests in a form that is efficient to compute and store, and that preserves enough information to characterize aspects of the workload, such as cache behaviour.

Rather than representing a trace as a sequence of requests for specific addresses, counter stacks maintain a list of counters, which are periodically instantiated while processing the trace. Each counter records the number of *unique* trace elements observed since the inception of that counter; this captures the size of the working set over the corresponding portion of the trace. Computing and storing samples of working set size, rather than a complete access trace, yields a very compact representation of the trace that nevertheless reveals several useful properties, such as the number of unique blocks requested, or the stack distances of all requests, or phase changes in the working set. These properties enable computation of MRCs over arbitrary portions of the trace. Furthermore, this approach supports composition and extraction operations, such as joining together multiple traces or slicing traces by time, while examining only the compact representation, not the original traces.

3.1 Definition

A counter stack is an in-memory data structure that is updated while processing a trace. At each time step, the counter stack can report a list of values giving the numbers of distinct blocks that were requested between the current time and *all previous* points in time. This data structure evolves over time, and it is convenient to display its history as a matrix, in which each column records the values reported by the counter stack at some point in time.

Formally, given a trace sequence $(e_1 \dots e_N)$, where e_i is the ith trace element, consider an $N \times N$ matrix C whose entry in the ith row and jth column is the number of distinct elements in the set $\{e_i \dots e_j\}$. For example, the trace (a, b, c, a) yields the following matrix.

(	a,	b,	c,	a,	)
	1	2	3	3	
		1	2	3	
			1	2	
				1	

The j^{th} column of this matrix gives the values reported by the counter stack at time step j, i.e., the numbers of distinct blocks that were requested between that time and all previous times. The i^{th} row of the matrix can be viewed as the sequence of values produced by the counter that was instantiated at time step i.

The in-memory counter stack only stores enough information to produce, at any point in time, a single column of the matrix. To compute our desired properties over arbitrary portions of the trace, we need to store the entire history of the data structure, i.e., the entire matrix. However, the history does not need be stored in memory. Instead, at each time step we write to disk the current column of values reported by the counter stack. This can be viewed as checkpointing, or incrementally updating, the on-disk representation of the matrix. This on-disk representation is called a *counter stack stream*; for conciseness we will typically refer to it simply as a *stream*.

3.2 LRU Stack Distances

Stack distances and MRCs have numerous applications in cache sizing [23], memory partitioning between processes or VMs [20,34,35,42], garbage collection frequency [39], program analysis [14,41], workload phase detection [31], etc. A significant obstacle to the widespread use of MRCs is the cost of computing them, particularly the high storage cost [4, 27, 30, 33, 40] – all existing methods require linear space. Counter stacks eliminate this obstacle by providinge xtremely efficient MRC computation while using sublinear space.

In this subsection we explain how stack distances, and hence MRCs, can be derived from counter stack streams. Recall that the stack distance of a given request is the number of distinct elements observed since the last reference to the requested element. Because a counter stack stores information about distinct elements, determining the stack distance is straightforward. At time step j one must find the last position in the trace, i, of the requested element, then examine entry C_{ij} of the matrix to determine the number of distinct elements requested between

times i and j. For example, let us consider the matrix given in Section 3.1. To determine the stack distance for the second reference to trace element a at position 4, whose previous reference was at position 1, we look up the value $C_{1,4}$ and get a stack distance of 3.

This straightforward method ignores a subtlety: how can one find the last position in the trace of the requested element? It turns out that this information is implicitly contained in the counter stack. To explain this, suppose that the counter that was instantiated at time i does not increase during the processing of element e_j. Since this counter reports the number of *distinct* elements that it has seen, we can infer that this counter has already seen element e_j. On the other hand, if the counter instantiated at time $i+1$ does increase while processing e_j, then we can infer that this counter has not yet seen element e_j. Combining those inferences, we can conclude that i is the position of last reference.

These observations lead to a finite-differencing scheme that can pinpoint the positions of last reference. At each time step, we must determine how much each counter increases during the processing of the current element of the trace. This is called the *intra-counter* change, and it is defined to be

$$\Delta x_{ij} \;=\; C_{i,j} - C_{i,j-1}$$

To pinpoint the position of last reference, we must find the newest counter that does not increase. This can be done by comparing the intra-counter change of adjacent counters. This difference is called the *inter-counter* change, and it is defined to be

$$\Delta y_{ij} \;=\; \begin{cases} \Delta x_{i+1,j} - \Delta x_{i,j} & \text{if } i < j \\ 0 & \text{if } i = j \end{cases}$$

Let us illustrate these definitions with an example. Restricting our focus to the first four elements of the example trace from Section 3.1, the matrices Δx and Δy are

{	a,	b,	c,	**a**	}
	1	1	1	0	
		1	1	1	
			1	1	
				1	

Δx

{	a,	b,	c,	**a**	}
	0	0	0	**1**	
		0	0	0	
			0	0	
				0	

Δy

Every column of Δy either contains only zeros, or contains a single 1. The former case occurs when the element requested in this column has never been requested before. In the latter case, if the single 1 appears in row i, then the last request for that element was at time i. For example, because $\Delta y_{14} = 1$, the last request for element a before time 4 was at time 1.

Determining the stack distance is now simple, as before. While processing column j of the stream, we infer that the last request for the element e_j occurred at time i by observing that $\Delta y_{ij} = 1$. The stack distance for the j^{th} request is the number of distinct elements that were requested between time i and time j, which is C_{ij}. Recall that the MRC at cache size x is the fraction of requests with stack distance exceeding x. Therefore given all the stack distances, we can easily compute the MRC.

4 Practical Counter Stacks

The idealized counter stack stream defined in Section 3 stores the entire matrix C, so it requires space that is quadratic in the length of the trace. This is actually *more* expensive than storing the original trace. In this section we introduce several ideas that allow us to dramatically reduce the space of counter stacks and streams.

Section 4.1 discusses the natural idea of decreasing the time resolution, i.e., keeping only every d^{th} row and column of the matrix C. Section 4.2 discusses the idea of pruning: eventually a counter may have observed the same set of elements as its adjacent counter, at which point maintaining both of them becomes unnecessary. Finally, Section 4.3 introduces the crucial idea of using probabilistic counters to efficiently and compactly estimate the number of distinct elements seen in the trace.

