TTLs Matter: Efficient Cache Sizing with TTL-Aware Miss Ratio Curves and Working Set Sizes

Sari Sultan[1], Kia Shakiba[1], Albert Lee[1], Paul Chen[2], and Michael Stumm[1]
[1]University of Toronto, [2]Huawei
{sari.sultan,kia.shakiba,albee.lee}@mail.utoronto.ca,paul.chen1@huawei.com,stumm@eecg.toronto.edu

Abstract

In-memory caches play a pivotal role in optimizing distributed systems by significantly reducing query response times. Correctly sizing these caches is critical, especially considering that prominent organizations use terabytes and even petabytes of DRAM for these caches. The Miss Ratio Curve (MRC) and Working Set Size (WSS) are the most widely used tools for sizing these caches.

Modern cache workloads employ Time-to-Live (TTL) limits to define the lifespan of cached objects, a feature essential for ensuring data freshness and adhering to regulations like GDPR. Surprisingly, none of the existing MRC and WSS tools accommodate TTLs. Based on 28 real-world cache workloads that contain 113 billion accesses, we show that taking TTL limits into consideration allows an average of 69% (and up to 99%) lower memory footprint for in-memory caches without a degradation in the hit rate.

This paper describes how TTLs can be integrated into today's most important MRC generation and WSS estimation algorithms. We also describe how the widely used HyperLogLog (HLL) cardinality estimator can be extended to accommodate TTLs, and show how it can be used to efficiently estimate the WSS. Our extended algorithms maintain comparable performance levels to the original algorithms. All our extended approximate algorithms are efficient, run in constant space, and enable more resource-efficient and cost-effective cache management.

CCS Concepts: • **General and reference → Performance**; • **Computing methodologies → Simulation tools**.

Keywords: Time to Live (TTL), Miss Ratio Curve (MRC), Working Set Size (WSS), HyperLogLog (HLL), In-memory Caches, Key-Value Stores, Cache Sizing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroSys '24, April 22–25, 2024, Athens, Greece
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0437-6/24/04
https://doi.org/10.1145/3627703.3650066

ACM Reference Format:
Sari Sultan, Kia Shakiba, Albert Lee, Paul Chen, and Michael Stumm. 2024. TTLs Matter: Efficient Cache Sizing with TTL-Aware Miss Ratio Curves and Working Set Sizes. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3627703.3650066

1 Introduction

In-memory caches play a critical role in many distributed systems. They offload backend storage services and reduce query response times significantly by serving data out of the cache's DRAM instead of a backend storage service. Memcached [1, 2] and Redis [3] are two popular in-memory caches. All major cloud providers offer in-memory caching as a service [4–13].

A key challenge in operating in-memory caches is deciding how much physical memory to allocate when provisioning each cache. Too little memory results in higher miss rates and thus less efficient operation, with higher storage server loads and longer response times. Allocating too much memory results in unnecessarily increased capital and operational costs. These costs can be substantial [14]. As one point of reference, a single 100GB enterprise tier Redis cache from Microsoft Azure costs $8700 per month [15]. Numerous organizations use terabytes or even petabytes of cache memory, including Google, Meta, Twitter, Pinterest, Airbnb, GitHub, and LinkedIn [16–23].

In this paper we address the critical issue of sizing in-memory caches in modern cloud environments. Although this is a well-trodden problem for which many tools have been developed over the last five decades to aid in better understanding cache size tradeoffs [24–39], it is surprising that none of these tools are able to support modern workloads. More specifically, none of these tools are able to take ***Time-to-Live (TTL)*** attributes of cached objects into account, even though many modern workloads use TTLs to limit the lifespan of cached objects [23, 40–43]. This paper rectifies this situation for the most important state-of-the-art Miss Ratio Curve generation and Working Set Size estimation algorithms.

Miss Ratio Curve (MRC). The MRC is perhaps the most effective tool for evaluating cache size tradeoffs. It plots the cache miss ratio as a function of the cache size for a given workload under a specific eviction policy. For example,

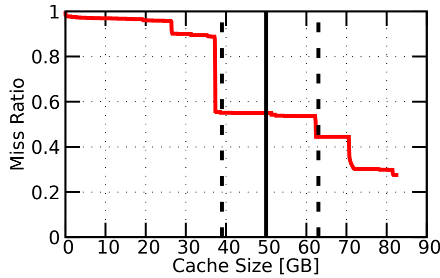

Figure 1. MRC for the web workload from MSR.

Fig. 1 shows the MRC for the "web" cache workload from Microsoft Research (MSR) [44] under the LRU eviction policy. Assuming the cache is configured with 50GB of memory, the MRC shows that the memory size can be reduced to 38GB with minimal effect on the miss ratio; alternatively it can be increased to 63GB for a 10% reduction in the miss ratio.

Mattson [24] and Olken [25] are two MRC-generation algorithms that produce exact MRCs. However, they are known to be computationally intensive and have large memory footprints, which makes them unsuitable for *online* MRC-generation [28, 29, 31]. Because of this, a number of algorithms have been introduced that generate *approximate* MRCs with significantly lower computation and memory overheads. Examples include Counterstacks [28], SHARDS [29], and AET [31]. Despite being approximate, these algorithms generally produce MRCs with acceptable errors; e.g., <2%.

Counterstacks stands out among the MRC-generation algorithms for pioneering the concept of *streaming*, in which portions of its internal state are periodically checkpointed (e.g., once per hour). As we outline further below, this offers a number of benefits: it enables more granular insight into the behavior of the cache as it processes its workload, and it enables understanding the effects of combining workloads.

Working Set Size (WSS). WSS is another important tool that aids in the management of caches. It refers to the aggregate size of all distinct objects accessed by a workload over a specified interval of time. The WSS identifies the minimum cache size needed to achieve the minimal miss rate. With TTLs, the WSS becomes the aggregate size of the *unexpired* distinct objects [23].

The WSS can be obtained in a number of ways. First, a hash table can be used to track objects accessed by the workload; the WSS is then the aggregate size of the distinct objects accessed, as recorded in the hash table. This approach requires memory proportional to the number of distinct objects accessed, which can be significant given that some workloads access billions of different objects.

Second, it is also possible to extract an estimate of the WSS from the workload's MRC: the point along the x-axis where the MRC first reaches its minimum miss ratio [26, 39, 45].

Finally, a WSS estimate can be obtained by using a cardinality estimator (a.k.a. F_0 estimator [46–54]) to identify the number of distinct objects accessed. A popular cardinality estimator is the HyperLogLog (HLL) counter [55, 56], also used by the CounterStacks MRC-generation algorithm. HLL is attractive because it is able to produce a WSS estimate using only a constant amount of space. Further, it enjoys the benefits of streaming as described further below.

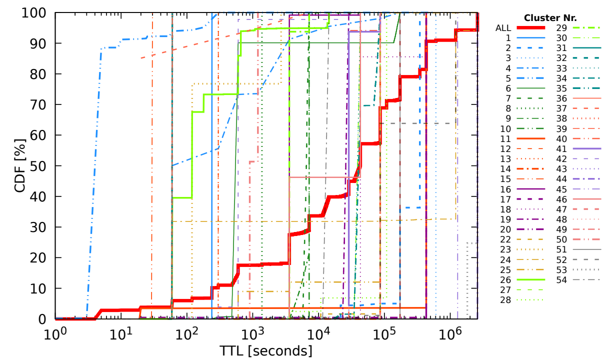

Figure 2. Cumulative distribution function (CDF) of the TTL limits for the workloads in the Twitter collection. The red curve: CDF of TTL limits across all Twitter workloads.

Time-to-Live (TTL) matters. For many modern workloads, TTLs play a critical role in cache management by providing a mechanism to expire cached objects based on their age [23, 40–43, 57–65]. TTLs are used, for example, to limit stale, inconsistent data in the cache or to implement General Data Protection Regulation (GDPR) mandated restrictions [23, 40, 41, 64]. In the Twitter cache workloads that have been made public [23], each cached object has an associated TTL attribute. Fig. 2 depicts the cumulative distribution of TTLs for each of the workloads from Twitter. The figure shows that the TTL distribution varies significantly for the different workloads. Overall, 27% of the cached objects expire in less than an hour, 50% of them expire in less than 12 hours, and 90% of them expire in less than 5 days.

TTL limits can significantly impact MRCs and WSSs. For example, Fig. 3 shows that neglecting TTLs when generating MRCs can result in MRCs that are substantially inaccurate. The figure shows two MRCs for Twitter's workload 50: one taking TTLs into account and the other not. The cache size needed to achieve the minimal miss rate is 7.3GB when TTLs are taken into account, but it increases to 123GB when they are not. Similarly, neglecting TTLs when generating WSSs can also be highly misleading. Fig. 4 shows that for Twitter's recommended workload 19 [66], the WSS never exceeds 5GB when taking TTLs into account, but reaches 40GB when neglecting TTLs.

Benefits of streaming. Streaming refers to the periodic saving of HLL counters [28]. It can be used with WSS estimators based on HLL counters and the CounterStacks MRC-generation algorithm which is also based on HLL counters. Streaming enables analysis of a given workload at a more

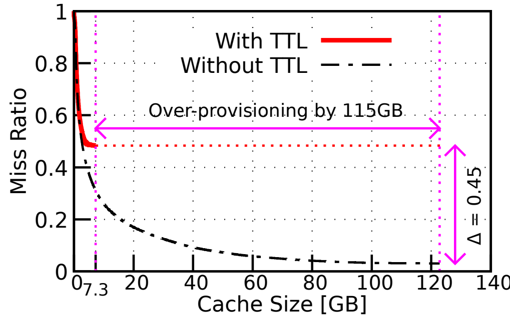

Figure 3. MRCs for the Twitter's workload 50: with TTL, it reaches steady-state at 7.3GB; without TTL at 123GB. This workload was recommended to users by Twitter on the discussion channel [66].

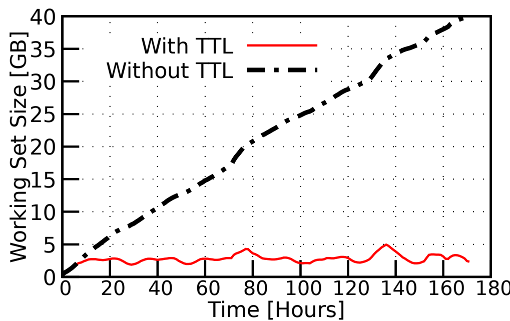

Figure 4. WSS for the Twitter workload 19 both with and without TTL. Each point at time t represents WSS from $[0, t)$.

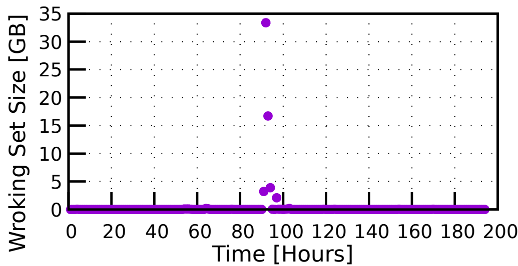

Figure 5. WSSs for the MSR src2 workload. Each point captures the WSS over a one hour time period. For most hours, the size of the WSS is under 200MB. In hours 91–97 the WSS of the workload is orders of magnitude larger.

granular level, and it enables analysis of the effects of combining multiple workloads to use a single cache.

As an example, consider Fig. 5 which depicts the WSS of the MSR src2 workload for each successive one hour period obtained using HLLs. The figure shows that the workload has outliers in hours 91-97 with significantly higher than normal WSSes. Understanding when exactly these outliers occur may help in identifying the cause. The outliers would not be apparent from the WSS of the entire workload.

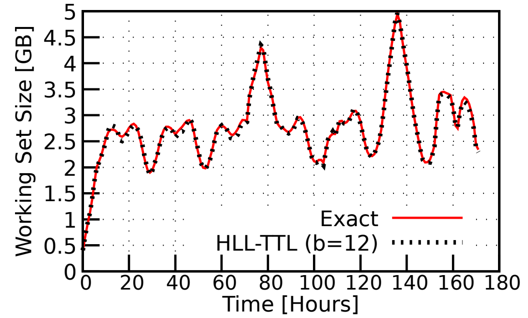

Figure 6. Twitter workload 19 WSS both with TTL. The solid line was obtained from exact WSS calculations using a hash table; each point at time t identifies the WSS over the interval $[0, t)$. The dotted curve was obtained using our extended HLL; each point at time t is the result of merging the individual one hour WSS estimates for hours 0 to t.

