Counter Stacks and the Elusive Working Set

JAKE WIRES, STEPHEN INGRAM, ZACHARY DRUDI, NICHOLAS J. A. HARVEY, AND ANDREW WARFIELD

Jake Wires is a principal software engineer at Coho Data and a doctoral candidate at the University of British Columbia. He is broadly interested in the design of storage systems and scalable data processing. jake@cohodata.com

Stephen Ingram is a software engineer at Coho Data. He received his PhD from the University of British Columbia in 2013, his MSc from UBC in 2008, and his BSc Honors degree in computer science from Georgia Tech in 2004. His research interests are information visualization and dimensionality reduction. stephen@cohodata.com

Zachary Drudi is a software engineer at Coho Data. He completed his MSc in computer science at the University of British Columbia. Zach is interested in placing streaming algorithms in containers. zach@cohodata.com

Nick Harvey is a consultant at Coho Data and an assistant professor at the University of British Columbia. His main research area is algorithm design. He completed his PhD in computer science at MIT in 2008. nick@cohodata.com

Andrew Warfield is co-founder and CTO of Coho Data and an associate professor at the University of British Columbia. He is broadly interested in software systems. andy@cohodata.com

Counter stacks are a compact and effective data structure for summarizing access patterns in memory and storage workloads. They are a stream abstraction that efficiently characterizes the *uniqueness of an access stream over time,* and are sufficiently low overhead as to allow both new approaches to online decision-making (such as replacement or prefetching policies) and new applications of lightweight trace transmission and archiving.

A fascinating shift is currently taking place in the composition of datacenter memory hierarchies. The advent of new, nonvolatile memories is resulting in larger tiers of fast random-access storage that are much closer to the performance characteristics of processor caches and RAM than they are to traditional bulk-capacity storage on spinning disks. There are two very important consequences to this trend:

The I/O gap is narrowing. Historically, systems designers have been faced with a vast and progressively widening gulf between the access latencies of RAM (~10 ns) and that of spinning disks (~10 ms). Storage-class memories (SCMs) are changing this by providing large, nonvolatile memories that are more similar to RAM than disk from a performance perspective.

Memory hierarchies are stratifying. SCMs are being built using different types of media, including different forms of NAND flash and also newer technologies such as Memristor and PCM. These memories also attach to the host over different interfaces, including traditional disk (SAS/SATA), PCIe/NVMe, and even the DIMM bus on which RAM itself is connected. These offerings have a diverse range of available capacities and performance levels, and a correlated range of prices. As a result, the memory hierarchy is likely to deepen as it becomes sensible to compose multiple types of SCM to balance performance and cost.

The result of these two changes is that there is now a greater burden on system designers to effectively design software to both determine the appropriate sizes and to manage data placement in hierarchical memories. This is especially true in storage systems, where the I/O gap has been especially profound: Fast memories used for caching data have historically been small, because they have been built entirely in RAM. As such, relatively simple heuristics (such as LRU and extensions such as ARC and CAR) could be used to keep a small and obvious set of hot data available for speedy access. The availability of larger fast memories, such as SCM-based caches, moves cached accesses farther out into the "tail" of the access distribution, where both sizing and prediction are much more formidable challenges. Put another way: a storage system has to work a lot harder to get value out of fast memories as it moves further into the tail of an access distribution.

We faced exactly these problems in the design of an enterprise storage system that attempts to balance performance and cost by composing a variety of memory types into a single coherent file system. One challenge we encountered early on involved understanding exactly how much high-performance storage is required to service a given workload. It turns out that while many storage administrators have a good understanding of the raw volume of data they're dealing with, they're often at a loss when it comes to predicting how much of that data is hot—and they lack the tools to find out.

The Elusive “Working Set”

The concept of a working set is well established within system design [1]. A working set is the subset of a program’s memory that is accessed over a period of execution. Working sets capture the concept of access locality and are the intuition behind the benefits of caching and hierarchical memories in general. A program’s working set is expected to shift over time as it moves between phases of execution or shifts to operate on different data, but the core intuition is that if a program can fit its working set entirely into fast memory, that it will run quickly and efficiently.

