Configuring In-Memory Caches: From TTL-Aware Sizing to Interval-Based Historical Analysis with HistoChron

by

Sari Sultan

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy

The Edward S. Rogers Sr. Department of Electrical & Computer Engineering
University of Toronto

Configuring In-Memory Caches: From TTL-Aware Sizing
to Interval-Based Historical Analysis with HistoChron

Sari Sultan
Doctor of Philosophy
The Edward S. Rogers Sr. Department of Electrical & Computer Engineering
University of Toronto
2024

Abstract

In-memory caches such as Memcached and Redis are crucial for enhancing the performance of distributed systems by significantly reducing query response times. Correctly sizing these caches is critical, especially considering that prominent organizations use terabytes to petabytes of Dynamic Random Access Memory (DRAM) for these caches. Configuring these caches to operate efficiently remains a challenging task, considering the dynamic nature of modern workloads where caching requirements can change significantly over time.

Our thesis is that *the state-of-the-art for in-memory cache performance analysis does not accommodate modern workloads.* This gap is evident in the lack of consideration for Time-to-Live (TTL) attributes and heterogeneous object sizes, as well as the absence of interval-based historical analysis to address the dynamic nature of these workloads. This dissertation introduces a comprehensive reevaluation of in-memory cache performance analysis tools. We propose novel tools that account for TTL attributes and heterogeneous object sizes, and we introduce a new tool that enables efficient interval-based historical analysis of in-memory cache workloads. In particular, one of our primary contributions is the development of Miss Ratio Curve (MRC) generation and Working Set Size (WSS) estimation algorithms that accommodate TTL attributes and heterogeneous object sizes. Our analysis of real-world cache workloads demonstrates that including TTLs can lead to an average reduction in cache memory footprint by 69%, and up to 99%.

Additionally, we introduce HistoChron, a novel methodology with a Graphical User Interface (GUI) that enables efficient interval-based historical analysis of caching workloads. Evaluated on over $5,000$ cache access traces from six real-world datasets, encompassing more than 300 billion accesses over an 18-year span, HistoChron demonstrates its efficacy by generating exact MRCs over any arbitrary time interval using just 24MiB of storage space weekly. We also present a lower-overhead variant of HistoChron that generates approximate results with a mean error of less than 1%. These contributions advance the field of in-memory cache management, offering a robust framework for optimizing in-memory caches in alignment with the dynamic demands of modern workloads.

"A man may say with some colour of truth that there is an Abecedarian ignorance that precedes knowledge, and a doctoral ignorance that comes after it: an ignorance that knowledge creates and begets, at the same time that it despatches and destroys the first." — Montaigne

Acknowledgements

· In the Name of God, the Merciful, the Compassionate ·

· • ·

Prophet Mohammad, peace and blessings be upon him, said:

"He who does not thank the people is not thankful to Allah."[1]

I would like to begin by thanking the many without whom this work would not have been possible. First, thanks to my family and friends for their support, patience, and encouragement. In the loving memory of my father, Mohammad M.A. A. Sultan Al-Tamimi [bin Tamim Al-Dari] (Jaffa, Palestine, 1947 - Jordan, 2020), who passed away during my PhD studies. His memory, along with the love and prayers of my mother, Adla Mukhlis Amin Sultan Al-Tamimi (Hebron, Palestine 1947), have been a guiding force in the arduous path of my PhD journey. By the end of my third year, I was involved in an accident that resulted in a temporary disability, which made me fear I might not be able to complete my PhD. I am grateful to my lawyers and recovery team during this challenging time, who were instrumental to my recovery. This accomplishment is as much yours as it is mine.

Second, I deeply appreciate the opportunity to study on this land, which has been home to many Indigenous people from across Turtle Island for thousands of years. As someone born and raised in a Palestinian refugee camp, whose parents and their families were afflicted by wars and displacement, I have a profound respect for the struggles and resilience of Indigenous people. This has significantly influenced my nuanced understanding of the importance of Indigenous rights, and human rights in general, which is a core part of my identity. I am also grateful to have become a Canadian citizen on Canada Day while finalizing this dissertation — a serendipitous timing that mirrors my birth on Jordan's Independence Day, thus interweaving my love for Jordan and Canada. Thanks to everyone who stands for these rights, from Canada to Palestine and across the world.

Third, thanks to my supervisor, Prof. Michael Stumm, for his support and guidance throughout this PhD journey. My PhD studies were funded by his research lab and the ECE Department through research and student fellowships, and my TA jobs. Thanks to the supervisory committee, Profs. Ashvin Goel and Ding Yuan, who provided valuable feedback that improved this research. Thanks to the examination committee and their helpful feedback as well, which included, in addition to the supervisor and supervisory committee, Prof. David Lie, and the external examiner, Prof. Samer Al-Kiswany from the University of Waterloo. Thanks to the comprehensive examination committee as well, Profs. Andreas Moshovos, Baochun Li, and David Johns. I also would like to thank the colleagues and collaborators from Stumm's research lab and Huawei.

Fourth, my MSc and BSc studies played a significant role in preparing me for my PhD. Thanks to Kuwait University for funding the MSc studies through the Excellence Scholarship. Thanks to the MSc thesis supervisor, Prof. Ayed Salman, who helped significantly improve my practical research skills. Thanks to the MSc committee members, Profs. Imtiaz Ahmad and Fawaz Al-Anzi, for their feedback. Moreover, thanks to Profs. Sa'ed Abed, Mohammad Alshayeji, Mohamad Awad, Tassos Dimitriou, and Sabah Al-Fedaghi for their guidance.

Lastly, for my BSc studies at the Hashemite University, thanks to my teachers who helped in my early academic journey, including Profs. Bassam Jamil, Thaier Hayajneh, Ghada Al-Mashaqbeh, Samer Khasawneh, Jamal Al-Karaki, and Sari Awwad. My first experiences in scientific research under the guidance of Profs. Awni Itradat (the BSc graduation project supervisor) and Sa'ed Abed were valuable experiences, and I am thankful for their support and encouragement.

[1]Source: Sunan Abi Dawud, Hadith Number 4811.

Contents

List of Tables

List of Figures

Chapter 1

Introduction and Motivation

"We are therefore forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible."

— In Preliminary Discussion of the Logical Design of Electronic Computing Instrument, 28 June 1946 [1]

The advent of modern computing has fundamentally transformed modern societies by embedding many computer systems into numerous aspects of modern life. An intricate relationship exists at the heart of computer systems between two important components, i.e., the processor and memory. Computations are performed by the processor using instructions and data stored in the system's memory. The overall performance of a system primarily depends on the speed of the processor and memory. If the memory provides data instantaneously, then the processor is the main determining factor of the system performance. However, this ideal situation is not the case because technological constraints have led to a performance gap between the processor and memory. This gap was first noted by the design team of the first general-purpose digital computer, Electronic Numerical Integrator and Computer (ENIAC), who recognized that this performance gap between the processor and memory could significantly bottleneck the system performance [1, 2]. This insight led them to develop the memory hierarchy model in the 1940s to balance speed, storage capacity, and cost efficiency to optimize the interactions between the processor and memory. The memory hierarchy model remains integral to virtually all computer systems nowadays. It is based on the premise that not all data can reside within the processor and must therefore be stored away from the processor.

The memory hierarchy model systematically organizes different storage technologies based on their access speed, capacity, and cost, as shown in Fig. 1.1. The model covers fast and expensive technologies like processor registers and caches as well as slower and cheaper alternatives such as Hard Disk Drives (HDDs). Despite of the technological advances in the last 70 years, a significant gap persisted between the speed of processors and memory access time. The performance of processors has dramatically improved while the performance of memory technologies like Dynamic Random Access Memory (DRAM), Solid State Drives (SSDs), and HDDs, did not keep up with processor speeds, though expanding in capacity [3–5]. For instance, accessing a 4KiB object from DRAM has a latency of 100 nanoseconds, while accessing the same object from an SSD has a 100 microseconds latency (i.e., 1,000 times slower), and an HDD exacerbates the latency to 10 milliseconds, i.e., a 100,000 times increase [5]. This gap has made the data location a crucial factor to determine the performance of modern applications, especially as they become increasingly data intensive [4]. Hence,

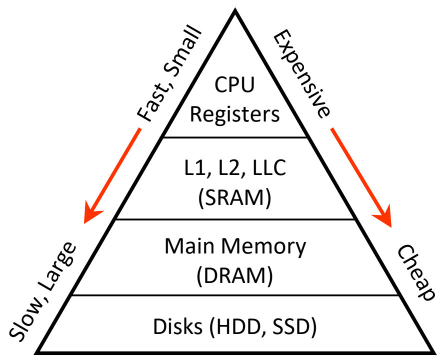


Figure 1.1: Illustration of the memory hierarchy model, showing the trade-off between memory speed, capacity, and cost at different levels.

optimizing data access in the memory hierarchy has been a significant area of research in academia and industry to mitigate the impact of latency on performance. For example, notable impacts of latency on major online platforms include: Amazon's reported 1% sales drop per 100ms increase in latency, Google's 20% traffic reduction from an extra half-second delay, Bing's 4.3% revenue decrease from a 2-second slowdown, and Shopzilla's 25% increase in page views and 12% revenue boost from few seconds latency improvements [6–10].

The *principal of locality* is widely used in modern computing to optimize data access in the memory hierarchy model. Simply put, it posits that recently accessed data is likely to be accessed again soon (temporal locality), while data near recently accessed data is likely to be accessed again soon (spatial locality) [11–14]. Thus, each level in the memory hierarchy model can be optimized using this insight through a mechanism called *caching*, which aims to keep a smaller subset of the data expected to be reused soon in a faster level in the memory hierarchy. By keeping data closer to the processor, data accesses can be streamlined, which improves their access speed. Since 1940s, caching has become a fundamental aspect of computing as it is employed in a myriad of applications, including but not limited to, databases, file systems, and web servers [15–21]. Virtually all Central Processing Units (CPUs), Graphical Processing Units (GPUs), network devices, HDDs, and SSDs also feature hardware caches.

The main concept behind caching is that when a processor wants to access data from a slow level in the memory hierarchy, it first requests the data from a cache stored in a faster level. A cache hit occurs if the data exists in the cache, which results is faster access than going to the slower level. If the data was not found in the cache (a cache miss), then the data can be retrieved from a slower level in the hierarchy. Depending on the cache's configuration (§2.1), a copy of the data might be stored in the cache to speed up future accesses. A metric called the *miss ratio* is used to evaluate the efficiency of caches, which is the ratio between accesses that result in cache misses and the total number of accesses. The lower the miss ratio is, the higher the efficiency. Several factors affect the miss ratio such as the size of the cache and the eviction policy used. Since the size of the cache is limited (typically smaller than the accessed dataset), a mechanism is needed to manage the contents of the cache, which is the *raison d'etre* of the eviction policy. The most widely used eviction policy is Least Recently Used (LRU) because it balances simplicity and effectiveness [22–24]. The role of caching is expected to become more pronounced in the future given the exponential growth of in-memory datasets, where their size doubles approximately every two years [4, 25–27].