4.1 Downsampling

The simplest way to improve the space used by counter stacks and streams is to decrease the time resolution. This idea is not novel, and similar techniques have been used in previous work [16].

In our context, decreasing the time resolution amounts to keeping only a small submatrix of C that provides enough data, and of sufficient accuracy, to be useful for applications. For example, one could start a new counter only at every d^{th} position in the trace; this amounts to keeping only every d^{th} row of the matrix C. Next, one could update the counters only at every d^{th} position in the trace; this amounts to keeping only every d^{th} column of the matrix C. We call this process *downsampling*.

Adjacent entries in the original matrix C can differ only by 1, so adjacent entries in the downsampled matrix can differ only by d. Thus, any entry that is missing from the downsampled matrix can be estimated using nearby entries that are present, up to additive error d. For large-scale workloads with billions of distinct elements, even choosing a very large value of d has negligible impact on the estimated stack distances and MRCs.

Our implementation uses a slightly more elaborate form of downsampling because we wish to combine traces that may have activity bursts in disjoint time intervals and

avoid writing columns during idle periods. As well as starting a new counter and updating the old counters after every d^{th} request, we also start a new counter and update the old counters every s seconds with one exception: we do not output a column if the previous s seconds contain no activity. Our experiments reported in Section 7 pick $d = 10^6$ and $s \in \{60, 3600\}$.

4.2 Pruning

Recall that every row of the matrix contains a sequence of values reported by some counter. For any two adjacent counters, the older one (the upper row) will always emit values larger than or equal to the younger one (the lower row). Let us consider the difference of these counters. Initially, at the time the younger one is created, their difference is simply the number of distinct elements seen by the older counter so far. If any of these elements reappears in the trace, the older counter will not increase (as it has seen this element before), but the younger counter will increase, so the difference of the counters shrinks.

If at some point the younger counter has seen every element seen by the older counter, then their difference becomes zero and will remain zero forever. In this case, the younger counter provides no additional information, so it can be deleted. An extension of this idea is that, when the difference between the counters becomes sufficiently small, the younger counter provides negligible additional information. In this case, the younger counter can again be deleted, and its value can be approximated by referring to the older counter. We call this process *pruning*.

The simplest pruning strategy is to delete the younger counter whenever its value differs from its older neighbor by at most p. This strategy ensures that the number of active counters at any point in time is at most M/p. (Recall that M is the number of distinct blocks in the entire trace.) In our current implementation, in order to fix a set of parameters that work well across many workloads of varying sizes, we instead delete the younger counter whenever its value is at least $(1 - \delta)$ times the older counter's value. This ensures that the number of active counters is at most $O(\log(M)/\delta)$. Our experiments reported in Section 7 pick $\delta \in \{0.1, 0.02\}$.

4.3 Probabilistic Counters

Counter stack streams contain the number of distinct blocks seen in the trace between any two points in time (neglecting the effects of downsampling and pruning). The on-disk stream only needs to store this matrix of counts, as the examples in Section 3 suggested. The in-memory counter stack has a more difficult job – it must be able to update these counts while processing the trace, so each counter must keep an internal representation of the set of blocks it has seen.

The naïve approach is for each counter to represent this set explicitly, but this would require quadratic memory usage (again, neglecting downsampling and pruning). A slight improvement can be obtained through the use of Bloom filters [6], but for an acceptable error tolerance, the space would still be prohibitively large. Our approach is to use a tool, called a *probabilistic counter* or *cardinality estimator*, that was developed over the past thirty years in the streaming algorithms and database communities.

Probabilistic counters consume extremely little space and have guaranteed accuracy. The most practical of these is the HyperLogLog counter [18], which we use in our implementation. Each count appearing in our on-disk stream is not the true count of distinct blocks, but rather an estimate produced by a HyperLogLog counter which is correct up to multiplicative factor $1+\epsilon$. The memory usage of each HyperLogLog counter is roughly *logarithmic* in M, with more accurate counters requiring more space. More concretely, our evaluation discussed in Section 7 uses as little as 53 MB of memory to process traces containing over a hundred million requests and distinct blocks.

4.4 LRU Stack Distances

The technique in Section 3.2 for computing stack distances and MRCs using idealized counter stacks can be adapted to use practical counter stacks. The matrices Δx and Δy are defined as before, but are now based on the downsampled, pruned matrix containing probabilistic counts. Previously we asserted that every column of Δy is either all zeros or contains a single 1. This is no longer true. The entry Δy_{ij} now reports the *number* of requests since the counters were last updated whose stack distance was approximately C_{ij}.

To approximate the stack distances of all requests, we process all columns of the stream. As there may be many non-zero entries in the j^{th} column of Δy, we record Δy_{ij} occurrences of stack distance C_{ij} for every i. As before, given all stack distances we can compute the MRC.

An online version of this approach which does not emit streams can produce an MRC of guaranteed accuracy using provably sublinear memory. In a companion paper [15] we prove the following theorem. The key point is that the space depends polynomially on ℓ and ϵ, the parameters controlling the precision of the MRC, but only logarithmically on N, the length of the trace.

Theorem 1. *The online algorithm produces an estimated MRC that is correct to within additive error ϵ at cache sizes $\frac{1}{\ell}M, \frac{2}{\ell}M, \frac{3}{\ell}M, \ldots, M$ using only*

$O(\ell^2 \log(M) \log^2(N)/\epsilon^2)$ *bits of space, with high probability.*

5 The Counter Stack API

The previous two sections have given an abstract view of counter stacks. In this section we describe the system that we have implemented based on those ideas. The system is a flexible, memory-efficient library that can be used to process traces, produce counter stack streams, and perform queries on those streams. The workflow of applications that use this library is illustrated in Figure 1.

5.1 On-disk Streams

The on-disk streams output by the library are produced by periodically outputting a new column of the matrix. As discussed in Section 4, a new column is produced if either d requests have been observed in the trace or s seconds have elapsed (in the trace's time) since the last column was produced, except for idle periods, which are elided. Each column is written to disk in a sparse format to incorporate the fact that pruning may cause numerous entries to be missing.