While the idea of a working set is relatively simple, it proves to be a very challenging characteristic to measure and model. One aspect of this is that working sets are very different depending on the period of time that they are considered over. A processor architect might consider working set phases to be the sort of thing that distill value from L1 or L2 caches: possibly megabytes of data over several thousand basic blocks of execution. In this domain, working sets may (and do) shift tens or hundreds of times a second. Conversely, a storage system may be concerned with workload characteristics that span minutes, hours, or even days of execution.

A second challenge in characterizing working sets is to measure them at all, at any range in time. Identifying working sets requires tracking the recency of access to addressable memory over time, which is generally both hard and expensive to do. One longstanding approach to this is Mattson’s stack algorithm, which is used to model hit ratio curves (also more pessimistically referred to as miss ratio curves in some of the literature) over an LRU replacement policy.

Mattson’s stack algorithm [4] is a simple technique that provides a really useful result: Given a stream of memory accesses over time, and assuming that those accesses are sent through a cache that is managed using an LRU replacement, Mattson’s algorithm can be run once over the entire trace and will report the hit rate at all sizes of the cache. The algorithm works by maintaining a stack of all addresses that have been accessed and an accompanying array of access counts at each possible depth within that stack. For each access in the stream, the associated address is located in the stack, the counter at that depth is incremented by one, and then that address is pulled to the front of the stack. At the end of the trace, the array is a histogram of reuse distances that directly reports the hit ratio curve. For progressively larger caches, it indicates the number of requests that would have hit in a cache of that size.

The hit ratio curves produced by Mattson (by plotting cache size on the x-axis and hit rate on the y-axis) are a useful way to identify working sets: Horizontal plateaus indicate a range of cache allocation that will not assist workload performance, while sudden jumps in hit rate indicate the edges of working sets, where a specified amount of cache is able to effectively serve a workload.

Unfortunately, calculating HRCs using Mattson is prohibitively expensive, in both time and space, for production systems. Even with optimizations that have been proposed over the decades that the technique has been studied, its memory consumption is linear with the amount of data being addressed, and lookups require log complexity over that set of addresses. This is far too heavyweight to perform at the granularity of every single access. The offline calculation of HRCs is similarly challenging because of the requirement that it carries for trace collection and storage: The resulting I/O traces are very large and challenging to ship to a central point of analysis.

So while modeling working sets has the potential to offer a great deal of insight into storage workloads, especially in regard to managing hierarchical memories, it is too expensive to run in production and so cannot be used for online decisions. Moreover, traces are prohibitively large to ship centrally, making it challenging for system designers to learn from and adapt products to customer workloads. To take full advantage of SCMs in the system, we wanted to achieve both of these things, and so needed a better approach to characterizing working sets.

Counter Stacks

The counter stack [5] is a data structure designed to provide an efficient measure of uniqueness over time. In the case of storage workloads, we are interested in measuring the number of unique block addresses accessed over a window of time. Mattson’s original algorithm (and its subsequent optimizations) measure this by tracking accesses to individual blocks, leading to high memory overheads. Counter stacks have much lower overheads because they do not bother recording accesses to individual addresses, but instead track only the *cardinality* of the accessed addresses. In other words, counter stacks measure *how many* unique blocks are accessed during a given interval, but they do not record the identity of those blocks. By making some relatively simple comparisons of the cardinalities observed over different intervals, we are able to compute approximate reuse distances and, by extension, miss ratio curves.