If these one hour WSSs are saved as HLL counters, then the HLL Merge operator (§3) can be used to combine any number of adjacent HLL counters to obtain the workload's WSS for the corresponding time interval. The effectiveness of this is demonstrated in Fig. 6 for Twitter workload 19 (when taking TTLs into account). One curve shows the WSS when calculated exactly using the hash table technique described earlier; the curve at point t represents the WSS over the time $[0, t)$. The other curve shows the WSS as obtained by combining the HLL-based WSS counters saved each hour from time 0 to time t. The two curves are for all intents and purposes indistinguishable.

Streaming can also be exploited to better understand caching patterns (e.g., diurnal patterns) using MRCs which in turn may help manage dynamic cache resizing [28]. For example, Fig. 7 shows the MRCs for three different publicly available workloads from MSR [44], IBM [67], and Twitter [23]. It shows that these workloads have substantially different MRCs for different 48 hour time windows. This behavior is not extractable from an MRC generated over the entire time period, but the MRC over the entire period becomes available by combining HLL counters streamed (e.g.) each hour by, say, CounterStacks.

Contributions. We have demonstrated in the previous discussion that TTLs matter for accurate MRC-generation and WSS estimation. In the remainder of the paper we show how the most important MRC-generation and WSS estimation algorithms can be adapted to take TTLs into account.

First, we show how Mattson and Olken, two exact MRC-generation algorithms, can be extended to account for TTLs (§2). These adaptations are straightforward; nevertheless, to the best of our knowledge, they have not been previously proposed. SHARDS generates approximate MRCs by using Olken on a sampled subset of cache accesses. SHARDS can similarly be extended using our extended Olken algorithm.

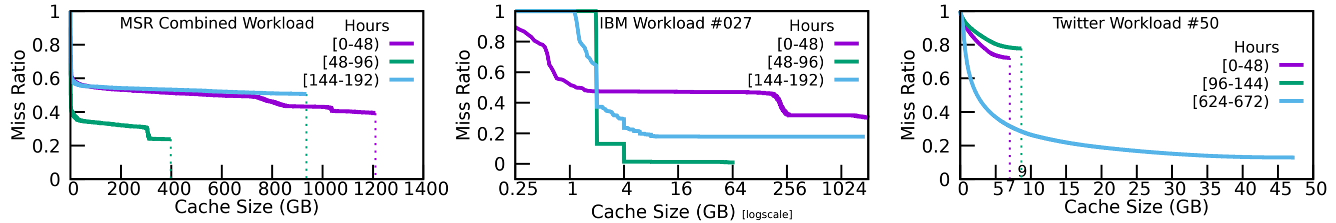

Figure 7. MRCs for 3 different publicly available workloads from MSR [44] (left), IBM [67] (middle), and Twitter [23] (right). Each figure shows the MRCs for 3 different 48 hour time periods to show how caching requirements change in these periods.

Second, we show how HLL counters can be extended to accommodate TTLs (§3). Our focus on extending HLLs is motivated by the fact that HLLs enable streaming for both MRC-generation and WSS estimation algorithms. The primary challenge in extending HLLs is the fact that HLLs (up to now) do not support deletes, a crucial operation needed to handle expired objects.

Third, we show how our extended HLLs can be used to extend the CounterStacks MRC-generation algorithm to accommodate TTLs (§4). We found that a straightforward integration of our extended HLL counters with CounterStacks led to a significant negative impact on accuracy. This issue arises because CounterStacks processes accesses in batches [28], which may result in inaccuracies when handling TTLs, as some accesses could expire within the batch before the last access's timestamp. To address this issue, we devised a new method for batch processing which terminates the batch prematurely whenever an object expires in the batch. This resulted in performance degradation due to the significant increase in the number of batches, which was further compounded by the extended HLL's larger memory footprint, leading to poorer cache locality. However, through crucial optimizations, we ultimately developed an algorithm that is far more efficient than CounterStacks and requires only a constant amount of space. As another application of our extended HLLs, we show how they can be utilized to estimate the WSS of TTL-endowed workloads in constant space (§5).

Finally, our experimental evaluation (§6) demonstrates the efficacy of our algorithms across 28 workloads from Twitter [23], totaling 113 billion accesses. The results indicate that accommodating TTLs lead to memory savings of 69% on average, and up to 99%. Our approximate algorithms achieve over 99% accuracy on average when incorporating TTLs. The throughput of our extended MRC-generation algorithms are comparable to their original counterparts, maintaining their performance despite the inclusion of TTLs.

2 Mattson^{++}, Olken^{++}, and Shards^{++}

In this section, we extend the seminal Mattson (§2.1), Olken (§2.2), and Shards (§2.3) MRC-generation algorithms to accommodate TTLs.

2.1 Mattson^{++}

Mattson introduced the first algorithm capable of generating an MRC in a single pass over a workload of cache accesses, assuming a compatible eviction policy such as LRU [24]. Prior to Mattson, generating an MRC required running a separate simulation for each different cache size on the MRC. For each access to an object in the workload, Mattson identifies the number of distinct objects accessed since the current object was last accessed. This number of distinct objects identifies the minimal cache size needed for the current object to remain in the cache; any smaller cache size would result in a miss and any larger cache size would result in a hit.

To identify the number of distinct objects accessed since the currently accessed object was last accessed, Mattson maintains a stack of distinct accessed objects ordered by access recency with the most recently accessed object at the top. While processing the workload, for each access, Mattson linearly searches the stack from the top for the currently accessed object. If found, the position in the stack, referred to as the *stack distance*, is recorded in a histogram of encountered stack distances, and the object is moved to the top of the stack as it is now the most recently accessed object. If the object is not found in the stack, then it is being accessed for the first time; in that case a stack distance of ∞ is added to the histogram and a new element representing the object is added to the top of the stack. After processing the entire workload, the MRC is constructed as the inverse Cumulative Distribution Function (CDF) of the histogram. For a workload with accesses to M distinct objects, space complexity is $\mathcal{O}(M)$ and each stack search has $\mathcal{O}(M)$ time complexity for $\mathcal{O}(NM)$ total time complexity with N accesses in the workload.

Mattson^{++} is a straightforward adaptation of Mattson's algorithm to accommodate TTLs. In Mattson^{++}, stack elements are extended to also record the eviction time of the accessed objects. While traversing the stack to determine the stack distance of an accessed object, any encountered object that has expired is removed and not considered in the stack distance calculation. Objects in the stack beyond the current stack distance need not be removed as they will be removed in a later operation. Mattson^{++} maintains the same time and space complexities as the original algorithm.

2.2 Olken^{++}

Olken optimized Mattson's algorithm by using a balanced binary search tree instead of a stack [25]. This brings the complexity of computing the stack distance down from $\mathcal{O}(M)$ to $\mathcal{O}(\log M)$, and overall from $\mathcal{O}(NM)$ to $\mathcal{O}(N \log M)$. Each node in the tree represents a distinct object and has a weight, defined as the total count of child nodes plus one for the node itself. Ordered by access recency, the tree simplifies counting the number of objects with a timestamp greater than the timestamp of the current access. The algorithm also maintains a hash table to track the last access time of each object, which is used to efficiently locate objects in the tree.

The computation of the stack distance for an accessed object is as follows. If the object is found in the hash table, then the timestamp obtained from the hash table is used to search for the object in the tree. The stack distance is then computed as the number of nodes in the tree with a timestamp larger than the obtained one, counted via the weights during the tree search. This stack distance is recorded in the histogram of encountered stack distances. The node is subsequently removed from the tree and reinserted with the current timestamp, and the hash table entry for the accessed object is updated to the current timestamp. If the object is not present in the hash table, then it is being accessed for the first time, so a new node with the current timestamp is added to the tree, a corresponding entry is added to the hash table, and a stack distance of ∞ is recorded in the histogram.

Olken^{++} extends Olken's algorithm to accommodate TTLs. The basic idea behind Olken^{++} is to track the expiry time of each object by maintaining a *priority queue*, *ET-PQ*, of ⟨`object, eviction time`⟩ tuples, ordered by eviction time. Then, on each access, all expired objects are first evicted before computing the stack distance. Olken^{++} maintains the same time and space complexities as the original algorithm.[1]

2.3 SHARDS^{++}

To reduce the overhead of exact MRC-generation algorithms (e.g., Olken), Waldspurger et al. introduced **S**patially **H**ashed **A**pproximate **R**euse **D**istance **S**ampling (SHARDS) [29]. SHARDS uses Olken to generate the MRC, but considers only a sampled subset of the total workload. Access i in the workload is sampled if the hash of the object's key H_i modulo P (a constant typically set to 2^{24}) is less than a threshold T; i.e., if $H_i \% P < T$. T is used to control the sampling rate $R = T/P$. This method of *spatial sampling* has the attractive property that if an access to an object is sampled, then all accesses to the same object will be sampled.

[1]We note that recording the eviction times in the tree nodes and evicting expired objects during tree traversal, similar to the strategy in Mattson^{++}, is inefficient and would increase the compute complexity of the algorithm from $\mathcal{O}(N \log M)$ to $\mathcal{O}(NM)$. This is because all nodes that might affect the stack distance of the current accessed object would have to be tested to determine whether they have expired. For example, if the accessed object is in the left sub-tree, then the entire right sub-tree would have to be traversed.

The sampling makes SHARDS an approximate algorithm, but generates surprisingly accurate MRCs for most workloads, even with a sampling rate of $R = 0.001 (= 0.1\%)$ [29]. Two variants of Shards have been proposed: **fixed-rate FR-SHARDS** which maintains a constant sampling rate and **fixed-size FS-SHARDS** which adjusts the sampling rate down to keep the number of sampled objects below a specified constant; Waldspurger et al. further introduced an extension to the two variants which adjusts the generated MRCs to address sampling biases: **FR-SHARDS**$_{adj}$ and **FS-SHARDS**$_{adj}$.

FR-SHARDS^{++}**.** FR-SHARDS uses a fixed sampling rate R when processing the workload to generate the MRC. To accommodate TTLs, FR-SHARDS can use Olken^{++} instead of Olken with no other changes required. We refer to this variant as FR-SHARDS^{++}. It has the same time and space complexity as FR-SHARDS, namely $\mathcal{O}(N \log M)$ and $\mathcal{O}(M)$, with compute and memory overheads reduced by a factor of $1/R$ compared to Olken^{++}.

FS-SHARDS^{++}**.** FS-SHARDS samples accesses to objects such that the number of distinct sampled objects does not exceed a constant S_{max}. Waldspurger et al. showed that with $S_{max} = 8K$, reasonably accurate MRCs can be generated for most workloads [29]. This variant achieves $\mathcal{O}(1)$ space overhead and $\mathcal{O}(N)$ compute overhead, making it the most efficient known MRC-generation algorithm as we will show in the evaluation section. The algorithm begins with a high sampling rate (typically $R = 0.1$) and reduces it monotonically to prevent sampling more than S_{max} distinct objects. For each object sampled for the first time, the object's sampling factor $F_i = H_i \% P$ is recorded in a priority queue, *F-PQ*, of ⟨`object,` F_i⟩ tuples, ordered by the sampling factor. Once the number of sampled objects is about to exceed S_{max}, the object with the largest sampling factor, F_{max}, is removed from both F-PQ and Olken's data structures and the algorithm's sampling threshold T is lowered to F_{max}.

To accommodate TTLs, FS-SHARDS^{++} uses Olken^{++}, as FR-SHARDS^{++} does, but with the following modifications. When an object is removed from F-PQ it must also be removed from Olken^{++}'s expiry time priority queue, ET-PQ (§2.2). Similarly, when an object expires from ET-PQ, it should be removed from F-PQ. One way to implement this is to add two new fields to Olken^{++}'s hash table of objects: a pointer to the object in F-PQ and a pointer to the object in ET-PQ. Thus, when an object is removed from F-PQ, it can efficiently be removed from ET-PQ, and vice versa. FS-SHARDS^{++} maintains the same complexities as the original algorithm.