In addition, the on-disk matrix C includes an extra row, called row R, which records the raw number of requests observed in the stream. That is, C_{Rj} contains the total number of requests processed at the time that the j^{th} column is output. Finally, the on-disk stream also records the trace's time of the current request.

5.2 Compute Queries

The counter stack library supports three computational queries on streams: *Request Count*, *Unique Request Count* and *MRC*.

The first two query operations are straightforward but useful, as we will show in Section 8.4. The Request Count query simply asks for the total number of requests that occur in the stream, which is C_{Rj} where j is the index of the last column. The Unique Request Count query is similar except that it asks for the total number of unique requests, which is C_{1j}.

The most complicated stream operation is the MRC query, which asks for the miss ratio curve of the given stream. This query is processed using the method described in Section 4.4.

5.3 Time Slicing and Shifting

It is often useful to analyze only a subset of a given trace within a specific time interval. We refer to this time-based selection as *slicing*. It is similarly useful when joining traces to alter the time signature by a constant time interval. We refer to this alteration as *shifting*.

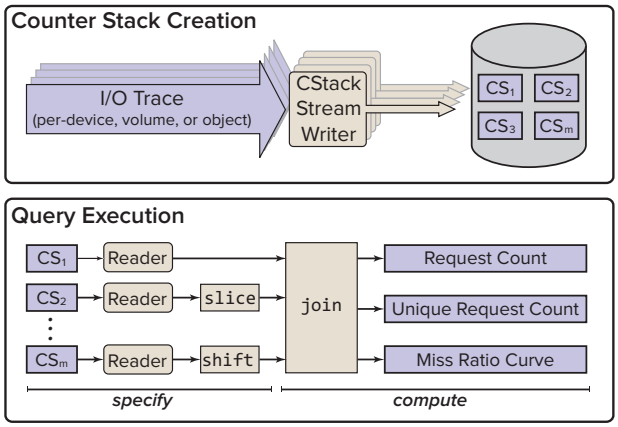

Figure 1: The counter stack library architecture.

The counter stack library supports slicing and shifting as specification operations. Given a stream containing a matrix C, the stream for the time slice between time step i and j is the submatrix with corners at C_{ii} and C_{jj}. Likewise, to obtain the stream for the trace shifted forward/backward s time units, we simply add/subtract s to each of the time indices associated with the rows and columns of the matrix.

5.4 Joining

Given two or more workloads, it is often useful to understand the behavior that would result if they were combined into a single workload. For example, if each workload is an I/O trace of a different process, one may want to investigate the cache performance of those processes with a shared LRU cache.

Counter stacks enable such analyses through the *join* operation. Given two counter stack streams, the desired output of the join operation is what one would obtain by merging the original two traces according to the traces' times, then producing a new counter stack stream from that merged trace. Our library can produce this new stream using only the two given streams, without examining the original traces. The only assumption we require is that the two streams must access disjoint sets of blocks.

The join process would be simple if, for every i, the time of the i^{th} request were the same in both traces; in this case, we could simply add the matrices stored in the two streams. Unfortunately that assumption is implausible, so more effort is required. The main ideas are to:

- *Expand* the two matrices so that each has a row and column for every time that appears in either trace.

time	1:00	1:02	1:05	1:14	1:17
A	a		b		b
$C_{\mathbf{A}}$	1	**1**	2	**2**	2
		0	**1**	**1**	**1**
			1	**1**	1
				0	**1**
					1
B		d		d	
$C_{\mathbf{B}}$	**0**	**1**	**1**	**1**	**1**
		1	**1**	1	**1**
			0	**1**	**1**
				1	**1**
					0
merge	a	d	b	d	b
$C_{\mathbf{A}} + C_{\mathbf{B}}$	1	2	3	3	3
		1	2	2	2
			1	2	2
				1	2
					1

Figure 2: An example illustrating the join operation.

- *Interpolate* to fill in the new matrix entries.
- *Add* the resulting matrices together.

Let us illustrate this process with an example. Consider a trace $\mathbf{A}$ that requests blocks (a, b, b) at times 1:00, 1:05, 1:17, and a trace $\mathbf{B}$ requests blocks (d, d) at times 1:02 and 1:14. The merge of the two traces is as follows:

time	1:00	1:02	1:05	1:14	1:17
A	a		b		b
B		d		d	
merge	a	d	b	d	b

To join these streams, we must expand the matrices in the two streams so that each has five rows and columns, corresponding to the five times that appear in the traces. After this expansion, each matrix is missing entries corresponding to times that were missing in its trace. We fill in those missing entries by an interpolation process: a missing row is filled by copying the nearest row beneath it, and a missing column is filled by copying the nearest column to the left of it. Figure 2 shows the resulting matrices; interpolated values are shown in bold blue.

Pruned counters can sometimes create negative values in Δx. For example, after pruning a counter in row j at time t, the interpolated value of the pruned counter at $t + 1$ is set to the nearest row beneath it, representing a younger counter. Often, this lower counter has a smaller value than the pruned counter. The interpolated value at $t+1$ will then be less than its previous value at t, producing a negative intra-counter change. We can avoid introducing negative values in Δx by replacing any negative values in Δx by the nearest nonnegative value beneath it. This replacement has the same effect of changing the value of the pruned counter to the lower counter in column t prior to calculating the intra-counter change for the column representing $t + 1$.

6 Error and Uncertainty

While each of the optimizations described in Section 4 dramatically reduce the storage requirements of counter stacks, they may also introduce uncertainty and error into the final calculations. In this section, we discuss potential sources of error, as well as how to modify the different operations described in Section 3 to compute lower and upper bounds on the stack distances.

6.1 Counter Error

HyperLogLog counters introduce error in two ways: count estimation and simultaneous register updates. HyperLogLog counters report a count of distinct elements that is only correct up to multiplicative factor ϵ, which is determined by a precision parameter. This uncertainty produces deviation from the true MRC and can be controlled by increasing the precision of the HyperLogLog counters, at the cost of a greater memory requirement.