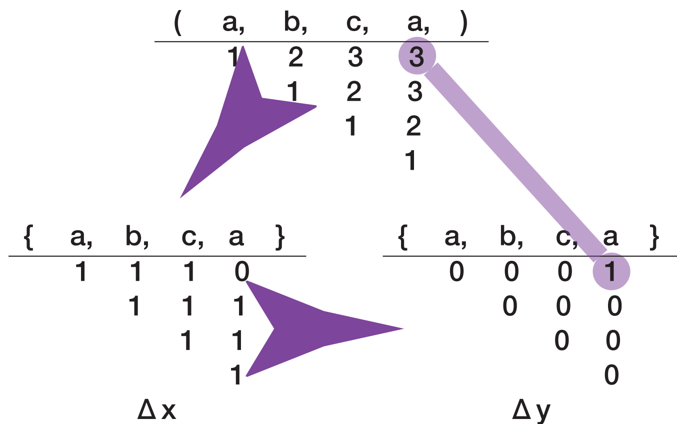

Figure 1: Using cardinality counters to characterize uniqueness over time

To see how this works, consider the sequence of requests for disk addresses {*a, b, c, a*} shown in Figure 1. Imagine that we instantiate a *cardinality counter* for each request we see. Cardinality counters support two operations: *update()* accepts arbitrary 64-bit values, and *count()* returns the total number of unique values passed to *update()*. Cardinality counters can be trivially implemented with a hash table; in practice, *probabilistic counters* like HyperLogLog (see sidebar) use approximation techniques to provide fairly accurate estimates with very low overheads.

In a counter stack, each cardinality counter records the number of unique addresses observed since that counter's inception. For each request, we update every existing counter and also instantiate a new one. If a request increases the value of a given counter, we know that the address has not been accessed at any time since the start of the counter; likewise, if the request does not increase a counter's value, we know that the address must have been previously accessed some time after the start of that counter.

This property makes it easy to pinpoint the logical time at which a requested address was last accessed. For every request, we iterate through the counters from youngest to oldest, updating each as we go. The first counter whose value does not change is necessarily the youngest counter old enough to have observed the last access to the address. Moreover, we know that the last access occurred at exactly the time that this counter was instantiated. If every counter's value changes for a given request, we know that address has never been observed before.

In the example from the diagram, the first matrix gives the values of the counters started for each request. Each row shows the sequence of values for a particular counter, and each column gives the values of the counters at a particular time. We can see that there are four requests for three unique addresses, and at the end of the sequence, each counter has a value of three or less, depending on how many requests it has observed.

We perform two transformations on the matrix to compute reuse distances. First, we calculate Δx, or the difference of each counter's value with its previous value. Each cell of Δx will have a value of 1 if the counter had not previously observed the request seen at that time, or 0 if it had. Then we calculate Δy, or the difference between adjacent rows of Δx. Each cell of Δy will have a value of 1 if and only if the corresponding request was not previously observed by the younger counter but was observed by the older counter.

A non-zero entry of Δy marks the presence of a repeated address in the request stream, and the row containing such an entry represents the youngest counter to have observed the previous access to the given address. We look up the cell's coordinates in the original matrix to obtain the cardinality value of the corresponding counter at the time of the repeated access, which gives us the number of unique addresses observed since the last access to the given address—in other words, the reuse distance of that address. Similar to Mattson's algorithm, we aggregate these reuse distances in a histogram, which directly gives a miss ratio curve.

Implementing counter stacks with a perfect counter (like a hash table) would be many orders of magnitude more expensive than Mattson's algorithm. Probabilistic counters go a long way towards making this approach feasible in practice, but at roughly 5 KB per counter, maintaining one per request for a workload with billions of requests is still prohibitively expensive. But as the diagram hints, the counter stack matrix is highly redundant and readily compressible.

We employ two additional lossy compression techniques to control the memory overheads of counter stacks. First, instead of maintaining a counter for every request in a workload, we only maintain counters for every kth request, and we only compute counter values after every kth request. This *downsampling* introduces uncertainty proportional to the value k. Second, we periodically *prune* requests as their values converge on those of their predecessors. Convergence occurs when younger counters eventually observe all the same values their predecessors have (it should be clear that counter values will never *diverge*). When the difference in the value of two adjacent counters falls below a pruning distance p, we can reap the younger counter since it provides little to no additional information.