The scope of the memory hierarchy model has significantly expanded with the advent of the Internet and cloud computing, with new technologies such as cloud storage [28–30]. This expansion has shifted the focus from intra-system optimization, where the goal was optimizing resources within a single physical system, to inter-system optimizations within distributed systems. Modern applications are commonly deployed on cloud platforms utilizing microservices and serverless tasks that use external data stores (e.g., a database server), which can reside on the same physical machine or in a data center halfway across the globe. Thus, it becomes crucial to optimize both local and remote data accesses, such as optimizing access to remote cloud storage solutions. The aforementioned advancements decoupled data storage from local hardware limitations and allowed the dynamic on-demand scaling of resources by leveraging data stored in different locations.

The evolution towards distributed systems has also emphasized the importance of a new type of caches called *software in-memory caches*, which is the focus of this dissertation. In-memory caching technologies like Memcached [31, 32] and Redis [33] play a pivotal role in mitigating the inherent latency of accessing data from slower levels in the memory hierarchy by utilizing DRAM. They reduce response times by serving data from the cache's DRAM instead of from back-end storage systems, as well as by offloading back-end storage systems themselves. In-memory caches differ from hardware caches in their elasticity; they can be started or stopped within seconds using dynamically chosen sizes, flexibility not typically present in hardware caches. In-memory caches have become indispensable in modern distributed systems as evidenced by their wide use and the fact that all major cloud providers offer them as a service (e.g., Amazon ElastiCache, Google Memory Store, Microsoft Azure Cache, Huawei DCS, Oracle Cloud Cache) [9, 10, 34–49].

Accurate cache sizing is pivotal for operational efficiency and cost management; the pricing model for cloud in-memory caches is determined by the size of the cache and the duration of its lease. Inadequate cache sizing leads to operational inefficiencies with concomitant increased response times, while over-provisioning can unnecessarily inflate costs. Memory, particularly DRAM, constitutes 30% – 60% of infrastructure and operational costs [26, 27, 50], and is responsible for 25% – 40% of data center power consumption [51, 52]. Major organizations, including $\mathbb{X}$ (formerly Twitter), Google, Meta (formerly Facebook), among others, deploy terabytes and even petabytes of DRAM for in-memory caches, involving substantial financial investments and environmental implications [9, 10, 34–39]. Inefficient cache sizing not only results in poor resource utilization and escalated expenses, but also significantly contributes to the carbon footprint of these large-scale operations.

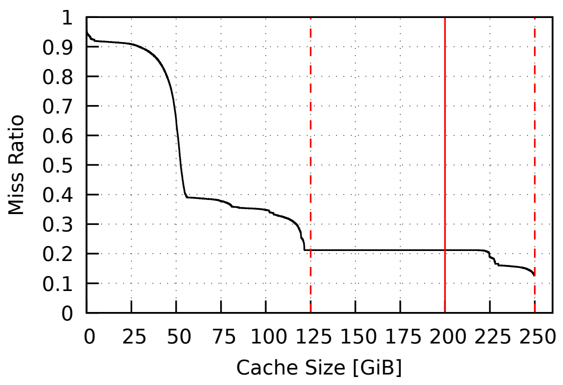


Figure 1.2: Miss ratio curve for the IBM workload #079.

How to Size In-Memory Caches?

Although in-memory caches have been used productively for decades, the best approach to determine the right cache size often remains unclear. As such, in-memory caches are often treated as black boxes, using coarse trial-and-error methods to improve their efficiency, or the caches are simply configured with a large amount of memory, hoping for the best [27, 50, 53–59]. In practice, many organizations simply over-provision the size of their caches [39, 58, 59].[1]

More strategic approaches to size in-memory caches utilize tools like the **Working Set Size (WSS)**, which measures the aggregate size of unique objects accessed within a specific interval [11, 14, 60]. By allocating a cache size equivalent to the WSS, one guarantees the achievement of the minimal miss ratio possible because all accessed objects can be cached. However, the WSS does not offer insights into the trade-off between cache size and miss ratio, thus limiting its usefulness to cases where the WSS is relatively small. This is expected to be further exacerbated by the exponential growth of in-memory datasets [25–27].

On the other hand, the **Miss Ratio Curve (MRC)** provides more insights into the trade-off between cache size and miss ratio because it plots the miss ratio as a function of the cache size. The MRC is the most widely studied tool to evaluate cache size trade-offs [9, 55, 56, 61–81]. As an example of an MRC and how it can be used to size caches, Fig. 1.2 shows the MRC for the in-memory cache workload #079 from IBM [22] under the LRU eviction policy. Assuming the cache is configured with 200GiB of memory, the MRC shows we can reduce the cache size to 125GiB with minimal effect on the miss ratio. Alternatively, the cache size can be increased to 250GiB for a 9% reduction in the miss ratio.

[1]In private communication with the authors of the Twitter paper that recently released in-memory cache workloads [39], they informed us, *"The production cache size cannot be disclosed and is often much larger than the working set size."* indicating significant over-provisioning.

1.1 Thesis Statement and Contributions

Despite the wealth of research on the WSS and MRCs since the 1970s [11, 14, 55, 56, 60–64, 66, 68, 69, 71, 75–77, 82–106], existing methods for cache performance analysis fall short in properly modeling modern in-memory cache workloads. The focus of historical studies has been on hardware caches and has not evolved in step with the advent of software in-memory caching, which has significantly transformed the computing landscape over the last 30 years. This transformation has engendered software in-memory caches such as Memcached and Redis, but the performance analysis tools designed for hardware caches were inherited and applied to in-memory caches without necessary modifications, leading to a critical misalignment in cache performance analysis. **Our thesis is that the state-of-the-art for in-memory cache performance analysis does not accommodate modern in-memory cache workloads.** This underscores the importance of a comprehensive reevaluation of performance analysis tools to accommodate modern in-memory cache workloads. To support our thesis, we highlight three primary gaps in the state-of-the-art.

1. Firstly, the state-of-the-art is oblivious to **Time-to-Live (TTL)** attributes, which are widely used for managing object expiry in in-memory caches, indicating a gap in understanding the impact of TTLs on cache modeling.

2. Secondly, although some research since the 1970s has considered the impact of **heterogeneous object sizes** within caches, the predominant trend in contemporary studies leans towards the simplification of using uniform object sizes. This trend overlooks the complex dynamics introduced by heterogeneous sizes, leading to a gap in fully comprehending their effect on cache performance. Despite the historical acknowledgment of object size variability, current methodologies often fail to incorporate or accurately model this variability, diminishing the applicability of such analyses to real-world scenarios where object sizes significantly vary.

3. Lastly, the dynamic nature of workloads, characterized by fluctuating caching requirements, poses a challenge not adequately addressed by current methodologies. These methodologies lack efficient techniques for **interval-based historical analysis of cache workloads**, thus complicating the diagnosis of issues and their causes in operational settings.

Addressing these gaps is crucial for advancing in-memory cache performance analysis and ensuring that in-memory caching systems are optimized for the dynamic demands of modern workloads. We briefly elaborate on each of the aforementioned points.

TTLs Matter

In-memory caches utilize TTL attributes to define the lifespan of cached objects, a feature essential for ensuring data freshness and adhering to regulations such as the General Data Protection Regulation (GDPR) [39]. TTLs are essential to improve the efficiency of in-memory caches [39, 107–109]. Despite the importance of TTLs in practice, the state-of-the-art in cache performance analysis remains oblivious to TTLs. More specifically, none of the existing WSS and MRC tools accommodate TTLs.

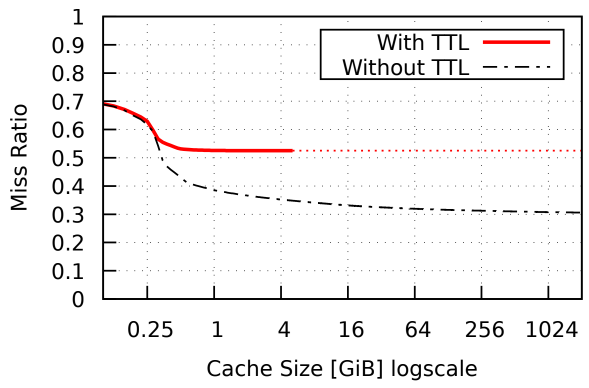


Figure 1.3: MRCs for the Twitter recommended workload #37: With TTL, the curve reaches steady-state at 5GiB, where the optimal miss ratio is achieved; without TTL, the curve reaches steady-state beyond 2TiB.

We have found that TTL attributes can significantly impact WSSes and MRCs. For example, Fig. 1.3 shows that neglecting TTLs when generating MRCs can result in MRCs that are substantially inaccurate. The figure shows two MRCs for Twitter's recommended workload #37 [110]: one taking TTLs into account and the other not. The cache size needed to achieve the minimal miss ratio is 5GiB when TTLs are taken into account, but it increases beyond 2TiB when they are not.

Taking TTLs into account in MRC generation involves tracking the expiry of objects while processing the workload's access stream or trace. This means that expired objects are proactively removed to ensure they do not affect the caching requirements depicted on the MRC. This tracking process does introduce some overhead to MRC generation, which is thoroughly explored in Chapter 3.

In Chapter 3, we also show that based on real-world workloads, when sizing caches using TTL-aware WSSes and MRCs, the cache size can be reduced by an average of 69% (and up to 99%) without a degradation in the miss ratio, compared to when using TTL-agnostic ones (i.e., the current state-of-the-art). Fig. 1.3 also shows that the best achievable miss ratio is 30% when TTLs are neglected, while with TTLs, it is 52%. Thus, not taking TTLs into consideration deviates significantly from real-world caches, typically by reporting a lower miss ratio than what is achievable.

To illustrate the potential savings by accommodating TTLs, we present two types of cost considerations using 54 publicly available workloads from Twitter [39]. The first one is for the cost of DRAM modules[2] — a conservative estimate of potential savings as it does not account for operational costs, such as DRAM contributing to 25% – 40% of data centers power consumption [51, 52], and other component costs (with DRAM accounting for 30% – 40% of infrastructure costs [26, 27]). The second cost consideration is for the cost of renting a caching server; for illustration, we use the pricing from Azure Cache for Redis [112].[3] For each type of cost consideration, we examine *three scenarios* to demonstrate the potential for cost savings, as shown in Fig. 1.4: On the left, we show the cost of DRAM modules, while on the right, we show the rental cost.

[2]The cost assumptions are derived from the pricing of 1TiB of DRAM, approximately $5,000 [111].

[3]The hourly prices for Azure Cache for Redis are as follows (as of May 02, 2023): 6GiB ($0.554), 13GiB ($1.11), 26GiB ($0.2.218), 53GiB ($4.44), 120GiB ($10.04) [112].