Simultaneous register updates introduce a subtler form of error. A HyperLogLog counter estimates unique counts by taking the harmonic mean of a set of internal variables called *registers*. Due to the design of HLLs, sometimes a register update might cause the older counter to increase in value more than the younger counter. This phonemoneon leads to negative updates in Δy, because older counters are expected to change more slowly than younger counters. Theorem 1 implies that the negative entries in the Δy matrix introduced by simultaneous register updates are offset by corresponding over-estimates when register modifications between counters are not simultaneous.

In some cases, the histogram of stack distances may accumulate enough negative entries that there are bins with negative counts. The cumulative sum of such a histogram will result in a non-monotonic MRC. We can enforce a monotonic MRC by accumulating any negative histogram bins in a separate counter, carrying the difference forward in the cumulative sum and discounting positive bins by the negative count. In practice, negative histogram entries make up less then one percent of the reported stack distances, with little to no visible effect on the accumulated MRC.

6.2 Downsampling Uncertainty

Whereas the scheme of Section 3.2 computes stack distances exactly, the modified scheme of Section 4.4 only computes approximations. This uncertainty in the stack distances is caused by downsampling, pruning and use of probabilistic counters. To illustrate this, consider the example shown in Figure 3, and for simplicity let us ignore pruning and any probabilistic error.

At every time step j, the finite differencing scheme uses the matrix Δy to help estimate the stack distances for all requests that occurred since time step $j-1$. More concretely, if such a request increases the $(i+1)^{\text{th}}$ counter but does not increase the i^{th} counter, then we know that the most recent occurrence of the requested block lies somewhere between time step i and time step $i+1$. Since there may have been many requests between time i and time $i+1$, we do not have enough information to determine the stack distance exactly, but we estimate it up to additive error d (the downsampling factor). A careful analysis can show that the request must have stack distance at least $C_{i+1,j-1}+1$ and at most C_{ij}.

7 Evaluation

In this section we empirically validate two claims: (1) the time and space requirements of counter stack processing are sufficiently low that it can be used for online analysis of real storage workloads, and (2) the technique produces accurate, meaningful results.

We use a well-studied collection of storage traces released by Microsoft Research in Cambridge (MSR) [26] for much of our evaluation. The MSR traces record the disk activity (captured beneath the file system cache) of 13 servers with a combined total of 36 volumes. Notable workloads include a web proxy (`prxy`), a filer serving project directories (`proj`), a pair of source control servers (`src1` and `src2`), and a web server (`web`). The raw traces comprise 417 million records and consume just over 5 GB in compressed CSV format.

We compare our technique to the 'ground truth' obtained from full trace analysis (using *trace trees*, the tree-based optimization of Mattson's algorithm [23, 27]), and, where applicable, to a recent approximation technique [37] which derives estimated MRCs from *average footprints* (see Section 9 for more details). For fairness, we modify the original implementation [13] by using a sparse dictionary to reduce memory overhead.

7.1 Performance

The following experiments were conducted on a Dell PowerEdge R720 with two six-core Intel Xeon processors and 96 GB of RAM. Traces were read from high-performance flash to eliminate disk IO bottlenecks.

Fidelity	Time	Memory	Throughput	Storage
low	17.10 m	78.5 MB	2.31M reqs/sec	747 KB
high	17.24 m	80.6 MB	2.29M reqs/sec	11 MB

Table 1: The resources required to create low and high fidelity counter stacks for the combined MSR workload (64 MB heap).

Throughout this section we present figures for both 'low' and 'high' fidelity streams. We control the fidelity by adjusting the number of counters maintained in each stream; the parameters used in these experiments represent just two points of a wide spectrum, and were chosen in part to illustrate how accuracy can be traded for performance to meet individual needs.

We first report the resources required to convert a raw storage trace to a counter stack stream. The memory footprint for the conversion process is quite modest: converting the entire set of MSR traces to high-fidelity counter stacks can be done with about 80 MB of RAM [1]. The processing time is low as well: our Java implementation can convert a trace to a high-fidelity stream at a rate of 2.3 million requests per second with a 64 MB heap and 2.7 million requests per second with a 256 MB heap.

The size of counter stack streams can also be controlled by adjusting fidelity. Ignoring write requests, the full MSR workload consumes 2.9 GB in a compressed, binary format. We can reduce this to 854 MB by discarding latency values and capping timestamp resolutions at one second, and we can shave off another 50 MB through domain-specific compaction techniques like delta-encoding time and offset values. But as Table 1 shows, this is more than 70 times larger than a high-fidelity counter stack representation.

The compression achieved by counter stack streams is workload-dependent. High-fidelity streams of the MSR workloads are anywhere from 12 (`hm`) to 1,024 (`prxy`) times smaller than their compressed binary counterparts, with larger traces tending to compress better. A stream of the combined traces consumes just over 1.5 MB per day, meaning that weeks or even months of workload history can be retained at very reasonable storage costs.

Once a trace has been converted to a counter stack stream, performing queries is very quick. For example, an MRC for the entire week-long MSR trace can be com-

[1] This is not a lower bound. Additional reductions can be achieved at the expense of increased garbage collection activity in the JVM; for example, enforcing a heap limit of 32 MB increases processing time for the high-fidelity counter stack by about 30% and results in a peak resident set size of 53 MB.

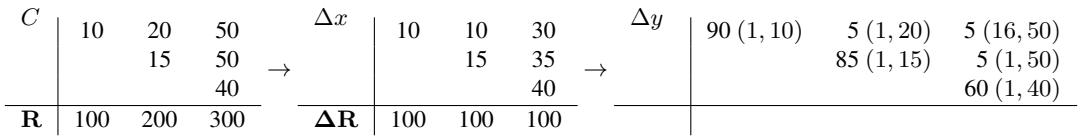

Figure 3: An example of computing stack distances using a downsampled matrix. The entries of Δy show the number of requests and the parenthesized values show the bounds on the stack distances that we can infer for those requests.

puted from the counter stack stream in just seconds, with negligible memory overheads. By comparison, computing the same MRC using a trace tree takes about an hour and reaches a peak memory consumption of 92 GB, while the average footprint technique requires 8 and a half minutes and 23 GB of RAM.