These compression techniques are quite effective in practice, and they provide a means of lowering memory and storage overheads at the cost of reduced accuracy. In our experiments, we have observed that counter stacks require roughly 1200x less memory than Mattson's original algorithm while producing miss ratio curves with mean absolute errors comparable to other approximation techniques that have much higher overheads. Moreover, counter stacks are fast enough to use on the hot path in production deployments: we can process 2.3 million requests per second, compared to about 600,000 requests per second with a highly optimized implementation of Mattson's algorithm.

Probabilistic Counters

Probabilistic counters are a family of data structures that are used to approximate the number of distinct elements within a stream. HyperLogLogs [2] are a common example of such a probabilistic cardinality estimator and have been characterized as allowing cardinalities of over 109 elements to be estimated in a single streaming scan within 2% accuracy using only 1.5 KB of memory. As a result, these estimators are now being used in the implementation of network monitoring, data mining, and database systems [3]. Counter stacks [5] take advantage of HyperLogLogs to efficiently count cardinality in individual epochs during the request stream.

In many senses, HyperLogLogs are a data structure that is similar to, but more restrictive than, Bloom filters. An appropriately sized Bloom filter can provide an accurate hint as to whether or not a specific object has been inserted into it, but does not encode how many objects have been inserted. By simply adding an integer counter, Bloom filters can be extended to estimate cardinality. A HyperLogLog summarizes *just* the total cardinality of distinct objects, and cannot directly answer questions about whether a given object has been inserted. The result of sacrificing tests of membership is that HyperLogLogs can accurately estimate cardinality with much lower space requirements than would be needed to achieve the same precision using a Bloom filter-based counter.

A detailed explanation of how HyperLogLogs work would require more space than is available here, but the core intuition is relatively simple: If we were to consider a long series of coin tosses, one approach to approximate the total number of flips would be to observe the longest series of consecutive "heads" over the entire stream. Probabilistically, it will take much longer to have 10 heads in a row than it will to have 2; the longest string of heads provides a rough approximation of the total number of tosses. HyperLogLogs work similarly: They hash each element in a stream and then count the number of leading zero bits in the resulting hashed value. By aggregating counts of leading zeros into a set of independently sampled buckets, and then taking the harmonic mean across those resulting independent counts, HyperLogLogs are able to provide a very accurate (significantly better than the coin toss example above) and very compact approximation of the total cardinality.

Strictly speaking, only the last two columns of the counter matrix are needed to compute a miss ratio curve: The values produced by computing Δx and Δy can be incrementally aggregated into a histogram as the algorithm works through the sequence of requests, and older columns can be discarded. However, the matrix provides a convenient record of workload history, and, with a simple transformation (amounting in essence to a column index shift), it can be used to compute miss ratio curves over arbitrary sub-intervals of a given workload. This functionality turns out to be very expensive with traditional techniques for computing miss ratio curves, but it can be quite useful for tasks like identifying workload anomalies and phase changes.

In fact, we've found that counter stacks can help to answer a number of questions that extend beyond the original problems that led us to develop them. In particular, they provide an extremely concise format for preserving workload histories in the wild. We use counter stacks to record and transfer access patterns in production deployments at the cost of only a few MB per month; the next best compression technique we evaluated had a roughly 50x overhead. The ability to retain extended workload histories—and ship them back for easy analysis—is invaluable for diagnosing performance problems and understanding how our system is used in general, and it is enabling a new data-driven approach to designing placement algorithms. As we learn more about real-world workloads, we expect to augment counter stacks with additional metadata, thereby providing a richer representation of application behavior.

References

[1] Peter J. Denning, "The Working Set Model for Program Behavior," *Communications of the ACM*, vol. 11, no. 5 (May 1968), pp. 323–333.

[2] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "HyperLogLog: The Analysis of a Near-Optimal Cardinality Estimation Algorithm," in *Proceedings of the 2007 International Conference on Analysis of Algorithms* (DMTCS, 2007), pp. 127–146.

[3] S. Heule, M. Nunkesser, and A. Hall, "HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm," in *Proceedings of the 16th International Conference on Extending Database Technology (EDBT '13)* (ACM, 2013), pp. 683–692.

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

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