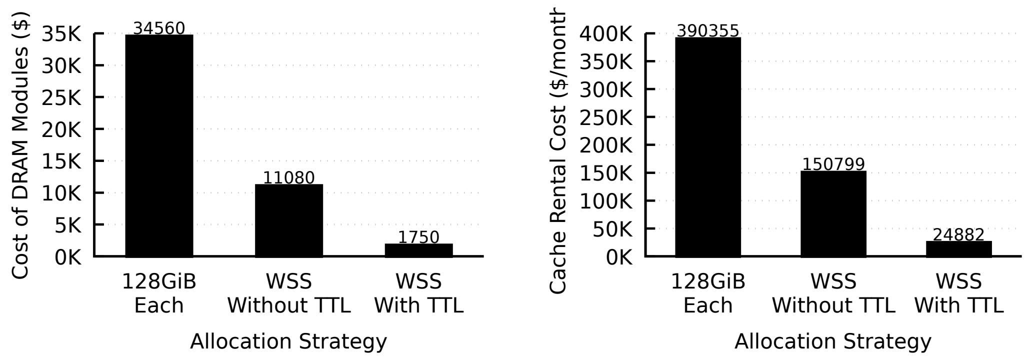


Figure 1.4: Cost savings comparison for different caching strategies illustrated through three bars: the leftmost bar represents the strategy of allocating 128GiB to each cache; the middle bar depicts the strategy of allocating memory equal to the WSS without considering TTLs; and the rightmost bar incorporates TTL considerations into the WSS allocation strategy.

First, allocating each workload to an over-provisioned caching server with 128GiB of DRAM results in a total DRAM requirement of 6,912GiB and a cost of \$34,560, with a monthly rental cost of \$390,355. *Second*, assigning each workload to a caching server with memory equal to the WSS of the workload (up to 128GiB for fairness), disregarding TTL values, reduces the total DRAM requirement to 2,216GiB, costing \$11,080 — a 67% cost reduction. Renting the smallest cache server to fit the WSS of each workload results in a monthly cost of \$150,799 — a 61% cost reduction. *Third*, using our WSS estimator (presented in Chapter 3), which accounts for TTL values, further decreases the aggregate memory requirement to 320GiB, incurring a cost of only \$1,750 — a remarkable cost reduction of 94%. Renting the smallest cache size to fit the WSS (with TTL) reduces the monthly cost to \$24,882 — a 93% cost reduction. These findings emphasize the potential impact of our approach on data center efficiency and operational expenditure in large-scale deployments, even when considering conservative estimates for cost savings.

Heterogeneous Object Sizes

The most widely used methods of modeling in-memory caches assume each accessed object has a uniform object size [55, 64, 113, 114]. This is understandable, as early on, MRCs were used to model caches for uniform disk blocks. However, today's in-memory caches allocate memory proportional to the size of the data being cached. Correspondingly, real-world workloads include accesses to heterogeneously sized objects [39, 56, 115–118]. For example, Figures 1.5 and 1.6 show the object size distributions for the Microsoft Research Cambridge (MSR) [115] and Twitter [39] workloads, respectively, illustrating the variability in object sizes used.

The effect of modeling heterogeneous object sizes on WSSes and MRCs has not been clarified in the literature, partly because it is often neglected. In Chapter 4, we show that not taking heterogeneous object sizes into account can result in substantially inaccurate WSS estimates and MRCs. For example, Fig. 1.7 shows that using previous simplistic attempts to take heterogeneous object sizes into account leads to substantial errors in MRCs. Two simple strategies are prevalent in prior work for accommodating heterogeneous object sizes. The first approach simply assumes all objects have the same size. For instance, Waldspurger et al. assumed uniform 16KiB objects [55]. The second approach converts each access to an object into as many uniformly sized objects as needed

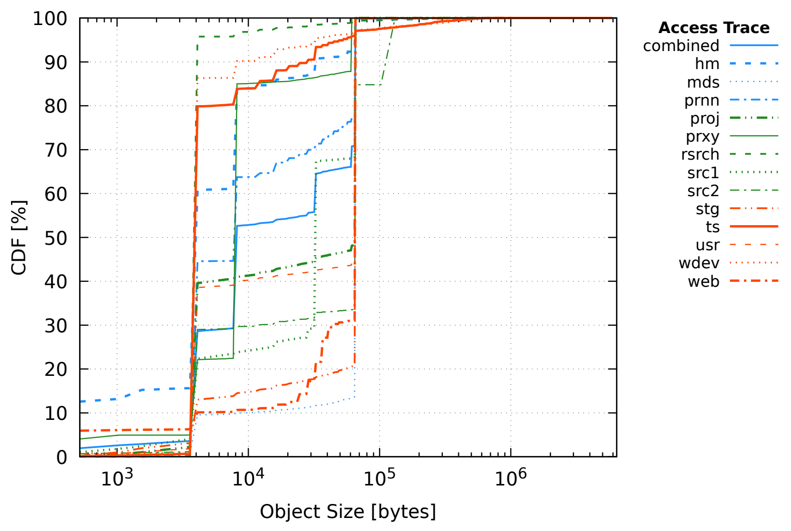


Figure 1.5: Cumulative distribution of object sizes for each trace in the MSR collection [115].

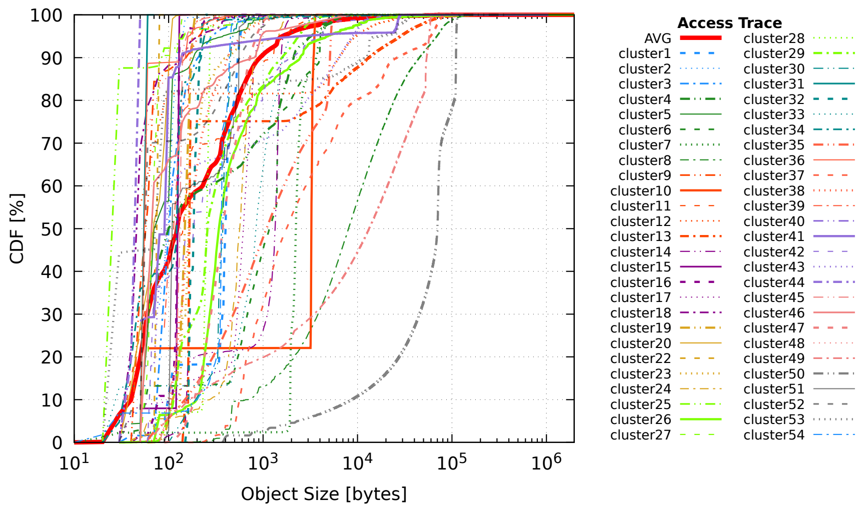


Figure 1.6: Cumulative distribution of object sizes for each trace in the Twitter collection [39]. The red curve represents the cumulative distribution of object sizes across all accesses in all traces.

to store the data belonging to the accessed object. For instance, an access to an 11KiB object would be converted to three different 4KiB accesses to what is assumed to be distinct objects. A similar approach is used by Wires et al. [64, 119].[4] This approach significantly increases the number of accesses and distinct objects accessed, and it also suffers from the issue of determining the base object size *a priori*. For example, the combined MSR workload contains 300 million accesses, but when converted to multiple 4KiB objects, the number of accesses becomes 2.1 billion. Moreover, although it has not been used in earlier studies, we have also added to Fig. 1.7 the MRC generated using the running average of object sizes, as we observed it can be computed efficiently online using only 16 bytes (§4). Chapter 4 describes how the state-of-the-art WSS estimation and MRC generation algorithms can be efficiently extended to support heterogeneous object sizes.

[4]The CounterStack paper (by Wires et al.) does not mention this explicitly, but Drudi's Master's thesis [120] does, and other researchers have also claimed that CounterStacks used 4KiB objects [55, 113].

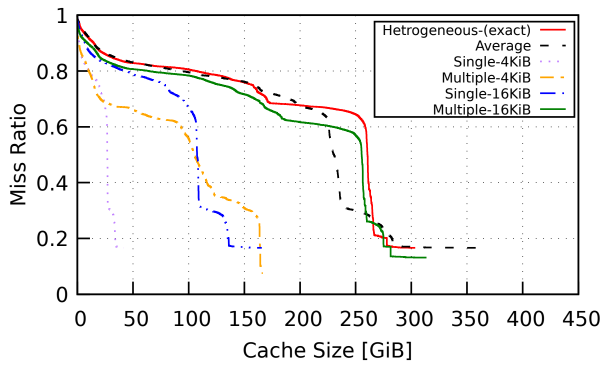


Figure 1.7: Effect of object size treatment on MRCs for the MSR workload `src1`.

Interval-Based Historical Analysis of Modern Workloads

Although the WSS and the MRC are useful for provisioning caches, the WSSes and MRCs can significantly change over time due to the dynamic nature of modern workloads [64, 106]. Therefore, analyzing the caching requirements over different time intervals is critical if resource efficiency is the goal. It is not uncommon for in-memory cache workloads to exhibit significant variability in their WSSes and MRCs when considering different time intervals. To illustrate this behavior, consider the real-world `prn` workload from the MSR dataset [115]. The MRC generated over the entire duration of this workload's access trace suggests that a cache size of 80GiB is needed to minimize the miss ratio (Fig. 1.8, top left). However, a more detailed, day-by-day analysis reveals severe fluctuations: from Friday to Sunday, less than 15GiB is needed to achieve the minimal miss ratio, but on Monday and Tuesday, the requirement jumps to 70GiB (Fig. 1.9). This shift is linked to a sudden potential scanning pattern observed at the end of Monday, involving accesses to a large number of new distinct objects (Fig. 1.8, top right). This sudden increase in accesses causes the WSS to spike from under 20GiB to 70GiB within just an hour (Fig. 1.8, bottom). Such patterns can obscure the true cache size requirements, as seen by the continued high cache size requirement into Tuesday caused by the displacement of hot objects in the cache. These scenarios, which we observed to be prevalent in modern workloads, highlight the importance of interval-based performance analysis.

The previously shown interval-based analysis cannot be extracted from the WSS/MRC generated for the entire duration of the workload's access trace. Prior to this work, the only one-pass algorithm that supported interval-based MRC generation was Counterstacks [64], proposed in 2014. This is achieved by periodically checkpointing the internal state of Counterstacks, and later on querying the outputted stream to generate the MRC over the desired interval. A year after Counterstacks was proposed, Waldspurger et al. proposed the SHARDS MRC-generation algorithm, which is more efficient than Counterstacks but does not support interval-based analysis [55]. To overcome this limitation, Waldspurger et al. suggested that multiple instances of SHARDS can be instantiated to generate the MRC over the desired periods. This is clearly inefficient as we must maintain $\frac{R\cdot(R+1)}{2}$ instances in order to study the possible ranges in the interval R.[5] For instance, to study the behavior for a week-long access trace with per-hour granularity, 14,196 SHARDS instances must be maintained. Regardless, both Counterstacks and SHARDS are approximate algorithms, and up to now, there has been no *exact* one-pass solution that supports interval-based analysis of caching workloads.