7.2 Accuracy

Figure 4 shows miss ratio curves for each of the individual workloads contained in the MSR traces as well as the combined `master` trace; superimposed on the baseline curves (showing the exact MRCs) are the curves computed using footprint averages and counter stacks. Some of the workloads feature MRCs that are notably different from the convex functions assumed in the past [35]. The `web` workload is the most obvious example of this, and it is also the workload which causes the most trouble for the average footprint technique.

Figure 5 shows three examples of MRCs produced by *joining* individual counter stacks. The choice of workloads is somewhat arbitrary; we elected to join workloads of commensurate size so that each would contribute equally to the resulting merged MRC. As described in Section 5.4, the join operation can introduce additional uncertainty due to the need to infer the values of missing counters, but the effects are not prominent with the high-fidelity counter stacks used in these examples.

We performed an analysis of curve errors at different fidelities, with `verylow` ($\delta = 0.46$, $d = 19M$, $s = 32K$) at one extreme and `high` ($\delta = 0.01$, $d = 1M$, $s = 60$) at the other. To measure curve error, we use the Mean Absolute Error (MAE) between a given curve and its ground-truth counterpart. The MAE is defined as the average absolute difference between two series mrc and mrc', or $\frac{1}{N}\sum|mrc(x) - mrc'(x)|$. Because MRCs range between 0 and 1, the MAEs are also confined to the same range, where a value of 0 implies perfectly corresponding curves. At the other extreme, it is difficult to know what constitutes a "bad" MAE because it is unlikely to be close to 1 except in singular cases. For example, the MAE between the `hm` and the `ts` Mattson curves is only 0.15. For the high fidelity counter stacks, we observe MAEs between 0.002 and 0.02, and for the average footprint algorithm, we observe MAEs between 0.001 and 0.04.

We find that curve error under compression is highly workload-dependent. We observed the largest errors on "jagged" workloads with sharp discontinuities, such as `src1` and `web`, while workloads with "flatter" MRCs such as `stg` and `usr` are almost invariant to compression. Figure 6 summarizes our findings on two such workloads. On the left, we illustrate the difference in the change in error as fidelity decreases for a jagged workload, `src1`, and a flat workload, `usr`. On the right, we show the smoothing effect of decreasing the counter stack fidelity by comparing the `verylow` and `high` fidelity curves against Mattson on `src1`.

8 Workload Analysis

We have shown that counter stacks can be used to produce accurate MRC estimations in a fraction of the time and space used by existing techniques. We now demonstrate some of the capabilities of the counter stack query interface through a series of case studies of the MSR traces.

8.1 Combined Workloads

Hit rates are often used to gauge the health of a storage system: high hit rates are considered a sign that a system is functioning properly, while poor hit rates suggest that tuning or configuration changes may be required. One problem with this simplistic view is that the combined hit rates of multiple independent workloads can be dominated by a single workload, thereby hiding potential problems.

We find this is indeed the case for the MSR traces. The `prxy` workload features a small working set and a high activity rate – it accesses only 2 GB of unique data over the entire week but issues 15% of all read requests in the combined trace. Table 2 puts this in perspective: the combined workload achieves a hit rate of 50% with a 550 GB cache; more than 250 GB of additional cache capacity would be required to achieve this same hit rate without the `prxy` workload. This illustrates why *combined hit rate is not an adequate metric of system behavior*. Diagnostic tools which present hit rates as an indicator of storage well-being should be careful to consider workloads independently as well as in combination.