FR-SHARDS$^{++}_{adj}$ ***and FS-SHARDS***$^{++}_{adj}$**.** Sampling bias is a known downside of SHARDS, as it may not sample frequently accessed objects, leading to MRCs with high errors [29]. For example, we observed that FR-SHARDS performed poorly on most workloads from SEC EDGAR [68, 69], with a Mean Absolute Error (MAE) of 12%, on average, even with a high sampling rate of $R = 0.1$ (=10%). This poor accuracy stems

from SHARDS not distinguishing between highly popular and less popular objects. In the case of the SEC workloads, SHARDS did not sample any of the 3 most frequently accessed objects which account for 47% of all the accesses.

To mitigate this issue, Waldspurger et al. proposed an extension to SHARDS called SHARDS$_{adj}$ [29]. This modification estimates the number of accesses expected to be sampled for a given sampling rate, and then adjusts the first bucket in the stack distance histogram by the difference between the expected and actual number of sampled accesses. The modification is based on the assumption that the difference between the expected and actual number of sampled accesses is primarily due to not sampling frequently accessed objects, and that most accesses for these popular unsampled objects would result in hits at relatively small stack distances. Increasing the frequency of the first bucket in the stack distance histogram addresses this issue. With the SEC workloads, we found this adjustment reduces the MAE to less than 2%.

The adjustment can easily be incorporated into FR-SHARDS^{++} and FS-SHARDS^{++} which we refer to as FR-SHARDS$^{++}_{adj}$ and FS-SHARDS$^{++}_{adj}$, respectively.

3 Extending HyperLogLog

HLL is a cardinality estimation algorithm that efficiently approximates the number of distinct elements in a multiset [55, 56]. It is one of the most efficient methods to estimate the WSS, which can be used to size caches. For our particular application, the multiset being considered contains one element for each access in the target workload. More specifically, each element is a hash of the key used to access an object in the cache. Below, we first briefly give a high-level overview of the HLL algorithm, and then show how HLLs can be extended to accommodate TTLs.

3.1 HLL Background

Assuming multiset $\mathcal{M}$ contains 64-bit integers, the algorithm identifies the number of leading zeros NLZ(x) in the binary representation of each $x \in \mathcal{M}$. If the maximum NLZ is $n = \max_{x \in \mathcal{M}} \text{NLZ}(x)$ then the algorithm estimates that $\mathcal{M}$'s cardinality is $\alpha \cdot 2^{n+1}$, where α is a constant fudge factor.

To improve accuracy, the algorithm actually first partitions the elements of $\mathcal{M}$ into 2^b buckets. The b-bit prefix of x, $P(x)$, is used to identify which bucket x belongs to. Each bucket separately tracks the maximum NLZ of the $(64-b)$ bit suffix $S(x)$ of each x assigned to the bucket; these maxima are maintained in a bucket array, which we denote `HLL[0 : 2^b − 1]`. See Fig. 8 (a). The overall count estimate is then $\alpha \cdot 2^{\overline{n+1}}$ where $\overline{n+1}$ is the harmonic mean of the bucket maxima in the bucket array `HLL[0 : 2^b − 1]`.

The estimation error of an HLL counter is $1.04/\sqrt{2^b}$, so b is effectively a precision parameter [55]; for example $b = 12$ provides over 98% accuracy in practice. The space used per bucket is typically 6 bits, so an HLL counter (in its entirety) requires $(2^b \cdot 6)$ bits (using a 64-bit hash function); that is, with $b = 12$, 3KB of space is needed for the bucket array.[2]

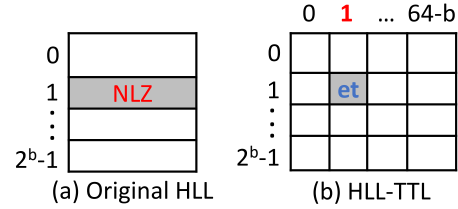

Figure 8. HLL & HLL-TTL bucket arrays (et: eviction time).

There are three main operations on HLL counters: `Insert`, `Count`, and `Merge`. Their implementations are surprisingly simple and shown in Fig. 9. `Insert` updates an HLL to reflect a new element x being added to a multiset. `Count` returns the count estimate. `Merge` generates an HLL to reflect the number of distinct objects in the union of two multisets, $\mathcal{M}_1 \cup \mathcal{M}_2$, given their respective HLLs. It is the `Merge` operation that makes streaming so powerful.

3.2 TTL Support with HLL-TTL

The original HLL does not provide the functionality to delete expired objects. We found that extending HLLs to support deletion of expired objects to be non-trivial. The core idea underlying our approach is to exclude the expired objects from the counting process, effectively treating them as deleted.

To accommodate TTLs, the basic HLL 1-dimensional array of buckets shown in Fig. 8 (a) is extended to a 2-dimensional matrix as shown in Fig. 8 (b). The size of the array is selected as follows. The number of rows is set to 2^b, the same as the number of buckets in the original algorithm. The number of columns is set to $64 - b$. The elements of $\mathcal{M}$ are also partitioned as before, but in this case into $2^b \times (64 - b)$ buckets. The first b bits of x, $P(x)$, are used to index into a bucket row, and the NLZ of x's suffix, $S(x)$, is used to index into a bucket column. The column index efficiently encodes the NLZ, and hence, there is no need to store the maximum NLZ values explicitly. Instead, each bucket is used to store the largest *eviction time* seen while updating the bucket.

We now consider the operations of the extended HLL, we call **HLL-TTL**. Their implementations are shown in Fig 10. When a new object x is added to the multiset $\mathcal{M}$, the most significant b bits of x, $P(x)$, are used to index into a row of the HLL-TTL matrix. The NLZ of x's least significant $64 - b$ bits, $S(x)$, is then used to index into an HLL-TTL column to identify a target bucket. If the eviction time of x is larger than the recorded value in the bucket, then the bucket's eviction time is updated to the larger value.

[2] In practice, using a byte instead of 6 bits per bucket makes the implementation easier, but increases the space required from $3KB$ to $4KB$. In our implementation, we use Heule et al.'s HLL++ implementation [56], which supports 64-bit hashes and has a sparse implementation. Throughout the paper, we refer to Heule's HLL++ as HLL.

Insert(x, HLL): update HLL to reflect x added to multiset
`HLL[P(x)] = max{HLL[P(x)], NLZ(S(x))}` where
`P(x)`=first b bits of x; `S(x)`=last $64-b$ bits of x.

Count(HLL): returns counter estimate
return $\alpha \cdot 2^{\overline{n+1}}$ where $\overline{n+1}$ = harmonic mean of HLL$[0..2^b-1]$
and α is a constant.

Merge(HLL_1, HLL_2) $\rightarrow$ HLL: merge HLL_1 and HLL_2
`for i=0..`2^b-1 `HLL[i] = max{HLL`$_1$`[i], HLL`$_2$`[i]}`

Figure 9. HLL operations

Insert(x, et,HLL): update HLL to reflect x added to multiset
with eviction time et
`HLL[P(x),NLZ(S(x))] = max{HLL[P(x), NLZ(S(x))],` et`}`

Count(HLL): returns counter estimate
return $\alpha \cdot 2^{\overline{n+1}}$ where $\overline{n+1}$ = harmonic mean of $n_0..n_{2^b-1}$
where α is a constant and
$n_i = \max\{j \in [0, 64-b]\}$: HLL$[i,j] \neq 0$ and not expired

Merge(HLL_1,HLL_2) $\rightarrow$ HLL: Merge HLL_1 and HLL_2
`for i=0..`2^b`-1 for j=0..64-`b
`HLL[i,j] = max(HLL`$_1$`[i,j],HLL`$_2$`[i,j])`

Figure 10. HLL-TTL operations

Cardinality estimation (`Count`) is now based on the harmonic mean of the column indices corresponding to the rightmost bucket in each row with a recorded expiry time that is not 0 and has not yet expired.[3] That is, given that NLZ increases as one moves right in the columns, we scan each row from right to left, and identify the rightmost bucket with a non-expired value; the column index of that bucket is used to compute the harmonic mean. The fundamental concept behind this approach is that the largest NLZ will be utilized for cardinality estimation, similar to the original HLL method. We note that when the current largest NLZ in a row expires, then the next largest NLZ that has not yet expired will be used.

The implementation of `Merge` is analogous to the one for the basic HLL. Merging two HLL-TTL counters, H_1 and H_2, results in an HLL-TTL counter such that the eviction time of each bucket is equal to the larger eviction time from H_1 and H_2 for the same index.

Performance considerations. Adding a second dimension to the HLL increases space usage by a large constant factor. Assuming the eviction time can be encoded as a 32-bit integer, then the extended version increases the space required from $2^b \times 6$ bits to $2^b \times (64-b) \times 32$ bits. For precision $b = 12$, the space requirement increases from $4KB$ to $832KB$. Henceforth, we will refer to this as the *dense implementation.*

Alternatively, a *dynamic implementation* utilizes a linked list similar to the approach proposed by Heule et al. to save space [56]. Each row is replaced with a linked list, allocating list elements as needed.[4] Each elements contains the tuple ⟨`NLZ, eviction time`⟩ and the list is ordered by NLZ. Whenever an element has an eviction time less than that of the next element in the list, it can be removed because it is guaranteed not to ever contribute to the results `Count ()`. While this reduces space usage in practice, the downside is that it incurs extra processing overhead for allocating and freeing list elements as well as for traversing the linked lists. In our evaluation over all workloads we considered, space usage is reduced by over 80%, on average, compared to the dense implementation, but throughput decreased by a factor of two.

[3]If every entry in a row equals zero (or if all entries have expired), then this row will not make any contribution to the harmonic mean. Rather, the count of rows where all entries are zero (or all have expired) will be used to make bias corrections, following the same procedure as detailed in the original HLL paper [55].

Table 1. Comparison of HLL-TTL Implementations

HLL precision	8	10	12	14	16
Space usage: Dynamic relative to Dense	80%	82%	83%	85%	88%
Slowdown: Dynamic relative to Dense (×)	1.83	1.96	2.25	2.43	3.03
Overhead converting Dynamic to Dense (ms)	0.1	0.2	1.5	4.2	36
Overhead serializing Dense impl. (ms)	0.2	0.7	3.0	14	38
Overhead serializing Dynamic impl. (ms)	0.1	0.5	1.7	5.8	23

In our practical implementation, we use a three-pronged approach. First, we use a *sparse implementation* based on a dynamically-sized closed hash table with one entry for each unique key countered. Each entry contains a ⟨`Hash(key)`, `expiry time`⟩ tuple. This is similar to Microsoft's HLL implementation's *direct counting* approach [70]. Second, when the size of the hash table reaches the size of the 2D array in the dense implementation, the sparse implementation is converted to the dense implementation. Finally, whenever the HLL needs to be stored to disk or sent over a communication channel (for streaming), then the dense implementation is converted to the dynamic implementation with the linked lists and marshalled.

The performance of the dynamic implementation is lower than that of the dense implementation. However, the dynamic implementation uses significantly less memory, because the dense implementation is a 2D matrix where many of the cells can be zeros. Table 1 compares the performance and space usage of the two implementations across precision levels 8 through 16, and it shows the overheads of converting a dynamic representation to a dense representation as well as the serialization overheads for the two representations.

4 CounterStacks^{++}

In this section, we provide background on Counterstacks (CS)(§4.1), extend CS to accommodate TTLs (§4.2), describe optimizations (§4.3), and discuss streaming (§4.4). We refer to the extended CS version that accommodates TTLs as CounterStacks^{++} (CS^{++}).

[4]The same approach can also be used for the columns.

c_i	a	b	b	c	**b**	a
c_1	1	2	2	3	3	3
c_2		1	1	2	2	3
c_3			1	2	2	3
c_4				1	2	3
c_5					1	2
c_6						1

Figure 11. Operation of CS: Rows represent counters from oldest (top) to newest (bottom). Columns represent time steps from left to right, with the accessed object identified at the top. Counter values reflect the processed access of each column. For instance, after first access to a, a counter c_1 with value 1 is initialized. With the second access to b, counters c_1 and c_2 update to 2 and 1, respectively. As an example for determining the stack distance, consider the processing of the third access to b (in bold): counter c_3 (boxed) does not increase, but counter c_4 (dash boxed) does, so the value of c_3, namely 2, is the stack distance for the third access to b.