[5]For example, with 3 intervals between timestamps [1-4], we need 6 instances: [1-2] [1-3] [1-4] [2-3] [2-4], and [3-4].

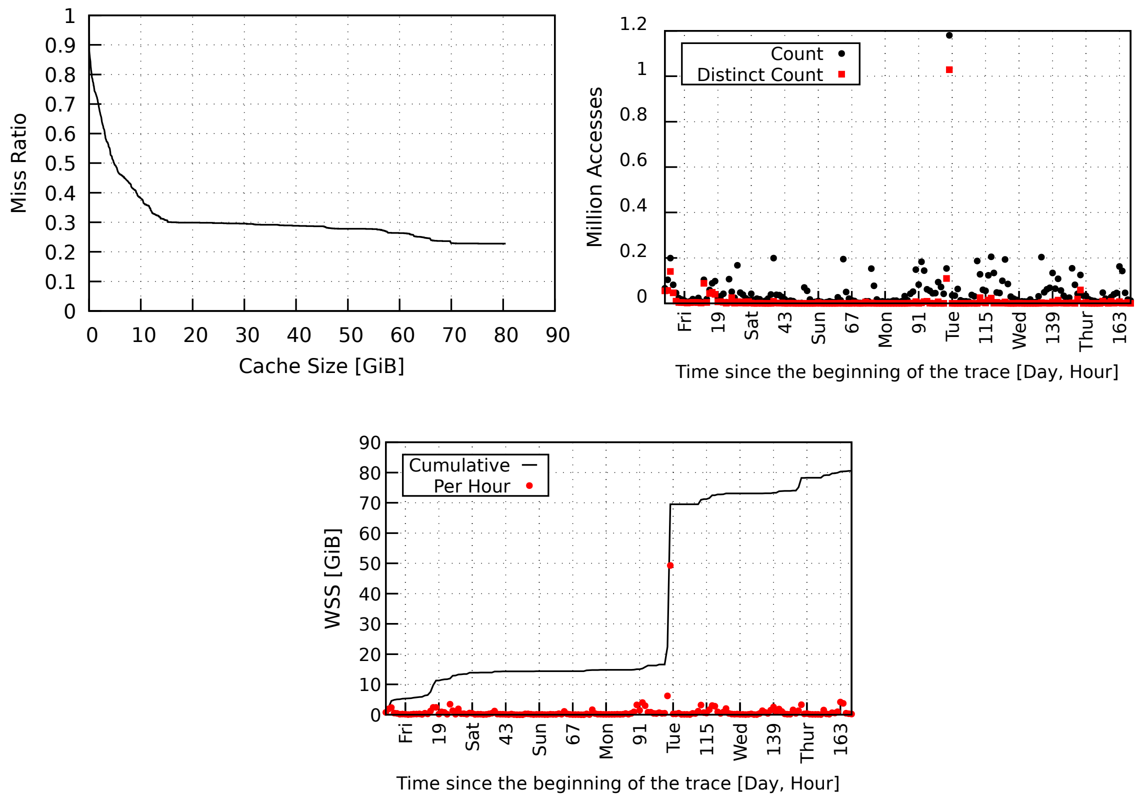


Figure 1.8: MSR `prn` workload. **Top Left**: MRC for the week. **Top Right**: Hourly access count statistics. **Bottom**: WSS.

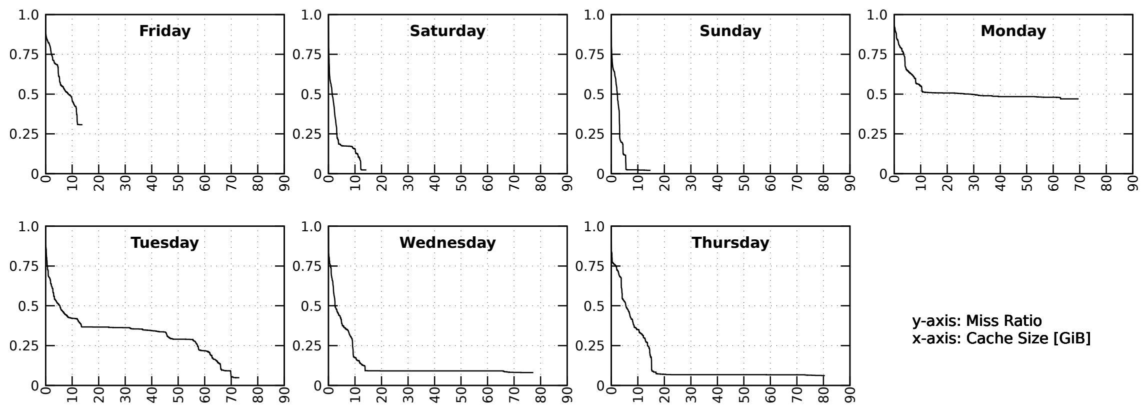


Figure 1.9: MSR `prn` workload: per-day MRCs. The curves end when increasing the cache size no longer reduces the miss ratio.

Addressing these limitations, this dissertation introduces two advances in Chapter 5. Firstly, we present the first *exact* one-pass algorithm for interval-based analysis of caching workloads, which we call HistoChron. HistoChron consumes just 24MiB of storage space weekly, showcasing its space efficiency in generating interval-based statistics such (i) the number of access requests, (ii) the number of objects accessed the very first time, (iii) the number of distinct objects accessed, (iv) the MRC, and (v) the WSS. This information enables identifying seasonal or diurnal patterns, and it enables detailed postmortem analysis of unexpected cache behaviors. Secondly, we adapt the SHARDS MRC-generation algorithm to enable efficient approximate interval-based analysis capabilities while using a single instance of SHARDS (compared to the initially proposed method of maintaining multiple instances). Together, these contributions mark a milestone in the field of cache performance analysis, providing robust tools for precise and efficient evaluation of dynamic caching workloads.

Contributions

In summary, this dissertation makes the following primary contributions. To the best of our knowledge, we are the first to:

- show that taking TTL attributes into account can significantly affect WSS estimates and MRCs;
- present exact one-pass MRC-generation algorithms that support TTLs (i.e., Mattson^{++} and Olken^{++});
- present approximate one-pass MRC-generation algorithms that support TTLs (i.e., SHARDS^{++} and CounterStacks^{++});
- extend the HyperLogLog (HLL) cardinality estimator to support evictions based on TTLs (i.e., HLL-TTL);
- show how heterogeneous object sizes affect WSS estimates and MRCs;
- show how Mattson, Olken, and SHARDS can be extended to support heterogeneous object sizes;
- extend the HLL cardinality estimator to support heterogeneous object sizes;
- extend the CounterStacks MRC-generation algorithm to support heterogeneous object sizes;
- and introduce a new tool (HistoChron) with an associated GUI that enables historical interval-based analysis of cache workloads.

We evaluated our TTL-aware algorithms on a large set of publicly available access traces and show that cache sizing with TTL-aware WSS estimates and MRCs leads to a 69% lower cache memory footprint on average (and up to 99%) without degrading the cache's miss ratio. We also evaluated HistoChron using over 5,000 cache access traces from six real-world datasets encompassing more than 300 billion accesses and, when combined, span a total of 18 years of cache accesses. We show that HistoChron provides exact interval-based MRCs while consuming 24MiB of storage space per week with overheads on par with state-of-the-art exact MRC-generation algorithms.

The following publications resulted from my work and collaborations during my PhD so far.

- **Sari Sultan**, Kia Shakiba, Albert Lee, Michael Stumm, Ming Chen, and Chung-Man Chow. Systems and Methods to Generate a Miss Ratio Curve where Cache Data has a Time-To-Live. U.S. Patent Application #17/982,136. Filed Oct 2022 [121].
- **Sari Sultan**, Kia Shakiba, Albert Lee, Michael Stumm, Ming Chen, and Chung-Man Chow. Systems and Methods to Generate a Miss Ratio Curve for a Cache with Variable-Sized Data Blocks. U.S. Patent Application #17/982,070. Filed Oct 2022 [122].
- **Sari Sultan**, Kia Shakiba, Albert Lee, Paul Chen, Michael Stumm. TTLs Matter: Efficient Cache Sizing with TTL-Aware Miss Ratio Curves and Working Set Sizes. In Proc. European Conference on Computer Systems (EuroSys'24). pp. 387-404. 2024 [123].
  - Awarded the Gilles Muller Best Artifact Award[6]
- **Sari Sultan**, Kia Shakiba, and Michael Stumm. Interval-based Historical Analysis of Cache Workloads with HistoChron. (Paper draft)
- Kia Shakiba, **Sari Sultan,** and Michael Stumm. Kosmo: Efficient Online Miss Ratio Curve Generation for Eviction Policy Evaluation. In Proc. USENIX Conference on File and Storage Technologies (FAST'24). pp. 89-105. 2024 [124].

1.2 Organization

The remaining parts of this dissertation are organized as follows. Background information and related work are presented in Chapter 2. In Chapter 3, we show how to accommodate TTLs in WSS estimation and MRC generation algorithms. Chapter 4 describes how WSS estimation and MRC generation algorithms can be adapted to accommodate heterogeneous object sizes. Interval-based historical analysis of in-memory caching workloads with HistoChron is presented in Chapter 5. Lastly, Chapter 6 presents concluding remarks and future research directions.

[6]Webpage: 2024.eurosys.org/awards.html. Artifact source: github.com/SariSultan/TTLsMatter-EuroSys24

Chapter 2

Background and Related Work

This chapter provides background information on related research that is important to understand the subject matter of this dissertation. It begins with a brief overview of in-memory caches in Section 2.1. These caches are widely used for optimizing data access latency in distributed systems. The role of TTL attributes in in-memory caches is explored in Section 2.1.1. Section 2.1.2 explores the relevance of the object size in these caches.

The Miss Ratio Curve (MRC) is an instrumental tool for evaluating the trade-offs between cache size and miss ratio, and it plays a key role in our work. The significance of the MRC lies in its ability to guide the allocation of cache resources, ensuring efficient cache utilization. We provide a comprehensive review of MRC generation in Section 2.2. The chapter also examines the Working Set Size (WSS) in Section 2.3, another critical tool for understanding an application's memory demand, facilitating effective cache sizing decisions. The importance of WSS is that it provides the cache size needed to achieve the minimal miss ratio, thus ameliorating over-provisioning, as increasing the cache size beyond the WSS does not decrease the miss ratio.

Finally, Section 2.4 provides an overview of cardinality estimation (also known as probabilistic counting), a concept gaining prominence for its utility in both MRC generation and WSS estimation. Cardinality estimation techniques, such as the HyperLogLog (HLL) algorithm, are discussed for their efficiency in approximating the number of unique objects accessed in an access stream or trace, a key factor in analyzing cache behavior under varying workloads. Cardinality estimation is one of the most efficient techniques for estimating the WSS.