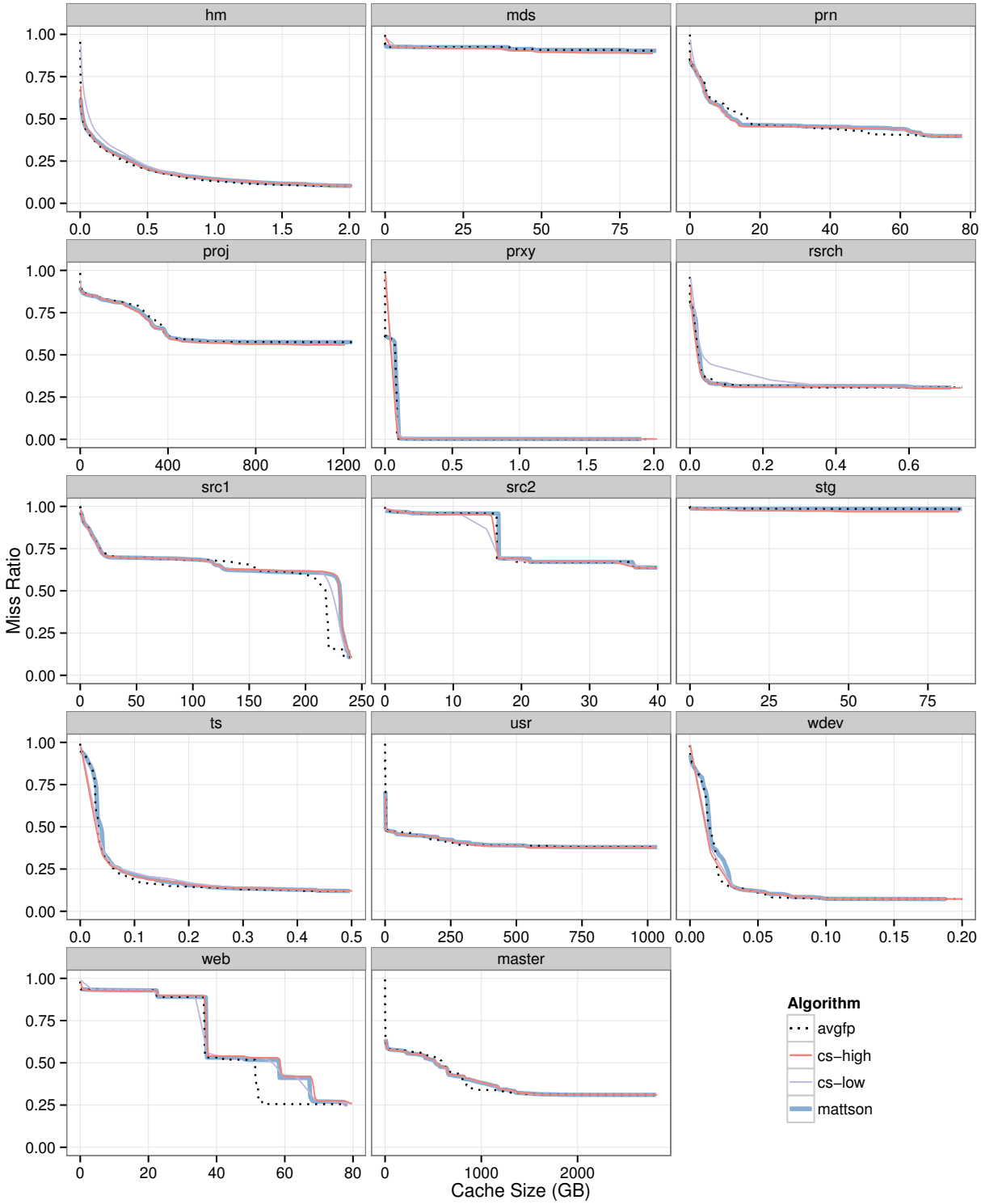

Figure 4: MSR miss ratio curves.

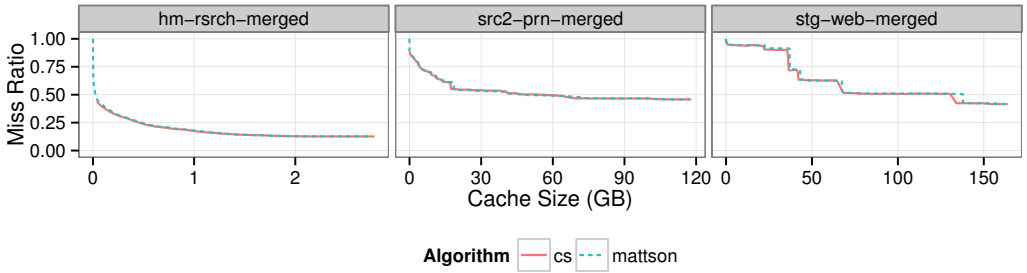

Figure 5: MRCs for various combinations of MSR workloads (produced by the *join* operation).

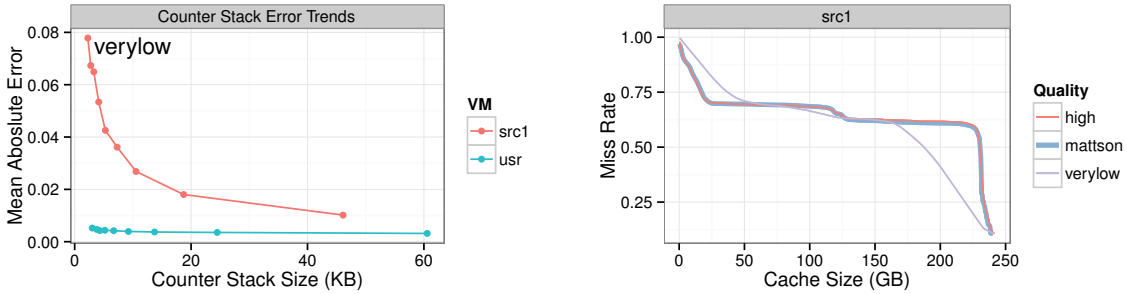

Figure 6: The qualitative effect of counter stack fidelity is workload-dependent. On the left, we show the curve error and file sizes of different fidelities. The `usr` workload is robust to compression to very low fidelity, while the `src1` workload degrades progressively. On the right, we show the visual outcome of compression to both *high* and *verylow* fidelity on `src1`.

Desired Hit Rate	Required Cache Size	
	With `prxy`	Without `prxy`
30%	2.5 GB	21.6 GB
40%	19.2 GB	525.5 GB
50%	566.6 GB	816.0 GB

Table 2: Cache sizes required to obtain desired hit rates for combined MSR workloads with and without `prxy`.

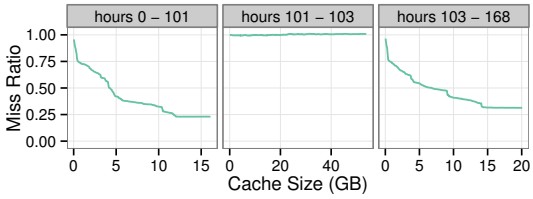

Figure 7: Time-sliced `prn` workload.

8.2 Erratic Workloads

MRCs can be very sensitive to anomalous events. A one-off bulk read in the middle of an otherwise cache-friendly workload can produce an MRC with high miss rates, arguably mischaracterizing the workload. We wrote a simple script that identifies erratic workloads by searching for hour-long slices with unusually high miss ratios. The script found several workloads, including `mds`, `stg`, `ts`, and `prn`, whose week-long MRCs are dominated by just a few hours of intense activity.

Figure 7 shows the effect these bursts can have on workload performance. The full-week MRC for `prn` (Figure 4) shows a maximum achievable hit rate of 60% at a cache size of 83 GB. The workload features a two-hour read burst starting 102 hours into the trace which accounts for 29% of the total requests and 69% of the unique blocks. Time-sliced MRCs before and after this burst feature hit rates of 60% at cache sizes of 10 GB and 12 GB, respectively. This is a clear example of how *anomalous events can significantly distort MRCs*, and it shows why it is important to consider MRCs over various intervals in time, especially for long-lived workloads.

8.3 Conflicting Workloads

Many real-world workloads exhibit pronounced diurnal patterns: interactive workloads typically reflect natu-

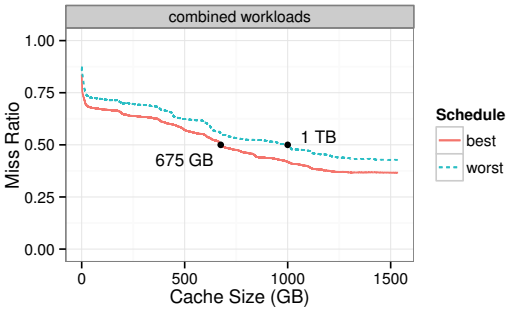

Figure 8: Best and worst time-shifted MRCs for MSR workloads (excluding `prxy`). We omit cache sizes greater than 1.5 TB to preserve details in the plot.