4.1 CounterStacks Background

CS uses Mattson's approach to build the MRC using a histogram of stack distances (§2.1). For ease of understanding, we first describe a basic CS algorithm that is highly inefficient and then describe three optimizations introduced by the CS authors to make it more efficient. CS works as follows. For each access to an object in the workload: (i) a new counter is instantiated and added to a stack of counters, and (ii) all previously created counters are incremented by one if they have not previously encountered an access to the same object. To obtain the stack distance of the current access, the counters are traversed from oldest to newest to find the first counter c_i that was not incremented while the next counter c_{i+1} was incremented. The value of c_i is taken as the stack distance, because counter c_i recorded an access to the same object previously, while counter c_{i+1} did not, and the value of c_i identifies how many distinct objects it has encountered. See Fig. 11 for an example.

Wires et al. introduced the following three optimizations to make the basic algorithm more efficient [28].

A. HLLs. HLL counters are used to estimate the number of distinct objects accessed since the counter was instantiated. Each HLL counter uses a fixed amount of space (and no longer needs to maintain a list of previously accessed objects). The accurate counters are replaced with HLLs, and their reported values are used in the same way as described earlier.

B. Pruning. Once two counters have the same count, their future values will remain the same in perpetuity, because each further access to a new distinct object adds one to both counters. Hence, pruning is used to keep only one of the two counters in order to save space and computation. CS is more aggressive and deletes a younger counter whenever its value is at least $(1-\delta)$ times the older counter, where δ is a fixed pruning parameter. This guarantees that the number of counters is at most $\mathcal{O}(\log M/\delta)$ [28]. Wires et al. used a pruning δ of 0.02 and 0.1 for their High-Fidelity (HiFi) and Low-Fidelity (LoFi) variants [28, 31], respectively.

C. Downsampling. Although pruning limits the number of counters in the stack, the number of instantiated counters is still N, which makes the algorithm slow. With downsampling, a new counter is instantiated only on every d-th access. This reduces the number of instantiated counters from N to N/d. Moreover, to further reduce the computational overhead, the counts of all counters are updated only on every $d-th$ access (instead of on each access). Wires et al. showed that downsampling has minimal impact on the accuracy of the MRCs for the MSR workloads [28]. Their experimental evaluation used a downsampling factor $d = 1$ million. They also add a new counter every 60 seconds for the HiFi variant and every 3,600 seconds for the LoFi variant (based on access reference time), if the downsampling factor is not reached within that time frame.

Downsampling changes the way the stack distance is computed because with downsampling up to d accesses are processed instead of a single access. To estimate the stack distances for those accesses, CS iterates over all the counters, from oldest to newest, and compare how much adjacent counters increased after processing these accesses. If two adjacent counters, H_i and H_j are increased by Δ_i and Δ_j, respectively, we can infer that $(\Delta_j - \Delta_i)$ accesses represent hits in the cache represented by H_i but misses in the cache represented by H_j [28]. Thus, CS increments the H_i histogram bin by $(\Delta_j - \Delta_i)$. For the last counter in the stack, H_n, histogram bin H_n is incremented by $d - H_n$. The reason is that H_n represents the number of distinct accesses in the last processed d accesses, and thus $d - H_n$ hits must have occurred. Wires et al. refer to this process as the *finite differencing scheme* [28]. Finally, the histogram bin corresponding to the ∞ stack distance is incremented by d minus the sum of all previous bin increments to ensure the histogram is updated by d.

4.2 TTLs Support in CounterStacks^{++}

To take TTLs into account, we replace CS's basic HLL with our extended HLL-TTL (§3). Simply integrating the extended HLLs into CS causes an issue related to downsampling. With TTL treatment, processing d accesses often introduces inaccuracies because some of the accesses may be mistakenly identified as misses when they are, in fact, hits. For example, assume object A with an expiry time of 10 is represented in the stack. If a new access to A at time 9 is processed, it should be considered a cache hit. But if we complete processing d accesses at time 15, then the access to A will be mistakenly treated as a miss, because at time 15 A will have been evicted from the stack. We have found that this significantly affects accuracy and hence needs to be addressed.

Our solution is to monitor the upcoming expiry times of objects represented in the stack. We prematurely terminate processing accesses whenever an object expires, thus

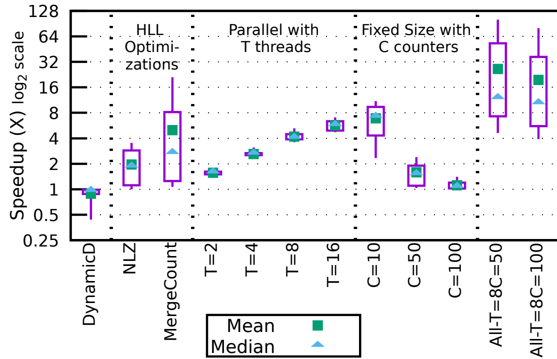

Figure 12. Speedup from each optimization against the original CS algorithm using all 14 MSR workloads, including the `combined` workload used in the CS paper [28]. Line ends show maximum and minimum speedups, while the box marks the 75th and 25th percentiles.

processing fewer than d accesses. This may substantially increase the number of instantiated counters, which has a negative impact on performance. To reduce the number of times a new counter is instantiated, we round eviction times to the closest f seconds. In our implementation, we set f to 30 seconds when CS^{++} is configured for HiFi, and 60 seconds for LoFi. We maintain a priority queue of expiry times to efficiently check for expired objects in $\mathcal{O}(1)$ time. Further, to confine memory usage, only the earliest $8K$ non-expired eviction times are recorded in the priority queue; limiting the recorded expiry times to $8K$ does not affect accuracy.

4.3 Optimizations

We optimized CS^{++} significantly, and these improvements also apply to the original CS algorithm for workloads without TTLs. This section details these optimizations and their effects on the CS algorithm using all the MSR workloads used in the CS paper (and other studies) [28, 29, 31]. Fig. 12 shows the speedups using HLL precision (b = 16) and the HiFi setup. The optimizations result in an average speedup of 26× with 50 counters and 8 threads. Lower HLL precisions yield speedups of 86× and 50× on average for b = 12 and 14. In the LoFi setup, average speedups for HLL precisions b = 12, 14, and 16 are 41×, 33×, and 22×, respectively.

O1. Hardware Supported Instruction (LZCNT). As CS uses HLLs (§3), identifying the NLZs in the binary representation of the hash of the accessed object's key consumes significant processing time. Existing HLL implementations, such as those by Redis[5] and Microsoft[6], use `for loop` and `shift` operations to count the NLZ. We observed that a hardware supported instruction, `LZCNT` (leading zeros count), is more efficient. For CS, this leads to an average speedup of 2×.

O2. New Specialized HLL Operation (`MergeCount`). As detailed in §4.1, after instantiating a new counter, up to d accesses are `Inserted` into all existing counters. The count of each counter is then calculated using the `Count` operation (§3), which requires accessing all buckets of every counter. We devised a new batch processing mechanism to enhance performance by exploiting the HLL `Merge` operation. Instead of adding accesses to all existing counters, we exclusively add them to a new instantiated counter, followed by the execution of our new `MergeCount` operation. This operation merges the latest counter with all previously existing ones, while performing the operations necessary for the `Count` operation. The `MergeCount` operation improved the performance of CS by a factor of 5 on average.

O3. Fixed Size Overhead. We found that we can prune much more aggressively than what was originally proposed for CS and do so with marginal impact on accuracy. In fact, we found we can limit the number of counters to a constant, ***making CS/CS^{++} an MRC-generation algorithm with constant space overhead***. The general strategy employed to limit the amount of memory used is as follows: whenever the existing counters are about to exceed the specified constant number of counters, we invoke the pruning operation with the smallest δ that guarantees that at least one of the existing counters is pruned. This optimization affects the accuracy of the algorithm as it reduces the number of counters. The original CS algorithm uses up to 265 counters, with an average of 141 counters. Limiting the number of counters to 10, 50, and 100 increases the MAE by an average of 6.34%, 0.26%, and 0.05%, respectively, but leads to an average speed up of 6.89×, 1.58×, and 1.11×, respectively.

O4. Parallel Processing. Using the earlier optimization of our batch processing mechanism with `MergeCount`, CS/CS^{++} can trivially be parallelized. The workload is still processed in batches: for each batch, the accesses in the batch are added to the newly instantiated HLL serially, and this HLL can then be `MergeCounted` to the existing HLLs in parallel. This parallel processing does not require locking as there are no conflicts: the newest HLL is read-only and the other HLL buckets are updated exactly once. Using 8 threads, this results in an average speed up of 4.1×.

O5. Dynamic Downsampling. CS uses a constant downsampling factor d, potentially causing inaccuracies for workloads with smaller WSS. We propose adjusting d using the formula $d = WSS_{GB} \times 10{,}000$ (capped at 1M, similar to the original algorithm). The WSS is obtained from the oldest counter in the stack. After this optimization, CS runs at 0.88× its original speed, but its average and max MAE improves from 0.59% and 2.60% to 0.42% and 1.70% respectively.

4.4 Streaming

A major feature of CS and CS^{++} is their ability to stream intermediate counter information; e.g., by persistently storing the most recent counter values after the processing of each

[5]Redis code (method hllPatLen()): https://git.io/JDBtu

[6]Microsoft code (method GetSigma()): https://git.io/JDBtR.

batch. The streams can be used to generate MRCs over arbitrary time intervals, and streams from different workloads can be merged to obtain MRCs for combined workloads.

For workloads without TTLs, the MRC for the accesses between any t_x and t_y can be generated by using the finite differencing scheme (§4.1-C) for the streamed counter values in each successive interval $(t_x, t_{x+1}), (t_{x+1}, t_{x+2}), \cdots,$ (t_{y-1}, t_y), updating the stack distance histogram each time.

Accommodating TTLs introduces a complication with the above method. After processing a batch, some objects previously added to the existing counters might have expired. Because CS^{++} uses HLL-TTL counters when computing the stack distances, objects that have expired while processing the batch are taken into account. However, this is not the case when CS^{++} streams counter values because they are static, which means that expired objects during the processing of the batch are not accounted for. As a result, based on the TTL workloads we analyzed, the generated MRCs becomes so inaccurate, they effectively become unusable.

To address this, we output two streams: a *PreMerge* stream and a *PostMerge* stream. The former records the counts of *unexpired* objects for the existing counter values just before the latest counter is merged with the existing counters, and the latter records the counter values immediately after the merge. The finite differencing scheme is then applied between the latest *PreMerge* and the *PostMerge* counter values (instead of between the latest *PostMerge* counts and the previous *PostMerge* counts).

As an alternative to streaming counter values, it is possible to stream the HLL (or HLL-TTL) counters directly. While this consumes significantly more storage space, it alleviates the need to use two streams since the HLL-TTLs reflect expiry times. Streaming HLL counters (as opposed to counter values) has the further advantage that it allows generating MRCs of combined workloads even if the workloads being merged access common objects. In contrast, when streaming counter values, the workloads being combined cannot access common objects (as observed by Wires et al. [28]) because otherwise common objects would be counted multiple times when the streams are merged. Streaming HLL counters resolves this issue because common objects do not increase the HLL (or HLL-TTL) counts; i.e., the `Insert` and `Merge` operations are idempotent.

5 Exploiting HLL-TTL for WSS Estimation

There are mixed opinions as to the utility of WSS. Some consider it *"not enough to guide memory allocation"* and find MRCs to offer more utility [29, 71]. Others have found WSS to be sufficient to guide memory allocations without the need for MRCs [72]. We have found that both the WSS and MRC have their own use cases. For small WSS values, directly setting the cache configuration to the WSS will minimize the miss rate. Conversely, for larger WSS values, the MRC provides insights into cache size versus miss rate trade-offs, which are not available when using WSS.

Table 2. Workloads used in this paper. For the Tencent dataset, it contains 5,584 traces mapped to 40 cache instances.