2.1 In-Memory Caches

Recall from the Introduction that extensive research has focused on optimizing data accesses within the memory hierarchy, given the latency gaps between different levels in the hierarchy. Such optimizations are crucial for enhancing system performance as they aim to minimize the time it takes for a processor or data consumer to access data. Caching is one of the primary mechanisms used to speed up data access among different levels in the memory hierarchy. As such, in-memory caches play a pivotal role in reducing access times by storing hot objects in DRAM to speed up subsequent accesses to these objects that would otherwise be stored in a back-end storage system.

Although the concepts presented herein may apply in principle to other types of caches, the focus of this dissertation is on ***software in-memory caches***, where in-memory refers to volatile memory or DRAM. In-memory caches are widely used in modern distributed systems; Memcached [31, 32] and Redis [33] are two popular examples. They aim to reduce query response times by serving data from an in-memory cache instead of from a back-end storage service. Numerous organizations use terabytes or even petabytes of DRAM for in-memory caches, including Google, Meta (formerly Facebook), 𝕏 (formerly Twitter), Pinterest, Airbnb, GitHub, and LinkedIn [9, 10, 34–39]. All major cloud providers offer software in-memory caches as a service [40–49].

The role of in-memory caches is illustrated in everyday scenarios, such as accessing a webpage. For instance, consider a user visiting a Wikipedia article that comprises numerous images. Without in-memory caching, whenever a user visits this article, it would require fetching the article's images from a back-end storage service. In-memory caches can significantly improve access time leading to an improved user experience by keeping a copy of the webpage's images in DRAM, thus trying to avoid requesting them from a slower back-end storage service.

In-memory caches typically operate on a key-value store model. The key is used to identify the cached data value, which we refer to as an ***object*** throughout the document. The two most common commands for in-memory caches are `GET` and `SET` (although modern in-memory caches support a wider range of commands):

- `GET` Command: When a user requests an operation from an application (e.g., access a Wikipedia article), the application first uses the GET command to request the required objects from the cache to complete the requested operation. If an object is present in the cache, it is quickly retrieved from DRAM and served to the application, significantly reducing response time. Otherwise, the application retrieves the object from the back-end storage service and serves it to the user (assuming it exists there). After retrieval, the application may decide to store the object in the cache using the `SET` command, readying it for quicker future access.

- `SET` Command: This command is used to store objects in the cache. Typically in-memory caches operate as a look-aside cache (also known as side cache), meaning they do not automatically fetch missing objects from the back-end storage service [10, 23, 125]. Instead, upon a cache miss, an in-memory cache simply informs the requesting application that the requested object is not available. The application then fetches the object from the back-end storage service and decides whether to `SET` a copy in the cache or not. In contrast, inline caches (e.g., Amazon DynamoDB Accelerator (DAX) [23]), automatically retrieve missing objects from the slower level in the memory hierarchy (if available). The look-aside model simplifies the cache design by decoupling it from the back-end storage interface, which could be a database or a computation service [125].

In-memory caches are elastic: their size is initially specified when instantiated, but they can be stopped and resized as needed. How effectively an in-memory cache operates depends on several factors, such as the eviction policy used and, more importantly, the amount of memory allocated to it [23]. Allocating too little memory results in higher miss ratios 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 given that organizations provision a lot of DRAM for these caches [9, 10, 34–39, 114] — in some cases, more than

Table 2.1: Azure's enterprise cache pricing for Redis (as of Feb 2024 obtained from [112])

Cache Name	Cache Size	Monthly Cost
E5	4GiB	$569
E10	12GiB	$975
E20	25GiB	$1,945
E50	50GiB	$3,822
E100	100GiB	$8,743
E200	200GiB	$17,816
E400	400GiB	$35,631

half of an application's operational costs [50]. As one point of reference, Table 2.1 shows the monthly pricing for Azure Cache for Redis: a single 100GiB cache costs $8,743 per month. The challenge, therefore, lies in determining an optimal cache size that balances efficiency with cost, a task that becomes increasingly complex in large-scale deployments. Correctly sizing these caches is critical, especially considering the fact that prominent organizations use terabytes and even petabytes of DRAM for in-memory caches.

2.1.1 Time to Live (TTL) Attributes

TTL attributes are prevalent in caching systems [39, 109, 114, 125–135]. They play a critical role in their management and efficiency [23, 39, 108, 114, 125]. TTLs ensure data stored in caches remains fresh and relevant. By assigning a finite lifespan to cached objects, TTLs help mitigate the risk of serving outdated information, a crucial aspect for dynamic environments where data updates are frequent. The TTL is typically set when the data is inserted into the cache and is measured in units of time, such as seconds. Once the TTL for a cached object expires, the object is considered stale and is removed [39, 125].[1] The significance of TTLs extends beyond data freshness, as they also impact cache eviction strategies and overall cache performance optimization [39, 107, 109, 125, 136].

TTLs serve various purposes in the management of in-memory caches [39, 109, 125, 133]:

- Implicit Deletion: TTLs facilitate automatic data removal, simplifying cache management. This is useful for temporary data or information with a predictable lifespan, such as session information in web applications. This is also important for compliance with privacy laws.
- Periodic Refresh: By setting appropriate TTL values, caches can ensure data is periodically refreshed. This is vital for services that depend on timely data but do not require consistency with instant updates for every change in the data source. This is also important for data that requires significant computational resources, wherein TTLs can help balance computational efficiency with data freshness.
- Bounding Inconsistency: TTLs limit the duration data can remain in the cache without being updated thus bounding the extent of data inconsistency. This is particularly important in dynamic environments where cache updates are best-effort.

The importance of TTLs is underscored by practices such as Twitter's TTL enforcement for in-memory caches [109, 125]. The rising significance of TTLs in storage systems, especially in the context of GDPR compliance, is also worth noting [133, 138].

[1] In-memory caches do not return data that has expired, even if it existed in the cache [109, 136, 137].

On the GitHub issues page for Memcached, where the maintainers were asked to consider changing the eviction policy from LRU-based to a more efficient eviction policy, the maintainers stated, *"pulling expired items out actively is better than almost any other algorithmic improvement I could think of"*, highlighting the importance of object expiry over eviction policy selection [107]. Further, although many in-memory caches used to evict objects lazily [139], 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 expiring objects more proactively [108]. 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 $1ms$ of their expiry [140, 141]. Similarly, several recent in-memory caches employ proactive object evictions [109, 136].

Despite the importance of TTLs in practice, the state-of-the-art in cache performance analysis remains oblivious to TTLs. Eviction policies used in modern caches are inherently TTL-aware, but the techniques used to analyze these policies neglect TTL attributes. One of the contributions of this dissertation bridges this gap by accommodating TTLs in the state-of-the-art of MRC generation and WSS estimation algorithms. The following sections provide background on state-of-the-art cache performance analysis techniques, setting the stage for the contributions presented herein.

2.1.2 Heterogeneous Object Sizes

In-memory caches such as Memcached and Redis allocate memory for each object in proportion to the size of the object being cached. This is a distinctive feature of modern in-memory caches compared to traditional hardware and disk block caches that cache uniformly sized objects or pages. Earlier in the Introduction, we showed the object size distribution for the MSR and Twitter datasets in Figures 1.5 and 1.6 to illustrate the wide range of object sizes used in modern workloads.

Accommodating heterogeneous object sizes introduces complexities in modern caches. For instance, Memcached uses a custom slab allocation system to handle heterogeneous objects, dividing memory into slabs of different sizes to improve memory allocation [32, 142]. This approach is necessary to efficiently manage varied object sizes, a task that would be straightforward if uniform sizes were used. Similarly, modeling heterogeneous object sizes presents significant challenges. In particular, the MRC and WSS can become skewed when object sizes are not taken into account, leading to inaccurate representations of cache performance. For example, earlier in the Introduction, we showed how neglecting the object size can significantly affect the MRC (Fig. 1.7). This dissertation introduces methods that accommodate heterogeneous object sizes in MRC generation and WSS estimation, providing more accurate insights into cache behavior.

2.2 Miss Ratio Curves (MRCs)

The MRC is the most effective tool for evaluating cache size trade-offs. It plots the cache miss ratio as a function of the cache size for a given workload under a specific eviction policy, as we illustrated earlier in the Introduction with Fig. 1.2. Over the last five decades, a wealth of MRC-generation algorithms have been proposed [55, 56, 61–64, 66, 69, 71, 75–77, 82–92, 106]. In this section, we focus on seminal algorithms that were considered breakthroughs in MRC-generation or where the efficiency of MRC-generation was improved significantly. Section 2.2.1 discusses Mattson's algorithm, which introduced the concept of *one-pass* MRC-generation in 1970. Virtually all one-pass MRC-generation nowadays rely on the approach proposed by Mattson et al., including, *inter alia*, [55, 62–64, 82, 113]. In Section 2.2.2, we present background on Olken's algorithm, which was introduced in 1981 and remains the most efficient *exact* one-pass MRC-generation algorithm today (see Table 2.2).

Although MRC-generation is widely used to optimize cache resources [9, 56, 65–78], exact MRC-generation algorithms such as Mattson and Olken are considered to be slow and inefficient. Due to their exactness, they require a significant amount of space that is proportional to the number of distinct objects accessed in an access stream or trace. This space requirement is typically not practical for online use cases. For instance, one workload required over 92GiB of DRAM and one hour of processing time to generate the MRC [120]. In our experiments, we used six publicly available datasets that contain over 300 billion accesses, and generating the exact MRCs for the access traces from these datasets consumed weeks of processing time for the LRU policy alone.

To address the inefficiency of exact MRC-generation algorithms, a plethora of *approximate* MRC-generation algorithms have been proposed with the common goal of reducing the space and computational complexities of MRC-generation in order to facilitate online applications [55, 64, 86, 88, 90, 113, 143]; applications that utilize an MRC generated online to make cache configuration decisions. Table 2.2 shows the key MRC-generation algorithms developed since the 1970s.

Here, we provide a comprehensive background on several state-of-the-art approximate MRC-generation algorithms, which we consider in different parts of this dissertation. CounterStacks introduced a breakthrough in MRC-generation in 2014 by using probabilistic counters (§2.4) to approximate the MRC while using logarithmic space [64]. CounterStacks is described in Section 2.2.3. The second algorithm came a few months after CounterStacks and introduced the first constant space MRC-generation algorithm called SHARDS [55], which we describe in Section 2.2.4. CounterStacks was considered to be more accurate than SHARDS, even by the authors of SHARDS, but our evaluation revealed that SHARDS is more accurate than CounterStacks. Nonetheless, CounterStacks has two primary benefits over SHARDS. The first is that it does not use sampling as SHARDS does, so each access affects the probabilistic counters used in CounterStacks.