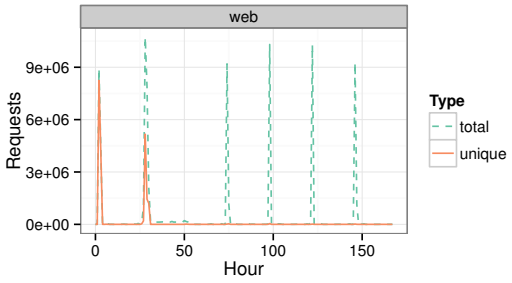

Figure 9: `web` total and unique requests per hour.

ral trends in business hours, while automatic workloads are often scheduled at regular intervals throughout the day [17, 22, 29]. When such workloads are served by the same shared storage, it makes sense to try to limit the degree to which they interfere with one another.

The time-shifting functionality of counter stacks provides a powerful tool for exploring coarse-grain scheduling of workloads. To demonstrate this, we wrote a script which computes the MRCs of the combined MSR trace (excluding `prxy`) in which the start times of a few of the larger workloads (`proj`, `src1`, and `usr`) are shifted by up to six hours. Figure 8 plots the best and worst MRCs computed by this script. As is evident, *workload scheduling can significantly affect hit rates*. In this case, shifting workloads by just a few hours changes the capacity needed for a 50% hit rate by almost 50%.

8.4 Periodic Workloads

MRCs are good at characterizing the raw capacity needed to accommodate a given working set, but they provide very little information about how that capacity is used over time. In environments where many workloads share a common cache, this lack of temporal information can be problematic. For example, as Figure 4 shows, the entire working set of `web` is less than 80 GB, and it exhibits a hit rate of 75% with a dedicated cache at this size. However, as shown in Figure 9, the workload is highly periodic and is idle for all but a few hours every day.

This behavior is characteristic of automated tasks like nightly backups and indexing jobs, and it can be problematic because *periodic workloads with long reuse distances tend to perform poorly in shared caches*. The cost of this is twofold: first, the periodic workloads exhibit low hit rates because their long reuse distances give them low priority in LRU caches; and second, they can penalize other workloads by repeatedly displacing 'hotter' data. This is exactly what happens to `web` in a cache shared with the rest of the MSR workloads: despite its modest working set size and high locality, it achieves a hit rate of just 7.5% in a 250 GB cache and 20% in a 500 GB cache.

Scan-resistant replacement policies like ARC [24] and CAR [3] offer one defense against this poor behavior by limiting the cache churn induced by periodic workloads. But a better approach might be to the exploit the highly regular nature of such workloads – assuming they can be identified – through intelligent prefetching. Counter stacks are well-suited for this task because they make it easy to detect periodic accesses to non-unique data. While this alone would not be sufficient to implement intelligent prefetching (because the counters do not indicate *which* blocks should be prefetched), it could be used to alert the system of the recurring pattern and initiate the capture of a more detailed trace for subsequent analysis.

8.5 Zipfian Workloads

We end with a brief discussion of synthetic workload generators like FIO [2] and IOMeter [32]. These tools are commonly used to test and validate storage systems. They are capable of generating IO workloads based on parameters describing, among other things, read/write mix, queue depth, request size, and sequentiality. The simpler among them support various combinations of random and sequential patterns; FIO recently added support for pareto and zipfian distributions, with the goal of better approximating real-world workloads.

Moving from uniform to zipfian distributions is a step in the right direction. Indeed, many of the MSR workloads, including `hm`, `mds`, and `prn`, exhibit roughly zipfian distributions. However, as is evident in Figure 4, the MRCs of these workloads vary dramatically. Figure 10 plots the MRC of a perfectly zipfian workload produced by FIO alongside two permutations of the same workload; as expected, request ordering has a significant impact on locality and cache behavior. These figures show that *syn-*

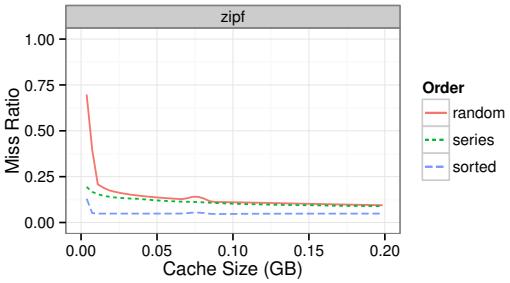

Figure 10: MRCs for three permutations of a single zipfian distribution: `random`, `series` (a concatenation of sorted series of unique requests), and `sorted` (truncated to preserve detail).

thetic zipfian workloads do not necessarily produce 'realistic' MRCs, emphasizing the importance of using real-world workloads when evaluating storage performance.

9 Related Work

Mattson et al. [23] defined stack distances and presented a simple $O(NM)$ time, $O(M)$ space algorithm to calculate them. Bennett and Kruskal [4] used a tree-based implementation to bring the runtime to $O(N \log(N))$. Almási et al. improved this to $O(N \log(M))$, and Niu et al. [27] introduced a parallel algorithm.

A different line of work explores techniques to efficiently approximate stack distances. Eklov and Hagersten [16] proposed a method to estimate stack distances based on sampling. Ding and Zhong [14] use an approximation technique inspired by the tree-based algorithms. Xiang et al. [37] define the footprint of a given trace window to be the number of distinct blocks occurring in the window. Using reuse distances, they estimate the average footprint across a logarithmic scale of window lengths. Xiang et al. [38] then develop a theory connecting the average footprint and the miss ratio, contingent on a regularity condition they call the *reuse-window hypothesis*. In comparison, counter stacks use dramatically less memory while producing MRCs with comparable accuracy.

A large body of work from the storage community explores methods for representing workloads concisely. Chen et al. [9] use machine learning techniques to extract workload features, Tarasov et al. [36] describe workloads with feature matrices, and Delimitrou et al. [11] model workloads with Markov Chains. These representations are largely incomparable to counter stacks – they capture many details that are not preserved in counter stack streams, but they discard much of the temporal information required to compute accurate MRCs.

Many domain-specific compression techniques have been proposed to reduce the cost of storing and processing workload traces. These date back to Smith's stack deletion [33] and include Burtscher's VPC compression algorithms [8]. They generally preserve more information than counter stacks but achieve lower compression ratios. They do not offer new techniques for MRC computation.