	# workloads	# accesses
MSR Cambridge [44]	13	434M
Twitter [23]	54	247B
Wikipedia [73]	1	2.5B
SEC EDGAR [68, 69]	58	25B
IBM [67]	98	1.6B
Tencent [35]	5,584	30B

Fig. 13 (left) shows the WSS CDF across the 264 workloads from 6 different collections shown in Table 2 (without considering TTLs). Nearly 20% of these workloads have a WSS of less than 1GB. By allocating 1GB of memory to each cache serving these workloads, we achieve the lowest possible miss rate. Fig. 13 (right) shows the WSS CDF across 28 workloads from the Twitter collection (described in §6). Taking TTLs into account drastically reduces the largest WSS from over 2TB to 64GB. The effect of TTLs is substantial: 80% of the analyzed workloads have a WSS of 16GB at most, compared to 256GB when disregarding TTLs. Furthermore, 60% of the workloads have a WSS of less than 6GB when considering TTLs, compared to 32GB when disregarding TTLs. In cases like these, WSS alone often suffices for allocation guidance without MRC-generation.

WSS estimation through cardinality estimation aims to measure the number of distinct objects in a multiset. HLL is one of the most efficient tools for this task [55, 56]. In contrast, tools tailored for object membership testing (like Bloom and Cuckoo filters) are less efficient than the HLL due to their broader scope [28].

To the best of our knowledge, our extended HLL-TTL (§3) is the first to accurately estimate the WSS of workloads in constant space when taking TTLs into account. Its worst-case memory usage for precision $b = 12$ is 832KB (140KB when using the dynamic implementation), and results in 98.8% accuracy. In contrast, exact WSS calculation requires space linear to the number of distinct objects in the workload. For example, the workloads in the Twitter collection combined include 247 billion accesses to 25 billion distinct objects, which requires 279 GB of memory to compute the exact WSS. Our HLL-TTL based estimate is up to *five orders of magnitude* more memory-efficient.

6 Evaluation

In this section we show that (1) significant memory savings can be achieved when sizing caches using TTL-aware WSSs and MRCs, (2) existing WSS and MRC algorithms, which do not take TTL into account, are highly inaccurate when applied to workloads with TTLs, (3) the performance of our TTL-aware algorithms are comparable to that of the existing

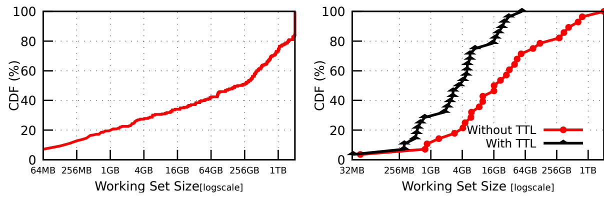

Figure 13. The left figure shows the WSS CDF for the workloads listed in Table 2. The right shows the WSS CDF for 28 TTL-related Twitter workloads (with and without TTLs).

TTL-agnostic algorithms, and (4) our extended algorithms maintain consistent accuracy across varied configuration parameters. We first consider WSS results (§6.1) and then MRC results (§6.2).

Experimental Setup and Workloads. Our experiments were conducted on a server equipped with an Intel 13900KS CPU and 128GB of DDR5-4800MHz DRAM. Workloads were read from a Corsair PCIe 4.0 MP600 PRO 8TB NVME SSD after they were formatted into a binary format. We use an in-house system that processes the access workloads and generates the WSS and MRC using each of the algorithms discussed in this paper. In all our generated MRCs, the stack distance histogram is divided into 64K buckets each representing 32MB, supporting a cache size up to 2TB; a similar approach was used in earlier studies [27, 29, 31].[7]

Since the Twitter workloads are the only publicly available workloads with TTLs, only they were used to evaluate claims regarding TTL. We included all workloads recommended by Twitter [66],[8] as well as the workloads with a median TTL less than the duration of the workload. These 28 Twitter workloads (out of 54) include 113B accesses.[9] We only used the GET requests from the workloads, as in previous studies [28, 31, 34].[10] We extracted the TTL information from the SET requests corresponding to the same key used in the GET request, as GET requests do not include TTL information.

6.1 WSS Results

To evaluate the impact of TTLs on the WSS, we compared the exact WSS_{ttl} and the exact WSS_{nottl} across all 28 workloads. Notably, the WSS_{ttl} fluctuates over time, as previously illustrated in Fig. 4. Hence, in our comparisons, we used the *high watermark* of the WSS_{ttl} values as measured at the end of each hour over the duration of the workload. The potential memory savings by accommodating TTLs is presented in terms of *Relative Savings* $\left(RS = \frac{WSS_{nottl} - WSS_{ttl}}{WSS_{nottl}}\right)$.

[7]AET uses logarithmic ranges [31].

[8]Recommended TTL clusters are: 6, 7, 11, 18, 19, 22, 25, 52 [66].

[9]The 28 workloads are: {4, 6, 7, 8, 11, 13, 14, 16, 18, 19, 22, 24, 25, 29, 30, 33, 34, 37, 40, 41, 42, 43, 46, 48, 49, 50, 52, 54}. This subset is larger than those of previous related studies [41, 72].

[10]The SET requests in the workloads were captured when running a specific (undisclosed) cache size, and the number and placement of SET requests would be different for different cache sizes.

Using WSS_{ttl} instead of WSS_{nottl} results in 69% memory savings, on average. Fig. 14 shows that sizing caches using WSS_{ttl} instead of WSS_{nottl} results in a relative savings of between 7.20% and 99.93% per workload, with an average of 69.38%. The aggregate WSS_{ttl} (242.3GB) is 96% less than the aggregate WSS_{nottl} (7.8TB).

The savings achieved depend primarily on the distribution of TTL limits in the workload and the access patterns. Typically, longer TTLs result in lower savings due to less frequent expiration, whereas shorter TTLs lead to increased savings. Our analysis of common TTLs used across the evaluated workloads supports this.

Consider workloads 52 and 6. For workload 52, 62% of the accesses have a TTL of 24 hours, and 36% have a TTL of 336 hours, resulting in approximately 30% savings. In contrast, for workload 6, where 90% of accesses have a TTL of at most 10 minutes, the savings increase to around 90%. These examples underscore the significant impact that TTL duration has on savings.

To further illustrate the effect of access patterns on savings, we compare workloads 14 and 49. Both workloads feature a TTL of 24 hours for all accessed objects. However, the savings differ markedly: workload 14 achieves 71% savings, whereas workload 49 has 19% savings. This discrepancy is clarified upon examining the high watermark of the number of non-expired objects present in the cache during these workloads. For workload 14, only 28% of the total objects coexist in the cache at any given time in the worst-case scenario, compared to 77% for workload 49. This analysis highlights the nuanced ways in which TTL settings and access patterns together influence cache efficiency and savings.

The estimation error of our HLL-TTL (when applied to workloads with TTLs) is in line with the expected error of the original HLL (when applied to workloads without TTLs). To evaluate the accuracy of our HLL-TTL, we used *Absolute Relative Error* $\left(ARE = \frac{|Exact\ WSS_{ttl} - Estimated\ WSS_{ttl}|}{Exact\ WSS_{ttl}}\right)$. We tested using different values for HLL precision parameter b within the range $[8-16]$. Fig.15 shows that precisions $b = 12, 13,$ and 14 exhibit AREs of 1.14%, 0.85%, and 0.7%, respectively. The figure also shows the original theoretical standard error of HLL-NoTTL, $\sigma = \frac{1.04}{\sqrt{2^b}}$ [55]. The error of our HLL-TTL is in line with that expected error.

For throughput, Fig.16 shows that precisions $b = 12, 13,$ and 14, achieve average throughput of $50M$, $47M$, and $43M$ accesses per second, respectively. The primary reason for the decrease in throughput as the precision increases is poorer cache locality. The memory usage of the dense and dynamic implementations of our HLL-TTL for different precision levels ($b = 8$ to 16) is summarized in Table 3.

6.2 MRC Results

MRCs are more complex to evaluate than WSS because each MRC identifies a tradeoff between the cache size and miss

Table 3. Comparison of space usage for dense and dynamic HLL-TTL implementations. Units are in KB.

HLL b=	8	9	10	11	12	13	14	15	16
Dense	56	110	216	424	832	1,536	3,195	6,267	12,288
Dynamic	11	21	39	73	136	252	466	851	1,536

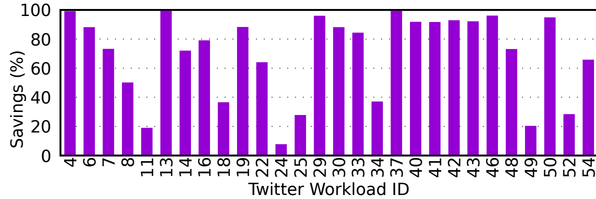

Figure 14. Relative memory savings across workloads when sizing memory with WSS_{ttl} instead of WSS_{nottl}. Savings range from 7.20% to 99.93%, with an average of 69.38%.

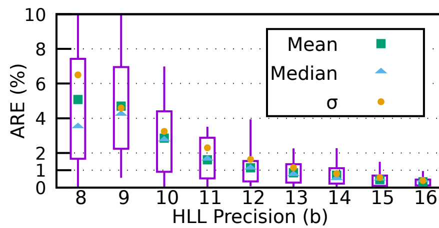

Figure 15. Sensitivity of the precision parameter, b, for HLL-TTL. σ is the theoretical Standard Error of the original HLL-NoTTL (when applied to workloads without TTL). In Figures 15, 16, 17, and 18, each line represents the range of results across the 28 workloads analyzed. The line's top and bottom represent maximum and minimum results. The box's top and bottom represent the 75th and 25th percentiles.

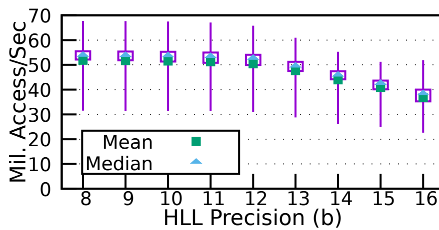

Figure 16. Throughput of HLL-TTL for different precisions.

rate. We observed that the MRCs of most workloads have long tails with a small negative slope before reaching steady-state. Thus, a system operator might decide to downsize the cache while accepting a slight increase in miss rate, depending on the application. For example, an operator may deem a miss rate increase of $t \in \{0\%, 0.1\%, 0.5\%, 1\%\}$ to be acceptable in return for lower memory requirements.

Utilizing MRC_{ttl} instead of MRC_{nottl} results in a memory saving of 66%, on average. For each of the workloads examined, we measured the smallest cache size needed to achieve the minimal miss rate $+t$ for both the exact MRC_{ttl} and exact MRC_{nottl}, then quantified the savings using the same relative saving metric, RS, described above. Utilizing MRC_{ttl}, instead of MRC_{nottl}, leads to memory savings of 66%, 56%, 52%, and 49%, on average per workload, for $t = 0\%$, 0.1%, 0.5%, and 1%, respectively. These results may appear counter-intuitive, as one might think that an increase in tolerance, t, should correspond to an increase in savings. However, MRC_{ttl} has a notably shorter tail than MRC_{nottl} (e.g., Fig. 3), making an increase in t less beneficial. We also measured the aggregate of the smallest cache sizes needed to achieve the minimal miss rate $+t$ for both the exact MRC_{ttl} and exact MRC_{nottl}, assuming one cache instance per workload, quantifying the savings as we did earlier. In aggregate, utilizing MRC_{ttl}, instead of MRC_{nottl}, leads to memory savings of 94% (from 4.49TB to 243.66GB), 93% (from 2.50TB to 173.81GB), 88% (from 1.08TB to 131.13GB), 83% (from 618.28GB to 104.94GB) for $t = 0\%$, 0.1%, 0.5%, and 1%, respectively.

Existing MRC-generation algorithms which do not accommodate TTLs can misreport miss rates by up to 38% on workloads with TTLs. To quantify how much existing MRC-generation algorithms deviate from the exact MRC_{ttl}, we measured the *Mean Absolute Deviation* $\left(MAD = \frac{\sum_1^B |MRC_{ttl}[i] - MRC_{nottl}[i]|}{B}\right)$ between MRC_{nottl} and MRC_{ttl}, where B is the maximum number of points in either MRC. We compared points on both MRCs (in increments of 32MB) up to the point where both MRCs cease to change, extending the shorter MRC as necessary.[11] MRC_{nottl} deviates by up to 38% from MRC_{ttl}, with an average MAD of 5%, as shown in Fig. 17 (a).