The second benefit of CounterStacks over SHARDS is its support for interval-based analysis, where CounterStacks allows generating MRCs over arbitrary time intervals. As we will show later, interval-based analysis is important for modern cache workload analysis, as application caching requirements can change significantly over time. This made us focus on improving all aspects of the CounterStacks algorithm. However, since our discovery that SHARDS is more efficient than CounterSacks, our focus has shifted to improving SHARDS. We have been able to incorporate interval-based analysis with SHARDS, as we will show in Chapter 5. We also introduced the first exact one-pass MRC-generation algorithm to support interval-based historical analysis of cache workloads.

Most MRC-generation studies focused on a set of cache eviction policies called stack policies (primarily LRU), which adhere to the *inclusion property*: If an object exists in a cache of size s, then it must also exist in all caches of larger sizes [61]. The only exact method to generate MRCs for non-stack-based policies is to simulate each cache size on the MRC, which is an inefficient process (similar to what was used prior to Mattson's algorithm). In 2017, a breakthrough in MRC-generation for non-stack-based policies was introduced by Waldspurger et al., where they incorporated SHARDS with simulations to make them more tractable; they presented an algorithm called *Miniature Simulations* [88]. Background on Miniature Simulations and recent developments are discussed in Section 2.2.5.

Table 2.2: Key MRC-generation algorithms. SD: Stack Distance. R: Sampling rate. Sim.: simulations. S: Number of simulations. HOS: heterogeneous object size support. RT: reuse time.

Ref (year)	Exact	LRU	Space Overhead	Compute Overhead	HOS	TTL	Record	Comment
Mattson 1970 [61]	✓	✓	$\mathcal{O}$(M)	$\mathcal{O}$(NM)	✗	✗	SD	First one-pass algorithm
Bennet 1975 [82]	✓	✓	$\mathcal{O}$(M)	$\mathcal{O}(N \log N)$	✗	✗	SD	Used partial sum tree
Olken 1981 [62]	✓	✓	$\mathcal{O}$(M)	$\mathcal{O}(N \log \text{M})$	✓	✗	SD	Used AVL tree instead of a stack
Almasi 2002 [63]	✓	✓	$\mathcal{O}$(M)	$\mathcal{O}(N \log \text{N})$	✗	✗	SD	Proposed interval tree approach for stack distance calculation
PARDA 2012 [86]	✓	✓	$\mathcal{O}$(M)	$\mathcal{O}(N \log \text{M})$	✗	✗	SD	Parallel Olken implementation
CounterStacks 2014 [64]	✗	✓	$\mathcal{O}$(log M)	$\mathcal{O}(N \log \text{M})$	✗	✗	SD	Approximated stack distances using HLL counters
SHARDS 2015 [55]	✗	✓	$\mathcal{O}$(RM) $\mathcal{O}$(1)	$\mathcal{O}$(NRlog M) $\mathcal{O}$(N)	✗	✗	SD	First constant space algorithm
AET 2016 [113, 143]	✗	✓	$\mathcal{O}$(1)	$\mathcal{O}$(N)	✗	✗	RT	Approximated stack distance from reuse time
MiniSim 2017 [88]	✗	✗	$\mathcal{O}$(MRS)	$\mathcal{O}$(NMRS)	✗	✗	Sim.	Applied SHARDS to simulations
EAET 2018 [144]	✗	✓	$\mathcal{O}$(1)	$\mathcal{O}$(N)	✓	✗	RT	Extended AET to support heterogeneous sizes
RAR-CM 2020 [56]	✗	✓	$\mathcal{O}$(M)	$\mathcal{O}$(N)	✗	✗	SD	Uses re-access ratio (based on reuse time) to estimate the stack distance
DFShards 2021 [91]	✗	✗	$\mathcal{O}$(MRS)	$\mathcal{O}$(NMRS)	✗	✗	Sim.	Dynamically adapted MiniSim configurations
APAC 2022 [76]	✗	✓	$\mathcal{O}$(M)	$\mathcal{O}$(N)	✗	✗	SD	Approximated the stack distance using an approach similar to reuse time estimation

2.2.1 Mattson

In 1970, Mattson et al. introduced the first MRC-generation algorithm capable of generating an MRC in *one-pass* over a workload's trace of cache accesses, assuming a compatible eviction policy such as LRU [61]. Thompson [83] provided a definition for the meaning of a one-pass algorithm in the context of cache performance analysis relevant to MRC-generation:

> *"A cache management algorithm is called a one-pass algorithm if it can be simulated in time which is $\mathcal{O}(NS)$ and space which is $\mathcal{O}(S)$, where N is the number of trace events, and S is the largest cache size of interest."*

This definition is crucial to distinguish whether MRC-generation algorithms are one-pass or not. For instance, if cache simulations are used, and all simulation instances are performed in parallel, wherein each access in the access trace is read only once, one might consider it to be a one-pass process. However, since we are maintaining multiple cache instances rather than the largest cache instance, simulation is not a one-pass process.

Prior to Mattson's algorithm, generating an MRC required running a separate simulation for each different cache size on the MRC (i.e., a multi-pass process). This is the primary reason why Mattson et al.'s seminal work is among the most influential in the area of cache performance analysis and in particular MRC-generation.

The main observation behind Mattson's algorithm is that for a set of eviction policies like LRU, each object that exists in a cache size s must exist in all cache sizes $\geq s$. In addition, for all possible cache sizes $(0, \infty)$, the order of objects that exist in each cache is the same. These two conditions are referred to by Mattson et al. as the *inclusion property*. With this observation, Mattson et al. found that only a single reference per object is required rather than maintaining multiple instances per object, thus requiring $\mathcal{O}(M)$ space to generate the exact MRC, where M is the number of distinct objects in an access stream or trace. The main requirement to generate the MRC is maintaining a histogram of the caching requirements for each access, i.e., the smallest cache size required to allow the access to an object to result in a cache hit. Due to the inclusion property, accessing the same object in caches of all larger sizes must result in a cache hit as well. From this histogram, the MRC can be generated as described below.

For each access to an object in an access stream (or trace), Mattson's algorithm identifies the number of distinct objects accessed since the currently accessed 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 cache miss, and any larger cache size would result in a cache hit. To identify the number of distinct objects accessed since the currently accessed object was last accessed, Mattson's algorithm maintains a stack of distinct accessed objects ordered by access recency with the most recently accessed object at the top.[2]

For each access in the access stream, Mattson's algorithm linearly searches the stack from the top for the currently accessed object. If found, then the position in the stack, referred to as the ***stack distance***,[3] 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.

For an example, we use the accesses from Fig. 4 of Mattson et al. [148], $\{a, b, b, c, \boldsymbol{b}, a, d, c, a, a\}$. Fig. 2.1 shows Mattson's stack organization just before processing the 5th access (to b). To process the access to object b at time 5, we search for b in the stack to determine the stack distance. In this case, the object is in the second position in the stack, so the stack distance is 2, which is recorded in a histogram that records the frequency of each encountered stack distance.

[2]This can be easily implemented with a linked list, where each list node contains the key of the accessed object. In all MRC-generation algorithms discussed herein, there is no need to store the actual object (the values), but we only rely on the metadata such as the hash of the key, the object size, and the eviction time.

[3]Another closely related metric is the reuse distance. Although the stack distance and reuse distance are sometimes used interchangeably in the literature, the difference between them is that the stack distance measures the number of unique objects accessed between two accesses to the same objects, while the reuse distance measures the number of accesses between two accesses to the same objects [145–147], i.e., one is distinct count, and the other is not.

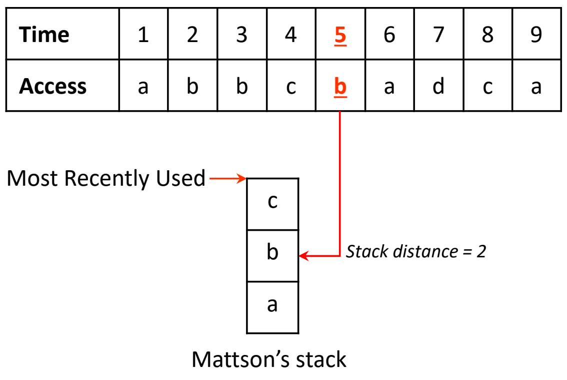


Figure 2.1: Example of the operation of Mattson's algorithm.

At any moment while processing the access stream, the MRC can be generated as the inverse Cumulative Distribution Function (CDF) of the histogram. The intuition behind this is that due to the inclusion property, any cache miss at cache size s must be a miss at cache sizes $\leq s$. Similarly, any cache hit at cache size s must be a hit at sizes $\geq s$. This means that the cache hits must be cumulative, which results in a monotonically decreasing MRC. We provide more details on the derivation of the MRC in Chapter 5. For a stream 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 stream. Mattson's approach is used by virtually all one-pass MRC-generation algorithms.

Side Note. In addition to LRU support, Mattson et al. presented an optimal eviction policy called OPT, and showed that it is a stack policy and proved it to be optimal [61]. Four years prior to that, Belady introduced a different optimal eviction policy called MIN, but the proof that it is optimal came roughly 6 years after Mattson et al. published the proof for the OPT algorithm, which Belady relied on to prove MIN is optimal [149]. Michaud dedicated an article to explain the properties of both MIN and OPT, which helps clear the confusion between the two as they are often used interchangeably in the literature even though they are different [149]. Optimal eviction plays an important role by serving as a baseline for comparing different eviction policies. Thus, OPT and MIN have been used as baselines to compare various eviction polices [88, 114, 150, 151]. One interesting observation is that although OPT and LRU are stack policies, most eviction policies that reduce the gap between LRU and OPT are not stack policies. Thus, it remains an open question if there is an eviction policy that outperforms LRU while remaining a practical stack policy given that OPT and MIN are not implementable in practice.

Reversing Conclusions About Prior Work

Although Mattson et al. stated that the Least Frequently Used (LFU) eviction policy is a stack policy (i.e., satisfies the inclusion property), Kelly et al. showed that there are two variants of LFU [152]; one is indeed a stack policy and another variant that is not. This distinction between the different variants of LFU was also noted by Breslau et al. [153]. Non-stack-based eviction policies can have non-monotonic MRCs, wherein increasing the cache size could increase the miss ratio rather than decreasing it, which is counterintuitive. Belady et al. [154] showed that the First-in-First-out (FIFO) policy exhibits this behavior, which is known as Belady's anomaly.[4]

[4]Page 2 from [154] shows a simple memory access sequence to produce the FIFO anomaly.