10 Conclusion

Sizing the tiers of a hierarchical memory system and managing data placment across them is a difficult, workload dependent problem. Techniques such as miss ratio curve estimation have existed for decades as a method of modeling workload behaviors offline, but their computational and memory overheads have prevented their incorporation as a means to make live decisions in real systems. Even as an offline tool, practical issues such as the overheads associated with trace collection and storage often prevent the sharing and analysis of memory access traces.

Counter stacks provide a powerful software tool to address these issues. They are a compact form of locality characterization that allow workloads to be studied in new interactive ways, for instance by searching for anomalies or shifting workloads to identify pathological load possibilities. They can also be incorporated directly into system design as a means of making more informed and workload-specific decisions about resource allocation across multiple tenants.

While the design and implementation of counter stacks described in this paper have been motivated through the design of an enterprise storage system, the techniques are relevant in other domains, such as processor architecture, where the analysis of working set size over time and across workloads is critical to the design of efficient, high-performance systems.

References

[1] G. S. Almási, C. Caşcaval, and D. A. Padua. Calculating stack distances efficiently. In *Proceedings of the 2002 workshop on memory system performance (MSP '02)*, pages 37–43, 2002.

[2] J. Axboe. Fio–flexible I/O tester, 2011.

[3] S. Bansal and D. S. Modha. CAR: Clock with adaptive replacement. In *FAST*, volume 4, pages 187–200, 2004.

[4] B. T. Bennett and V. J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19(4):353–357, 1975.

[5] M. Blaze. NFS tracing by passive network monitoring. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 333–343, 1992.

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[7] A. D. Brunelle. Block I/O Layer Tracing: blktrace. *HP, Gelato-Cupertino, CA, USA*, 2006.

[8] M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan, and N. B. Sam. The vpc trace-compression algorithms. *Computers, IEEE Transactions on*, 54(11):1329–1344, 2005.

[9] Y. Chen, K. Srinivasan, G. Goodson, and R. Katz. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 43–56. ACM, 2011.

[10] B. Cully, J. Wires, D. Meyer, K. Jamieson, K. Fraser, T. Deegan, D. Stodden, G. Lefebvre, D. Ferstay, and A. Warfield. Strata: scalable high-performance storage on virtualized non-volatile memory. In *Proceedings of the 12th USENIX conference on File and Storage Technologies*, pages 17–31. USENIX Association, 2014.

[11] C. Delimitrou, S. Sankar, K. Vaid, and C. Kozyrakis. Decoupling datacenter studies from access to large-scale applications: A modeling approach for storage workloads. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 51–60. IEEE, 2011.

[12] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[13] C. Ding. Program locality analysis tool. `https://github.com/dcompiler/loca`, 2014.

[14] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *PLDI*, pages 245–257. ACM, 2003.

[15] Z. Drudi, N. J. A. Harvey, S. Ingram, A. Warfield, and J. Wires. Approximating MRCs using counter stacks. Technical report, Coho Data, 2014.

[16] D. Eklov and E. Hagersten. StatStack: Efficient modeling of LRU caches. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 55–65. IEEE, 2010.

[17] D. Ellard, J. Ledlie, P. Malkani, and M. I. Seltzer. Passive NFS tracing of email and research workloads. In *FAST*. USENIX, 2003.

[18] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *DMTCS Proceedings*, 0(1), 2008.

[19] B. Jacob, P. Larson, B. Leitao, and S. da Silva. SystemTap: instrumenting the Linux kernel for analyzing performance and functional problems. *IBM Redbook*, 2008.

[20] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *ASPLOS*, pages 14–24. ACM, 2006.

[21] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122. IEEE Computer Society, 2004.

[22] A. W. Leung, S. Pasupathy, G. R. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX Annual Technical Conference*, pages 213–226, 2008.

[23] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[24] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[25] P. Mochel. The sysfs Filesystem. In *Linux Symposium*, page 313, 2005.

[26] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10, 2008.

[27] Q. Niu, J. Dinan, Q. Lu, and P. Sadayappan. Parda: A fast parallel reuse distance analysis algorithm. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1284–1294. IEEE, 2012.

[28] M. K. Qureshi and Y. N. Patt. 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*, pages 423–432. IEEE Computer Society, 2006.

[29] M. Shamma, D. T. Meyer, J. Wires, M. Ivanova, N. C. Hutchinson, and A. Warfield. Capo: Recapitulating storage for virtual desktops. In *FAST*, pages 31–45, 2011.

[30] X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality approximation using time. In *POPL*, pages 55–61. ACM, 2007.

[31] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *ASPLOS*, pages 165–176. ACM, 2004.

[32] J. Sievert. Iometer: The I/O performance analysis tool for servers, 2004.

[33] A. J. Smith. Two methods for the efficient analysis of memory address trace data. *Software Engineering, IEEE Transactions on*, 3(1):94–101, 1977.

[34] G. Soundararajan, D. Lupei, S. Ghanbari, A. D. Popescu, J. Chen, and C. Amza. Dynamic resource allocation for database servers running on virtual storage. In *FAST*. USENIX, 2009.

[35] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *Computers, IEEE Transactions on*, 41(9):1054–1068, 1992.

[36] V. Tarasov, S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok. Extracting flexible, replayable models from large block traces. *FAST*, 2012.

[37] X. Xiang, B. Bao, C. Ding, and Y. Gao. Linear-time modeling of program working set in shared cache. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 350–360. IEEE, 2011.

[38] X. Xiang, C. Ding, H. Luo, and B. Bao. HOTL: a higher order theory of locality. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 343–356. ACM, 2013.

[39] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *OSDI*, pages 103–116. ACM, 2006.

[40] W. Zhao, X. Jin, Z. Wang, X. Wang, Y. Luo, and X. Li. Low cost working set size tracking. In *Annual Technical Conference*, pages 223–228. USENIX, 2011.

[41] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *PLDI*, pages 255–266. ACM, 2004.

[42] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *ASPLOS*, pages 177–188. ACM, 2004.

[43] Y. Zhou, J. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *USENIX Annual Technical Conference, General Track*, pages 91–104, 2001.