The errors introduced by our extended MRC-generation algorithms (when applied to workloads with TTLs) are comparable to those of their original counterparts (when applied to workloads without TTLs). To evaluate the accuracy of our TTL-accommodating MRC-generation algorithms, we measured the *Mean Absolute Error* $\left(MAE = \frac{\sum_1^B |MRC_{ttl}[i] - MRC_{nottl}[i]|}{B}\right)$ between approximate MRC_{ttl} and the exact MRC_{ttl} obtained using our Olken^{++} algorithm. The MAE is widely used to quantify MRC errors [28, 29, 31–33, 37, 72, 74, 75]. Fig. 17 (b) shows the errors for different configuration parameters to illustrate the sensitivity of these algorithms to their configuration parameters. We make several observations. First, the SHARDS$_{adj}$ variant should always be used over SHARDS as it has significantly lower MAEs. For example, FR-SHARDS^{++}−0.001 has a worst-case MAE of 8.7%, when not adjusted, while having a worst-case MAE of 3.5%, when adjusted (FR-SHARDS$^{++}_{adj}$−0.001). Second, FS-SHARDS$^{++}_{adj}$ with $S_{max} = 1K$ is surprisingly accurate with an average MAE of 0.4%. Increasing S_{max} to $8K$ and

[11]This is crucial because if we compare up to a much larger size (e.g., 2TB), the deviation (or error) will be dominated by the last point on the MRC.

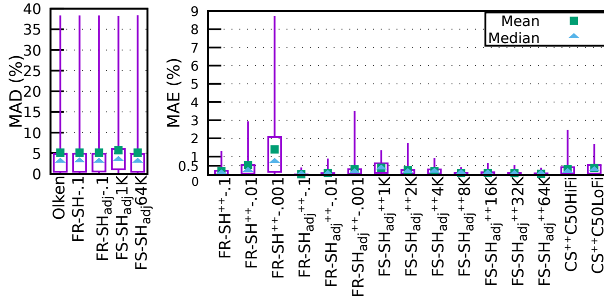

Figure 17. (Left) Deviation of exact and approximate MRC_{nottl} from exact MRC_{ttl} in terms of MAD. **(Right)** MAE between approximate MRC_{ttl} and exact MRC_{ttl} in terms of MAE.

64K, reduces the average MAE to 0.09% and 0.06%, respectively, but increases memory usage by a factor of 8× and 64×, respectively. Third, CS^{++}, running with only 50 counters, has an average MAE of 0.32% and 0.39% for the HiFi and LoFi variants, respectively.[12]

The throughput of our extended algorithms is comparable to those of their original counterparts. Fig. 18 shows the throughput of the discussed algorithms using different configuration parameters. The throughput of Olken^{++} is, on average, 15% faster than the original Olken algorithm (which does not support TTLs) because the number of objects in the Olken^{++} tree is reduced due to TTL evictions. The throughput of FR-Shards is not affected by our extensions. FS-Shards$^{++}_{adj}$ is 4.7%, 13.7%, and 24% slower that the original FS-Shards$_{adj}$ for S_{max} = 1K, 8K, and 64K, respectively, due to the extra processing overhead introduced by evictions from two priority queues *F-PQ* and *ET-PQ* (§2.3). CS^{++} is 762% and 170% faster that the original CS for the HiFi and LoFi variants.

As a side observation, our study yielded a surprising result: the Shards^{++} variants exhibited a significantly higher accuracy compared to CS^{++}. This outcome was unanticipated as CS has generally been perceived as being more accurate than Shards, as even suggested by the authors of Shards [29]. To validate this unexpected finding, we undertook a comparative analysis between Shards and CS (*both without our extensions*), using the 264 workloads listed in Table 2. This comparison confirmed that, without TTLs, Shards$_{adj}$ indeed outperforms CS in all aspects (accuracy, throughput, and memory usage). The primary advantage of CS over Shards, however, is streaming.

[12]Surprisingly, the HiFi variant has a *worst-case* MAE of 2.4%, compared to LoFi's 1.7%. Similarly, increasing S_{max} in FS-Shards$^{++}_{adj}$ from 1K to 2K raises the worst-case MAE from 1.3% to 1.7%. The reason for this is being investigated. Average MAE decreases with higher precision as anticipated.

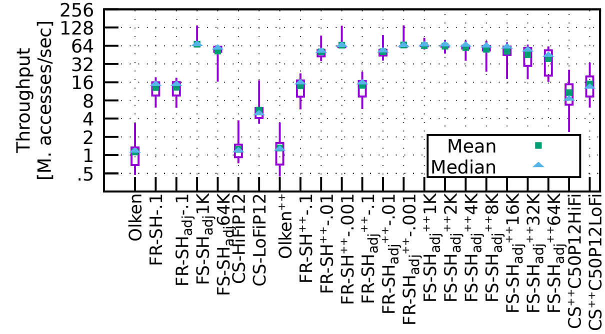

Figure 18. Throughput of all tested algorithms.

7 Some Practical Considerations

In this section we address two practical considerations. The first is related to accommodating heterogeneous objects sizes and the second is comparison to real-world caches.

Accommodating heterogeneous object sizes. Using a uniform object size has been the most widely used approach in past MRC studies [28, 29, 31]. Carra et al. proposed an extension to Olken and Shards algorithms to support heterogeneous object sizes, thus improving MRC-generation accuracy when modeling real-world caches that use heterogeneous object sizes [65].[13] Carra et al.'s extension, however, requires modifying these algorithms' internal data structures [65]. In contrast, Olken described how to accommodate heterogeneous object sizes without replacing the original data structure (i.e., AVL tree), by adjusting node weights to reflect the total size of child nodes [25]. The same approach can be applied to Olken^{++}, Shards^{++}, and Shards to accommodate heterogeneous object sizes.

Computing the WSS with heterogeneous object sizes can be achieved by using multiple HLLs, one for each object size group (e.g., grouped as powers of 2), and track the average object size per HLL to reduce the variance. Thus, the WSS becomes the aggregate sum of each HLL's cardinality multiplied by the HLL's average object size. The downside of this approach is that it increases the space required by a factor of the number of object size groups maintained. The same approach can be applied to HLL-TTL, which in turn enables CounterStacks^{++} to also process heterogeneous object sizes.

As an alternative for both the MRC and WSS, one can simply use the average object size in the workload. The average object size can be computed efficiently while processing the workload using the Welford's method [76], which only requires extra 16 bytes: a `double` for the average and an `integer` for the count. However, this approach of just using the object size average results in lower accuracy than the previously described approaches.

[13]However, they did not report that the state-of-the-art MRC-generation does not accommodate TTLs, which causes a significant gap when used to model workloads with TTLs, as we have shown.

Comparison to real-world caches. Most existing in-memory caches do not implement eviction policies in their ideal form to improve the efficiency of the production system. LRU, which is the most widely used eviction policy [67, 77], is often approximated as well. For instance, sampling is used in Redis to approximate LRU evictions [78], and CacheLib promotes accessed objects to the most recently accessed position in the LRU stack lazily in order to reduce contention [79]. Other resource constrained caches such as flash caches approximate even more to reduce the amount of metadata per object [80, 81].

To understand how well MRC-generation algorithms track the behavior of real-world in-memory caches, we compared our Olken^{++} results with actual measurements obtained from experiments with Redis instances using the same 28 workloads listed in §6 and found that the miss rates deviate by less than 1% in the worst case. Since our Olken^{++} is exact, our approximate algorithms are expected to have error rates inline with the error rates shown in §6. Hence, our approximately generated MRCs and WSSs will track the behavior of real-world caches reasonably well. In contrast, the original Olken miss rates when not accommodating TTLs, deviate by 6.2% on average and 45.8% for the worst case from Redis (confirming our results in Fig. 17). Since Olken is exact (without TTLs), then the the state-of-the-art approximate MRC-generation algorithms that aim to approximate Olken's results also significantly deviate from real-world use cases.

Our results are further supported by Redis documentation which notes: *"In simulations we found that using a power law access pattern, the difference between true LRU and Redis approximation was minimal or non-existent."*[78]. Twitter workloads also follow a power law distribution [23, 40]. Similarly, the approximations presented by CacheLib and RIPQ do not affect miss rates in practice [79, 81].

In addition, although real-world in-memory caches are used to evict expired objects lazily [79], this has changed recently due to the benefits of proactive evictions. For instance, in 2019, Twitter found that Redis' memory usage can be reduced by up to 25% by employing more proactive object expiry [82]. This motivated Redis to introduce proactive object expiry using a radix tree of expiry times starting with Redis 6.0 which guarantees objects are removed within 1*ms* of their expiry [83, 84]. Other in-memory caches have also started to proactively evict objects [41, 77].

8 Related Work

MRCs have long served as a foundational tool for analyzing the relationship between cache size and miss ratio. Over the last five decades, a wealth of MRC-generation algorithms have been proposed [24, 25, 27–30, 32, 35–38, 72, 74, 75, 85–94]. In this paper, we covered the seminal algorithms: Mattson, Olken, SHARDS, and Counterstacks. MRCs are widely utilized to optimize cache resources [22, 35, 38, 75, 88, 90, 92, 93, 95–102]. Dynacache and LAMA use MRCs to improve Memcached's slab allocation [22, 90]. Talus and eMRC utilize MRCs to remove performance cliffs through cache partitioning [30, 38]. Centaur uses MRCs to manage cache allocations for virtual machines [99]. OSCA, mPart, ORCA, and APAC utilize MRCs to optimize cache allocations [35, 38, 92, 101].

The WSS has also been a focal point of research, offering insights into application memory demands. Its importance is evidenced by numerous studies in the last five decades [39, 45, 72, 97, 103–116]. WSS has practical applications in cache management and has been a guiding metric for memory allocations [45, 72, 97, 114, 115, 117]. WSS estimation is related to cardinality estimation, which has been the subject of extensive research [46–56, 118]; both aim to measure statistics concerning the number of distinct objects [28].

TTL limits are prevalent in caching systems [23, 40–43, 57–65]. Their importance is underscored by practices such as Twitter's TTL enforcement for in-memory caches [40, 41]. Yang et al. have also highlighted the necessity of TTLs for in-memory caching, emphasizing their role in effective WSS management [40]. The rising significance of TTLs in storage systems, especially in the context of GDPR compliance, is worth noting [64]. TTLs are more generally becoming increasingly important for compliance with privacy laws. To the best of our knowledge, we are the first to extend MRC-generation and WSS estimation algorithms to accommodate TTLs.

9 Concluding Remarks

This paper addresses the critical issue of tools to support the sizing of in-memory caches in modern cloud environments. The key issue we focused on is to incorporate TTL handling into the state-of-the-art MRC-generation and WSS estimation algorithms. We demonstrated the importance of modeling TTLs for in-memory caches. We adapted the Mattson, Olken, and SHARDS algorithms to handle TTLs, and we extended HLLs to accommodate deletion of expired objects. These extensions, in turn, enabled efficient WSS estimation and a TTL-enabled CounterStacks MRC-generation algorithm. Extensive evaluation on a large number of workloads shows the effectiveness of our algorithms.

Future research directions include adapting our extended HLLs to the sliding window model [118], and investigating the effects of delayed hits [119] on MRC-generation. Also, we intend to utilize the presented tools in a real-time decision making system to optimize the resources of in-memory caches.

Acknowledgments.

Thanks to the anonymous reviewers, our shepherd Daniel Berger, and Ashvin Goel for their valuable feedback that helped improve the quality of this paper.

References

[1] Memcached.org. Memcached. https://memcached.org. [Online; 2023].

[2] Brad Fitzpatrick. Distributed caching with Memcached. *Linux J.*, 2004(124), 5 2004.

[3] Redis Labs. Redis. https://redis.io. [Online; 2023].

[4] Amazon. Amazon Elasticache. https://aws.amazon.com/elasticache/. [Online; 2023].