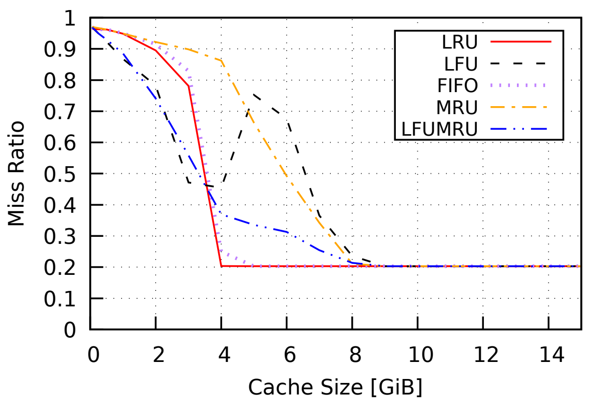


Figure 2.2: LFU's non-monotonic MRC for workload #067 from IBM. (interval [529539s-702339s])

Table 2.3: Example illustrating that MRU violates Mattson et al.'s inclusion property. At each time step, we show the content of Mattson's stack. We highlight in red when the violation occurs.

Cache Size / Access	a	b	c	d	b
1	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{b\}$
2	$\{a,-\}$	$\{b,a\}$	$\{c,a\}$	$\{d,a\}$	$\{b,c\}$
3	$\{a,-,-\}$	$\{b,a,-\}$	$\{c,b,a\}$	$\{d,b,a\}$	stack distance = 2
4	$\{a,-,-,-\}$	$\{b,a,-,-\}$	$\{c,b,a,-\}$	$\{d,c,b,a\}$	stack distance = 3

We discovered the first real-world example that shows LFU produces a non-monotonic MRC as shown in Fig. 2.2. The figure shows that LFU, for the particular workload shown, is the best policy up to a cache size of roughly 4GiB, but then suddenly it becomes significantly worse.

Similarly, the Most Recently Used (MRU) eviction policy was considered to be a stack policy [55, 91, 155], but we found it is not. An example is shown in Table 2.3, where a simple access sequence shows how the inclusion property is violated when using the MRU eviction policy. The access sequence is $\{a, b, c, d, b\}$. If we consider four cache sizes, from 1 to 4, then we notice after processing the 4th access that the stack distance for cache size 3 is 2, while it is 3 for cache size 4. In Chapter 4, we also show that incorporating heterogeneous size objects violates the inclusion property for eviction policies, including LRU, for any cache size smaller than the largest object size considered.

2.2.2 Olken

Mattson's algorithm is considered to be slow because it has to linearly search the stack on each access, thus having a computational overhead of $\mathcal{O}(M)$ per access. Several algorithms focused on improving Mattson's algorithm computational complexity. One of the earliest attempts was in 1975, when Bennet and Kruskal observed that instead of computing the stack distance as the position in a stack, it could be computed as the number of distinct objects accessed since the last access to the same object (plus one for the accessed object itself) [82]. This observation, while seemingly trivial, is important as it facilitated a breakthrough in approximate MRC-generation in 2014 with CounterStacks (discussed in Section 2.2.3). With this observation, Bennet and Kruskal introduced a more efficient LRU stack processing technique than Mattson by using a binary tree, which reduced the computational complexity from $\mathcal{O}(NM)$ to $\mathcal{O}(N \log N)$ $\big($i.e., $\mathcal{O}(\log N)$ per access to find the stack distance instead of Mattson's $\mathcal{O}(M)\big)$.

In 1981, Olken further optimized Bennet and Kruskal's approach by reducing the complexity of computing the stack distance per access from $\mathcal{O}(\log N)$ to $\mathcal{O}(\log M)$ by using a balanced binary search tree [62]. 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 $\mathcal{O}(N \log M)$. This is the most efficient *exact* one-pass MRC-generation algorithm to date.[5]

Olken described his algorithm by using an Adelson-Velsky and Landis (AVL) tree [62], but any balanced binary search tree would work as well (e.g., Niu et al. implemented Olken using a Splay tree [86]).[6] Each node in Olken's 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 while performing a tree search to compute the stack distance.

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 (plus one for the node itself), counted via the weights during the tree search. 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. Lastly, the MRC is generated from the stack distance histogram using Mattson's approach, described in Section 2.2.1.

As an example, consider the following access pattern $\{a, b, c, d, e, \boldsymbol{d}\}$. At each time step, an object is accessed, and a letter identifies the accessed object starting at time 1. After time 5, when object e is accessed, the state of the AVL tree is shown in Fig. 2.3. At time 6, object d is accessed again. Olken's hash table will be consulted to determine the last access time of object d, which is 4. The tree is then searched from the root for timestamp 4, and the stack distance is calculated based on the number of objects with larger timestamps. For object d, only object e has a larger timestamp, so the stack distance is equal to the weight of the right sub-tree of the node containing object d (plus 1 for the node itself), which is 2.

Side Note. In 2012, Niu et al. presented a parallel implementation of Olken's algorithm they called PARDA [86, 157]. In June 2023, a new MRC-generation algorithm called Increment-and-Freeze was presented [158]. The main motivation behind Increment-and-Freeze is the observation that tree-based algorithms (e.g., Olken and PARDA) have poor locality, which leads to poor performance. Their new algorithm aimed to improve locality, and they showed improved results over PARDA. However, although we consider analyzing this new algorithm for future work (in particular to compare with single-threaded PARDA, i.e., Olken), one of its main potential downsides is that its space complexity is $\mathcal{O}(N)$ while Olken's is $\mathcal{O}(M)$.

[5] It remains an open research question what the lowest computational boundary for *exact* MRC generation is. The space complexity boundary is clearly $\mathcal{O}(M)$, because we must keep a copy of each distinct object accessed in order to compute the exact stack distance. However, it is unclear whether $\mathcal{O}(\log M)$ is the lowest achievable compute overhead for computing the stack distance on each access, or whether this could be further improved.

[6] Splay trees were invented in 1985 [156], i.e., 4 years after Olken's thesis. It is unclear which one is the most efficient balanced binary search tree for Olken's algorithm.

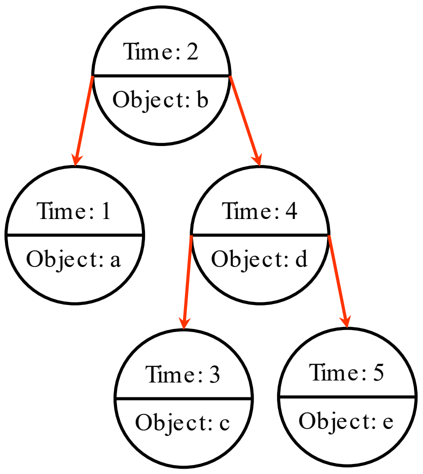


Figure 2.3: State of Olken's AVL tree after processing the fifth access in the pattern $\{a, b, c, d, e, d\}$.

2.2.3 CounterStacks

Bennet and Kruskal observed that the stack distance for an access can be obtained as the number of distinct objects accessed since the last access to the same object instead of the position of the object in Mattson's stack [82]. CounterStacks [64, 119, 120, 159–161] approximates this number of distinct objects accessed using cardinality estimation (also known as probabilistic counting and F_0 estimators [162–172]), and in particular, the HLL counter [166] (which we describe in Section 2.4). CounterStacks is explained in greater detail here because we extend this algorithm in Chapters 3 and 4, so a deeper understanding of CounterStacks is necessary to understand our extensions.

We first describe a basic CounterStacks algorithm that is computationally highly inefficient for ease of understanding. We then describe a number of optimizations the CounterStacks team added to the basic algorithm that make it more efficient. These optimizations result in CounterStacks having a compute complexity of $\mathcal{O}(N \log M/\delta)$ and a space complexity of $\mathcal{O}(\log M/\delta)$, where δ is a fixed pruning parameter described below [64, 119]. The basic CounterStacks algorithm works as follows. For each access to an object in the access stream or trace:

1. a new counter is instantiated and added to a stack of counters, and
2. all previously created counters are incremented by one if and only 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. Thus, the value of c_i corresponds to the smallest cache size (in number of objects) needed to achieve a hit for the accessed object.

Fig. 2.4 shows an example of the operation of CounterStacks. When the first access (to a) is processed, CounterStacks initializes a new counter c_1. The value of c_1 will equal the number of unique objects encountered since it was initialized, which is 1. A new counter, c_2, is added when the second access (to b) is processed. The value of c_1 and c_2 will be updated such that $c_1 = |\{a, b\}| = 2$ and

					$\longrightarrow$ time (j) $\longrightarrow$					
Time	1	2	3	4	5	6	7	8	9	10
c_i	a	b	b	c	**b**	a	d	c	a	a
c_1	1	2	2	3	3	3	4	4	4	4
c_2		1	1	2	2	3	4	4	4	4
c_3			1	2	$\boxed{2}$	3	4	4	4	4
c_4				1	$\boxed{2}$	3	4	4	4	4
c_5					1	2	3	4	4	4
c_6						1	2	3	3	3
c_7							1	2	3	3
c_8								1	2	2
c_9									1	1
c_{10}										1

Figure 2.4: Example of the operation of CounterStacks using the accesses from Fig. 4 in Mattson et al.'s paper [61]. Each row represents a counter, going from oldest at the top to the newest at the bottom. Each column represents a time step, from left to right, with the data of the current access identified at the top. The counter values shown are those obtained after processing the access at the top of the respective column.

$c_2{=}|\{b\}|{=}1$, and so on.[7] As an example for determining the stack distance, consider the processing of the third access to b at time step 5 (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.

Optimizations. The basic CounterStacks algorithm, as described above, is highly inefficient, as each counter needs to record all distinct references it has encountered, and these then need to be searched through on each access. This makes the basic algorithm's time complexity $\mathcal{O}(N^2M)$ and space complexity $\mathcal{O}(NM)$. We now describe a number of optimizations to the basic algorithm which were introduced by the authors of CounterStacks [64].

HLLs. Using counters requires CounterStacks to maintain a list of previous references for each counter. To alleviate this burden, CounterStacks ingeniously uses probabilistic counters to reduce space usage. One of the most practical probabilistic counters selected for this task is the HLL counter, which has good accuracy while using a fixed amount of space (e.g., for 98.5% accuracy it uses approximately 4KiB of space) [159, 166]. HLLs are used in CounterStacks to approximate the number of unique references for each counter while eliminating the need to maintain a list of previous references. More details on HLLs are presented in Section 2.4.

To use HLLs, the accurate counters are simply replaced with HLLs and their reported values are used in the same way. Using HLLs in CounterStacks could introduce negative frequencies in the stack distance histogram due to the probabilistic nature of the counters. To overcome this issue, the process of generating the MRC from the stack distance histogram (i.e., Mattson's approach) is adjusted to ensure that the cumulative frequency is monotonically increasing which should prevent the miss ratio from increasing for larger cache sizes.

[7] In this context, $|\mathcal{X}|$ refers to the cardinality of the multiset $\mathcal{X}$, i.e., the number of distinct objects.