[5] Google. Google Memorystore. https://cloud.google.com/memorystore. [Online; 2023].

[6] Microsoft. Microsoft Azure cache. https://azure.microsoft.com/en-us/services/cache/. [Online; 2023].

[7] Huawei. Huawei Distributed Cache Service. https://www.huaweicloud.com/en-us/product/dcs.html. [Online; 2023].

[8] IBM. IBM Cloud Databases for Redis. https://www.ibm.com/cloud/databases-for-redis. [Online; 2023].

[9] Oracle. Using caches in Oracle application container cloud service. https://docs.oracle.com/en/cloud/paas/app-container-cloud/cache/index.html. [Online; 2023].

[10] DigitalOcean. Managed Redis. https://www.digitalocean.com/products/managed-databases-redis/. [Online; 2023].

[11] Alibaba. ApsaraDB for Memcache. https://www.alibabacloud.com/product/apsaradb-for-memcache. [Online; 2023].

[12] ObjectRocket. ObjectRocket for Redis. https://www.objectrocket.com/managed-redis/. [Online; 2023].

[13] Tencent. TencentDB for Redis. https://intl.cloud.tencent.com/product/crs. [Online; 2023].

[14] Timothy Zhu, Anshul Gandhi, Mor Harchol-Balter, and Michael A Kozuch. Saving cash by using less cache. In *Proc. 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'12)*, 2012.

[15] Azure. Azure cache for Redis pricing. https://azure.microsoft.com/en-us/pricing/details/cache/. [Online; 2023].

[16] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, Peter Vajgel, et al. Finding a needle in Haystack: Facebook's photo storage. In *Proc. 9th Conf. on Operating Systems Design and Implementation (OSDI'10)*, pages 1–8, 2010.

[17] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling Memcache at Facebook. In *Proc. 10th USENIX Symp. on Networked Systems Design and Implementation (NSDI'13)*, pages 385–398, 2013.

[18] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[19] Alex Hall, Alexandru Tudorica, Filip Buruiana, Reimar Hofmann, Silviu-Ionut Ganceanu, and Thomas Hofmann. Trading off accuracy for speed in powerdrill. *Google Research*, 2016.

[20] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proc. 12th Intl. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'12)*, pages 53–64, 2012.

[21] Qi Huang, Ken Birman, Robbert Van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. An analysis of Facebook photo caching. In *Proc. 24-th Symp. on Operating Systems Principles (SOSP'13)*, pages 167–181, 2013.

[22] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Dynacache: Dynamic cloud caching. In *Proc. 7th Workshop on Hot Topics in Cloud Computing (HotCloud'15)*, 2015.

[23] Juncheng Yang, Yao Yue, and KV Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proc. 14th Symp. on Operating Systems Design and Implementation (OSDI'20)*, pages 191–208, 2020.

[24] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[25] Frank Olken. Efficient methods for calculating the success function of fixed-space replacement policies. Master's thesis, University of California, Berkeley), 1981.

[26] Weiming Zhao, Xinxin Jin, Zhenlin Wang, Xiaolin Wang, Yingwei Luo, and Xiaoming Li. Efficient LRU-based working set size tracking. *Michigan Technological University Computer Science Technical Report*, 2011.

[27] Qingpeng Niu, James Dinan, Qingda Lu, and P. Sadayappan. PARDA: A fast parallel reuse distance analysis algorithm. In *Proc. 26th Intl. Parallel and Distributed Processing Symp. (IPDPS'12)*, pages 1284–1294. IEEE, 2012.

[28] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, and Andrew Warfield. Characterizing storage workloads with counter stacks. In *Proc. 11th Symp. on Operating Systems Design and Implementation (OSDI'14)*, pages 335–349, 2014.

[29] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *Proc. 13th Conf. on File and Storage Technologies (FAST'15)*, pages 95–110, 2015.

[30] Nathan Beckmann and Daniel Sanchez. Talus: A simple way to remove cliffs in cache performance. In *Proc. 21st Intl. Symp. on High Performance Computer Architecture (HPCA'15)*, pages 64–75. IEEE, 2015.

[31] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. Kinetic modeling of data eviction in cache. In *Proc. USENIX Annual Technical Conf. (ATC'16)*, pages 351–364, 2016.

[32] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *Proc. USENIX Annual Technical Conf. (ATC'17)*, pages 487–498, 2017.

[33] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Zhenlin Wang, Chen Ding, and Chencheng Ye. Fast miss ratio curve modeling for storage cache. *ACM Trans. on Storage (TOS'18)*, 14(2):1–34, 2018.

[34] Cheng Pan, Xiameng Hu, Lan Zhou, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. Pace: Penalty aware cache modeling with enhanced AET. In *Proc. 9th Asia-Pacific Workshop on Systems*, pages 1–8, 2018.

[35] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. OSCA: An online-model based cache allocation scheme in cloud block storage systems. In *Proc. USENIX Annual Technical Conf. (ATC'20)*, pages 785–798, 2020.

[36] Jiangwei Zhang and YC Tay. PG2S+: Stack distance construction using popularity, gap and machine learning. In *Proc. The Web Conf. (WWW'20)*, pages 973–983, 2020.

[37] Damiano Carra and Giovanni Neglia. Efficient miss ratio curve computation for heterogeneous content popularity. In *Proc. USENIX Annual Technical Conf. (ATC'20)*, pages 741–751, 2020.

[38] Zhang Liu, Hee Won Lee, Yu Xiang, Dirk Grunwald, and Sangtae Ha. eMRC: Efficient miss ratio approximation for multi-tier caching. In *Proc. 19th Conf. on File and Storage Technologies (FAST'21)*, pages 293–306, 2021.

[39] Zhilu Lian, Yangzi Li, Zhixiang Chen, Shiwen Shan, Baoxin Han, and Yuxin Su. eBPF-based working set size estimation in memory management. In *Proc. 2022 intl. conf. on Service Science (ICSS'22)*, pages 188–195, 2022.

[40] Juncheng Yang, Yao Yue, and KV Rashmi. A large-scale analysis of hundreds of in-memory key-value cache clusters at Twitter. *ACM Transactions on Storage (TOS'21)*, 17(3):1–35, 2021.

[41] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: A memory-efficient and scalable in-memory key-value cache for small objects. In *Proc. 18th USENIX Symp. on Networked Systems Design and Implementation (NSDI'21)*, pages 503–518, 2021.

[42] Alexander Fuerst and Prateek Sharma. Faascache: Keeping serverless computing alive with greedy-dual caching. In *Proc. the 26th ACM Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, pages 386–400, 2021.

[43] Gerhard Hasslinger, Mahshid Okhovatzadeh, Konstantinos Ntougias, Frank Hasslinger, and Oliver Hohlfeld. An overview of analysis methods and evaluation results for caching strategies. *Computer Networks*, 228:109583, 2023.
[44] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Trans. Storage (TOS'08*, pages 10:1 – 10:23, 10 2008.
[45] Vlad Nitu, Aram Kocharyan, Hannas Yaya, Alain Tchana, Daniel Hagimont, and Hrachya Astsatryan. Working set size estimation techniques in virtualized environments: One size does not fit all. In *Proc. the ACM on Measurement and Analysis of Computing Systems*, pages 1–22, 2018.
[46] Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.
[47] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.
[48] Ziv Bar-Yossef, TS Jayram, Ravi Kumar, D Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *Proc. Intl. Workshop on Randomization and Approximation Techniques in Computer Science*, pages 1–10, 2002.
[49] Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities. In *Proc. European Symp. on Algorithms*, pages 605–617, 2003.
[50] Frédéric Giroire. Order statistics and estimating cardinalities of massive data sets. *Discrete Applied Mathematics*, 157(2):406–427, 2009.
[51] Kevin Beyer, Peter J Haas, Berthold Reinwald, Yannis Sismanis, and Rainer Gemulla. On synopses for distinct-value estimation under multiset operations. In *Proc. Intl. Conf. on Management of Data (SIGMOD'07)*, pages 199–210, 2007.
[52] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. on Database Systems (TODS'90)*, 15(2):208–229, 1990.
[53] Odysseas Papapetrou, Wolf Siberski, and Wolfgang Nejdl. Cardinality estimation and dynamic length adaptation for Bloom filters. *Distributed and Parallel Databases*, 28(2):119–156, 2010.
[54] Hazar Harmouch and Felix Naumann. Cardinality estimation: An experimental survey. *Proc. of the VLDB Endowment*, 11(4):499–512, 2017.
[55] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *Proc. Discrete Mathematics and Theoretical Computer Science*, pages 137–156, 2007.
[56] Stefan Heule, Marc Nunkesser, and Alexander Hall. Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proc. 16th Intl. Conf. on Extending Database Technology*, pages 683–692, 2013.
[57] Internet Engineering Task Force (IETF). Domain name system (DNS) IANA considerations. https://aws.amazon.com/elasticache/. [Online; 2023].
[58] Jaeyeon Jung, Arthur W Berger, and Hari Balakrishnan. Modeling TTL-based internet caches. In *Proc. 22nd Annual Joint Conf. of the IEEE Computer and Communications Societies*, volume 1, pages 417–426, 2003.
[59] Nicaise Choungmo Fofack, Philippe Nain, Giovanni Neglia, and Don Towsley. Performance evaluation of hierarchical TTL-based cache networks. *Computer Networks*, 65:212–231, 2014.
[60] Daniel S Berger, Philipp Gland, Sahil Singla, and Florin Ciucu. Exact analysis of TTL cache networks. *Performance Evaluation*, 79:2–23, 2014.
[61] Ningwei Dai, Yunpeng Chai, Yushi Liang, and Chunling Wang. ETD-cache: An expiration-time driven cache scheme to make SSD-based read cache endurable and cost-efficient. In *Proc. 12th ACM Intl. Conf. on Computing Frontiers*, pages 1–8, 2015.
[62] Soumya Basu, Aditya Sundarrajan, Javad Ghaderi, Sanjay Shakkottai, and Ramesh Sitaraman. Adaptive TTL-based caching for content delivery. *Transactions on Networking*, 26(3):1063–1077, 2018.
[63] Damiano Carra, Giovanni Neglia, and Pietro Michiardi. TTL-based cloud caches. In *Proc. Conf. on Computer Communications*, pages 685–693, 2019.
[64] Aashaka Shah, Vinay Banakar, Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. Analyzing the impact of GDPR on storage systems. In *Proc. 11th Workshop on Hot Topics in Storage and File Systems (HotStorage'19)*, 2019.
[65] Damiano Carra, Giovanni Neglia, and Pietro Michiardi. Elastic provisioning of cloud caches: A cost-aware TTL approach. *Transactions on Networking*, 28(3):1283–1296, 2020.
[66] Twitter. Anonymized cache request traces from Twitter production. https://github.com/twitter/cache-trace. [Online; 2023].
[67] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen Kat. It's time to revisit LRU vs. FIFO. In *Proc. 12th Workshop on Hot Topics in Storage and File Systems (HotStorage'20)*, 2020.
[68] U.S. Securities and Exchange Commission (SEC). Electronic data gathering, analysis, and retrieval system (EDGAR) log file dataset. https://www.sec.gov/dera/data/edgar-log-file-data-set.html. [Online].
[69] James Ryans. Using the EDGAR log file data set. *Available at SSRN 2913612*, 2017.
[70] Saguiitay. Cardinality estimation. https://github.com/microsoft/CardinalityEstimation/. [Online; 2023].
[71] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proc. 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, pages 177–188, 2004.
[72] Rong Gu, Simian Li, Haipeng Dai, Hancheng Wang, Yili Luo, Bin Fan, Ran Ben Basat, Ke Wang, Zhenyu Song, Shouwei Chen, Beinan Wang, Yihua Huang, and Guihai Chen. Adaptive online cache capacity optimization via lightweight working set size estimation at scale. In *Proc. USENIX Annual Technical Conference (ATC'23)*, pages 467–484, 2023.
[73] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, 2009.
[74] David K Tam, Reza Azimi, Livio B Soares, and Michael Stumm. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *Proc. Intl. Conf. on Architectural Support for Programming Languages & Operating Systems (ASPLOS'09)*, pages 121–132, 2009.
[75] Trausti Saemundsson, Hjortur Bjornsson, Gregory Chockler, and Ymir Vigfusson. Dynamic performance profiling of cloud caches. In *Proc. Symp. on Cloud Computing (SoCC'14)*, pages 1–14, 2014.
[76] BP Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.
[77] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and KV Rashmi. Sieve is simpler than LRU: an efficient turn-key eviction algorithm for web caches. In *Proc. 21st USENIX Symp. on Networked Systems Design and Implementation (NSDI'24)*, 2024.
[78] Redis. Redis eviction. https://redis.io/docs/reference/eviction/. [Online; 2024].
[79] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. The CacheLib caching engine: Design and experiences at scale. In *Proc. 14th Symp. on Operating Systems Design and Implementation (OSDI'20)*, pages 753–768, 2020.

[80] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S Berger, Nathan Beckmann, and Gregory R Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proc. 28th Symp. on Operating Systems Principles (SOSP'21)*, pages 243–262, 2021.

[81] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: Advanced photo caching on flash for Facebook. In *Proc. 13th USENIX Conf. on File and Storage Technologies (FAST'15)*, pages 373–386, 2015.

[82] Twitter. Improving key expiration in Redis. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2019/improving-key-expiration-in-redis. [Online; 2024].

[83] Redis. Diving into Redis 6.0. https://redis.com/blog/diving-into-redis-6/. [Online; 2024].

[84] Redis. Discussion by the maintainers of Redis on Twitter's article [82]. https://news.ycombinator.com/item?id=19662501. [Online; 2024].

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

[86] James Gordon Thompson. *Efficient analysis of caching systems*. PhD thesis, University of California, Berkeley, 1987.

[87] George Almási, Călin Caşcaval, and David A Padua. Calculating stack distances efficiently. In *Proc. Workshop on Memory System Performance*, pages 37–43, 2002.

[88] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proc. the 11th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, page 177–188, 2004.

[89] David Eklov and Erik Hagersten. Statstack: Efficient modeling of LRU caches. In *Proc. Intl. Symp. on Performance Analysis of Systems & Software (ISPASS'10)*, pages 55–65. IEEE, 2010.

[90] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. LAMA: Optimized locality-aware memory allocation for key-value cache. In *Proc. USENIX Annual Technical Conf. (ATC-15)*, pages 57–69, 2015.

[91] Ailing Yu, Yujuan Tan, Congcong Xu, Zhulin Ma, Duo Liu, and Xianzhang Chen. DFShards: Effective construction of MRCs online for non-stack algorithms. In *Proc. the 18th ACM intl. Conf. on Computing Frontiers 2021 (CF'2021)*, page 63 – 72, 2021.

[92] Rongshang Li, Yingtian Tang, Qiquan Shi, Hui Mao, Lei Chen, Jikun Jin, Peng Lu, and Zhuo Cheng. Accurate probabilistic miss ratio curve approximation for adaptive cache allocation in block storage systems. In *Proc. the 2022 Design, Automation and Test in Europe Conf. and Exhibition (DATE'22)*, page 1197 – 1202, 2022.

[93] Yuchen Wang, Junyao Yang, and Zhenlin Wang. Multi-tenant in-memory key-value cache partitioning using efficient random sampling-based LRU model. *IEEE Transactions on Cloud Computing*, 11(4):3601–3618, 2023.

[94] Jun Xiao, Yaocheng Xiang, Xiaolin Wang, Yingwei Luo, Andy Pimentel, and Zhenlin Wang. FLORIA: A fast and featherlight approach for predicting cache performance. In *Proc. 37th intl. Conf. on Supercomputing*, pages 25–36, 2023.

[95] Mark D Hill and Alan Jay Smith. Evaluating associativity in CPU caches. *IEEE Trans on Computers*, 38(12):1612–1630, 1989.

[96] Moinuddin K Qureshi and Yale N Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. 39th intl. Symp. on Microarchitecture (MICRO'06)*, pages 423–432, 2006.

[97] Weiming Zhao and Zhenlin Wang. Dynamic memory balancing for virtual machines. In *Proc. 2009 ACM SIGPLAN/SIGOPS Intl. Conf. on Virtual Execution Environments (VEE'09)*, page 21–30, 2009.

[98] Ioan Stefanovici, Eno Thereska, Greg O'Shea, Bianca Schroeder, Hitesh Ballani, Thomas Karagiannis, Antony Rowstron, and Tom Talpey. Software-defined caching: Managing caches in multi-tenant data centers. In *Proc. Symp. on Cloud Computing (SoCC'15)*, pages 174–181, 2015.

[99] Ricardo Koller, Ali José Mashtizadeh, and Raju Rangaswami. Centaur: Host-side SSD caching for storage performance control. In *Proc. Intl. Conf. on Autonomic Computing (ICAC'15)*, pages 51–60, 2015.

[100] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *Proc. 13th Symp. on Networked Systems Design and Implementation (NSDI'16)*, pages 379–392, 2016.

[101] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. mPart: Miss-ratio curve guided partitioning in key-value stores. In *Proc. Intl. Symp. on Memory Management (ISMM'18)*, pages 84–95, 2018.

[102] Song Liu, Chen Zhang, Shiqiang Nie, Keqiang Duan, and Weiguo Wu. PC-Allocation: Performance cliff-aware two-level cache resource allocation scheme for storage system. *Applied Sciences*, 13(6):3556, 2023.

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

[104] Donald R. Slutz and Irving L. Traiger. A note on the calculation of average working set size. *Communications of the ACM*, 17(10):563–565, 1974.

[105] M.C. Easton and B.T. Bennett. Transient-free working-set statistics. *Communications of the ACM*, 20(2):93 – 99, 1977.

[106] P.J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, 1980.

[107] Ashutosh S Dhodapkar and James E Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proc. 29th Intl. Symp. on Computer Architecture (ISCA'02)*, pages 233–244, 2002.

[108] Sameh Elnikety, Steven Dropsho, and Willy Zwaenepoel. Tashkent+: memory-aware load balancing and update filtering in replicated databases. In *Proc. 2nd ACM SIGOPS/EuroSys European Conf. on Computer Systems 2007 (EuroSys'07)*, page 399–412, 2007.

[109] Changwoo Min, Inhyuk Kim, Taehyoung Kim, and Young I.K. Eom. Hardware assisted dynamic memory balancing in virtual machines. *IEICE Electronics Express*, 8(10):748 – 754, 2011.

[110] Xiaoya Xiang, Bin Bao, Chen Ding, and Yaoqing Gao. Linear-time modeling of program working set in shared cache. In *Proc. 2011 Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'11)*, pages 350–360, 2011.

[111] Lei Cui, Jianxin Li, Tianyu Wo, Bo Li, Renyu Yang, Yingjie Cao, and Jinpeng Huai. HotRestore: A fast restore system for virtual machine cluster. In *Proc. 28th Large Installation System Administration Conf. (LISA'2014)*, pages 10–25, 2014.

[112] Aparna Mandke Dani, Bharadwaj Amrutur, and Y.N. Srikant. Toward a scalable working set size estimation method and its application for chip multiprocessors. *IEEE Trans. on Computers*, 63(6):1567 – 1579, 2014.

[113] Kishore Kumar Pusukuri. Working set model for multithreaded programs. In *Proc. 2014 Conf. on Timely Results in Operating Systems (TRIOS'2014)*, 2014.

[114] Jui-hao Chiang, Tzi-Cker Chiueh, and Han-Lin Li. Memory reclamation and compression using accurate working set size estimation. In *Proc. 2015 IEEE 8th Intl. Conf. on Cloud Computing (CLOUD'15)*, pages 187–194, 2015.

[115] Dulcardo Arteaga, Jorge Cabrera, Jing Xu, Swaminathan Sundararaman, and Ming Zhao. CloudCache: On-demand flash cache management for cloud computing. In *Proc. 14th USENIX Conf. on File and Storage Technologies (FAST'16)*, pages 355–369, 2016.

[116] Peter J Denning. Working set analytics. *Computing Surveys*, 53(6):1–36, 2021.

[117] Weiming Zhao, Xinxin Jin, Zhenlin Wang, Xiaolin Wang, Yingwei Luo, and Xiaoming Li. Low cost working set size tracking. In *Proc. 2011 USENIX Annual Technical Conf. (ATC'11)*, 2011.

[118] Yousra Chabchoub and Georges Heébrail. Sliding hyperloglog: Estimating cardinality in a data stream over a sliding window. In *Proc. Intl. Conf. on Data Mining*, pages 1297–1303, 2010.

[119] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S Berger. Caching with delayed hits. In *Proc. Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM'20)*, pages 495–513, 2020.

A Artifact Appendix

A.1 Abstract

The TTLs Matter artifact provides our implementations of the following algorithms:

- Olken's MRC-generation algorithm [25]
  - TTL-agnostic **Olken**
  - TTL-aware **Olken**$^{++}$
- Fixed-rate SHARDS MRC-generation algorithm [29]
  - TTL-agnostic **FR-Shards** and **FR-Shards**$_{adj}$
  - TTL-aware **FR-Shards**$^{++}$ and **FR-Shards**$^{++}_{adj}$
- Fixed-size SHARDS MRC-generation algorithm [29]
  - TTL-agnostic **FS-Shards** and **FS-Shards**$_{adj}$
  - TTL-aware **FS-Shards**$^{++}$ and **FS-Shards**$^{++}_{adj}$
- The HyperLogLog (HLL) cardinality estimator [56]
  - TTL-agnostic **HLL**
  - TTL-aware **HLL-TTL**
- CounterStacks MRC-generation algorithm [28]
  - TTL-agnostic **CounterStacks**
  - TTL-aware **CounterStacks**$^{++}$

A.2 Description & Requirements

The README associated with the artifact includes details on compilation and usage.

A.2.1 How to access.

- Repository: https://github.com/SariSultan/TTLsMatter-EuroSys24
- Zenodo: https://doi.org/10.5281/zenodo.10783873

A.2.2 Hardware dependencies. Our experiments were conducted on a server equipped with an Intel 13900KS CPU and 128GB of DDR5-4800MHz DRAM.

A.2.3 Software dependencies. The artifact depends on `dotnet sdk v8.0.100`. Generating the plots depends on `Gnuplot v5.4+`. The artifact was tested on a server running `Ubuntu 23.04`.

A.2.4 Benchmarks. Our evaluation utilized 28 workloads from the Twitter dataset [23]. The 28 workloads are: 4, 6, 7, 8, 11, 13, 14, 16, 18, 19, 22, 24, 25, 29, 30, 33, 34, 37, 40, 41, 42, 43, 46, 48, 49, 50, 52, 54. The workloads can be downloaded from https://github.com/SariSultan/TTLsMatter-EuroSys24.

A.3 Set-up

To compile the source code, go to the directory "src/TTLsMatter" in the artifact, and execute the script "./publish.sh". This will create a compiled binary file called "TTLsMatter" inside the directory "src/TTLsMatter/bin".

A.4 Evaluation workflow

A.4.1 Major Claims.

- *(C1): Significant memory savings can be achieved when sizing caches using TTL-aware Working Set Sizes (WSSs) and Miss Ratio Curves (MRCs).*
- *(C2): Existing WSS and MRC algorithms, which do not take TTL into account, are highly inaccurate when applied to workloads with TTLs.*
- *(C3): The performance of our TTL-aware algorithms are comparable to that of the existing TTL-agnostic algorithms.*
- *(C4): Our extended algorithms maintain consistent accuracy across varied configuration parameters.*

A.4.2 Experiments. We have streamlined the process of reproducing all the results from our evaluation section (§6) by running a single command.[14] This will also reproduce the figures from the evaluation section, namely, figures 14, 15, 16, 17-(a), 17-(b), and 18, which support the aforementioned claims. These claims are further discussed in Sections 6.1 and 6.2.

After compiling the source code as described earlier, run "./TTLsMatter" inside the "bin" directory. The figures will be generated in a new directory named "Figures". The WSS results will be generated in a directory named "WSS-AE-2024". The MRC results will be generated in a directory named "MRCs-AE-2024". The estimated compute time for this experiment is 5 days.

[14] Comprehensive details are provided in the artifact's GitHub repository README, under "Reproducing the evaluation section results"