Pruning. Once two counters have the same number of unique references, their future values will remain the same in perpetuity because each new access reference adds one to both counters. Hence, pruning is used to keep only one of these counters in order to save space and computation. CounterStacks is more aggressive and deletes a younger counter when its value is at least $(1 - \delta)$ times the older counter. This guarantees that the number of counters is at most $\mathcal{O}(\log M/\delta)$, where δ is a fixed pruning parameter [64]. There are two variants of CounterStacks: *High-Fidelity* (HiFi), which uses a pruning δ of 0.02, and *Low-Fidelity* (LoFi), which uses a δ of 0.1.

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), while taking into account all d accesses since the last update. Wires et al. showed that downsampling has minimal impact on the accuracy of the MRCs for the MSR workloads [64]. Their experimental evaluation used a downsampling factor $d = 1$ million, but 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.

CounterStacks uses Mattson's stack distance histogram approach to generate the MRC, wherein a histogram of stack distances is built and then used to generate the MRC (§2.2.1). Downsampling changes the way the stack distance is computed because with downsampling, up to d accesses are processed at a time instead of a single access. To estimate the stack distances for those accesses, CounterStacks iterates over all the counters, from oldest to newest, and compares how much adjacent counters increased after processing these accesses. Between time j and $j+1$, if two adjacent counters, c_i and c_{i+1} are increased by Δ_i and Δ_{i+1}, respectively, we can infer that $(\Delta_{i+1} - \Delta_i)$ accesses represent hits in the cache represented by c_i but misses in the cache represented by c_{i+1} [64]. Thus, CounterStacks increments the c_i histogram bin by $(\Delta_{i+1} - \Delta_i)$. For the last counter in the stack, c_n, the histogram bin corresponding to c_n is incremented by $d - c_n$. The reason is that c_n represents the number of distinct objects accessed in the last processed d accesses, and thus $d - c_n$ hits must have occurred. 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. Wires et al. refer to the aforementioned process as the *finite differencing scheme* [64]. Eq. 2.1 details the histogram increments for each counter while performing the finite differencing scheme.

$$\text{Number of Hits} = \begin{cases} d - c_{i,j} & \text{, last counter} \\ (c_{i+1,j} - c_{i+1,j-1}) - (c_{i,j} - c_{i,j-1}) & \text{, otherwise} \end{cases} \tag{2.1}$$

Downsampling is the most complex aspect of the CounterStacks algorithm, and it is counterintuitive, thus we provide a numerical example for better understanding. Consider the scenario shown in Fig. 2.5, where three batches are processed, each containing three accesses (i.e., $d = 3$). After the first batch, b_1, is processed, the finite-differencing scheme is applied. Note that by the end of the first time step, there will only be a single counter, i.e., c_1. Thus, the finite differencing scheme will result in incrementing the ∞ histogram bin by 3 as there are no hits: i.e., when applying Eq. 2.1, then $d - c_{1,1}$ will be $3 - 3$, which equals 0 hits.

	$\longrightarrow$ Batch (b) at time (j) $\longrightarrow$		
Time	1	2	3
c_i	$b_1\{a,b,c\}$	$b_2\{a,b,d\}$	$b_3\{c,e,e\}$
c_1	3	4	5
c_2	-	3	5
c_3	-	-	2
histogram update analysis after processing b_j using Eq. 2.1			
	• Increment the ∞ bin by 3 misses.	• $+2$ hits $\big((3-0)-(4-3)\big)$ at sd 4 (i.e., $c_{i,j}$). • $+0$ hit $(d-3)$ at sd 3. • $+1$ miss at sd ∞.	• $+1$ hit $((5-3)-(5-4))$ at sd $\{5\}$. • $+0$ hit $((2-0)-(5-3))$ at sd $\{5\}$. • $+1$ hit (d-2) at sd 2. • $+1$ miss at sd ∞.

Figure 2.5: Example of CounterStacks with downsampling ($d = 3$). The figure illustrates the stack distance histogram update process for three batches of accesses to cache objects, each containing three accesses. For example, in batch b_1, accesses to objects $\{a, b, c\}$ result in counter c_1 being initialized to 3. Applying Eq. 2.1 shows that there are no cache hits for the accesses from this batch, and all three accesses result in misses, thus incrementing the ∞ bin by 3. (*sd: stack distance*).

After processing the second batch, b_2, a new counter c_2 is instantiated. The accesses from b_2 are added to c_1 and c_2. Applying the finite differencing scheme to c_1 and c_2 results in the following histogram updates. First, for c_1, there are two cache hits at stack distance 4 (i.e., $(3-0)-(4-3) = 2$), thus the stack distance histogram bin corresponding to the stack distance 4 is incremented by 2. Second, for c_2, there are no cache hits (i.e., $d - 3 = 0$). Lastly, the ∞ histogram bin is incremented by d minus the sum of all previous cache hits, which is $3 - 2 = 1$.

For the third batch, b_3, a new counter c_3 is instantiated and the accesses from this batch are added to c_1, c_2, and c_3. Applying the finite differencing scheme results in the following histogram updates. First, for c_1, there is one cache hit at stack distance 5 (i.e., $(5-3)-(5-4)$). Second, for c_2, there are no extra cache hits at stack distance 5 (i.e., $(2-0)-(5-3)$). Third, for c_3, there is 1 cache hit at stack distance 2 (i.e., $d - c_{3,3} = 3 - 2$). This means that for this batch that contained 3 accesses, two of them are hits, and there is a remaining single miss, which is for the first access to object e that was never seen before (i.e., a compulsory miss). Lastly, the ∞ histogram bin is incremented by 1 to reflect that compulsory miss.

Streaming. A major feature of CounterStacks is its ability to stream intermediate counter information; e.g., by persistently storing the most recent counter values after the processing of each batch. For example, each column shown in Fig. 2.5 can be streamed after processing the corresponding 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 (assuming accesses are for disjoint objects [64]). The MRC for the accesses between any time points t_x and t_y can be generated by computing the stack distances using the finite differencing scheme described earlier 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. Mattson's histogram approach is then applied to generate the MRC form the stack distance histogram (as we described in Section 2.2.1).

2.2.4 SHARDS

To reduce the overhead of exact MRC-generation algorithms, such as Olken, Waldspurger et al. introduced Spatially Hashed Approximate Reuse Distance Sampling (SHARDS) [55, 173]. SHARDS runs on top of an exact algorithm such as Olken to generate the MRC but considers only a sampled subset of the total accesses in the workload. Access i in the workload is sampled if the hash of the accessed object's key, H_i, modulo P (a constant typically set to 2^{24}) is less than a threshold T; i.e., if $H_i \% P < T$. Thus, 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.

Sampling makes SHARDS an approximate algorithm but generates surprisingly accurate MRCs for most workloads the original authors tested, even with a low sampling rate of R=0.001 (= 0.1%) [55]. Two variants of SHARDS were 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**$_{adj}$. We now describe each of these variants.

FR-SHARDS

FR-SHARDS uses a fixed sampling rate, R, when processing the workload to generate the MRC. It requires less time and space to generate an MRC than the underlying MRC-generation algorithm it uses, such as Olken, where the time complexity becomes $\mathcal{O}(RN \log M)$ and space complexity becomes $\mathcal{O}(RM)$. For example, with a sampling rate of 0.001, memory and compute overheads are decreased by approximately a factor of 1,000. The primary challenge with FR-SHARDS is determining a suitable sampling rate, R, for a given workload to achieve the desired accuracy *a priori*.

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 were generated for most workloads they have tested [55]. This variant achieves $\mathcal{O}(1)$ space overhead and $\mathcal{O}(N)$ compute overhead, making it one of the most efficient approximate MRC-generation algorithms. FS-SHARDS begins with a high sampling rate (typically $R = 0.1$ or even $R = 1.0$) 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 $\langle$`object,` $F_i\rangle$ tuples, ordered by F_i. Once the number of sampled objects is about to exceed S_{max}, the object with the largest sampling factor, F_{max}, is removed from F-PQ and Olken's data structures, then the algorithm's sampling threshold, T, is lowered to F_{max}.

SHARDS$_{adj}$

Sampling bias is a known downside of SHARDS, as it may not sample frequently accessed objects, leading to highly inaccurate MRCs [55]. This poor accuracy stems from SHARDS not being able to distinguish between highly popular and less popular objects. To mitigate this issue, Waldspurger et al. proposed an extension to SHARDS called SHARDS$_{adj}$ [55]. This modification estimates the number of accesses expected to be sampled for a given sampling rate and then adjusts the first bin in the stack distance histogram by the difference between the expected and actual number of sampled accesses. This extension is based on the premise 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 should result in hits at relatively small stack distances. Increasing the frequency of the first bin in the histogram addresses this issue.

Extensions to Shards

Carra et al. proposed an extension to Shards to reduce its error for small cache sizes [90]. The primary idea behind their work is to build two MRCs for an access trace instead of one. The first MRC is exact and the other one is sampled using Shards. For the exact MRC, the stack size (recall from Mattson's algorithm) is limited to include up to B objects, which effectively builds the MRC for a maximum of B objects. Then, the two MRCs are combined into a single one by simply merging the two for their corresponding portions. Their primary claim is that high errors occur with Shards when generating MRCs only in the portions of the MRC corresponding to small cache sizes. They tested synthetic Zipfian workloads and a few real-world workloads (with 30 million accesses) and showed that their extension can significantly reduce the errors.

We found that Carra et al.'s extension presents challenges in practical application for two reasons. Firstly, building an exact MRC requires processing all the accesses in the workload, which can significantly reduce the benefits of Shards sampling. For instance, in our experiments using the access traces from MSR, we observed that the original Olken's algorithm can process 2.7 million accesses per second on average. Using FR-Shards with a sampling rate of $R = 1\%$ increases the throughput to 89.2 million accesses per second on average. However, using Carra et al.'s approach with the same FR-Shards sampling rate of 1% ($B = 8{,}000$) reduces the throughput to 4.8 million accesses per second on average.

Secondly, the benefit of the exact MRC portion is primarily for small cache sizes, which are typically not used in practice (e.g., $<$ 64MiB). Another complication of this work is that it assumed uniform object sizes, which makes it simple to build an exact MRC for up to B objects. However, once extended for heterogeneous object sizes, limiting the number of objects to B will no longer result in an exact MRC, and in fact, it would cause significant errors. The reason is that the aggregate size of the B objects will be different at different time points while processing the access trace.

2.2.5 MRC-Generation for Non-Stack-Based Policies

A myriad of cache eviction policies have been developed to try to approach the clairvoyant optimal miss ratios (e.g., OPT/MIN) for different workloads [130, 136, 145, 146, 150, 151, 174–203]. Examples include LRU, Least Recently Frequently Used (LRFU) [183], Frequency-Based Replacement (FBR) [182], Early Eviction LRU (EELRU) [175], The Low Interreference Recency Set (LIRS)[194], 2Q [181], Multi-Queue algorithm [195], Adaptive Replacement Cache (ARC) [150], LHD [196], Tiny-LFU [197], S3-FIFO [184], and Sieve [136]. Domain-specific eviction policies, such as policies for SSD caches, have also been developed [130, 174, 198–203].

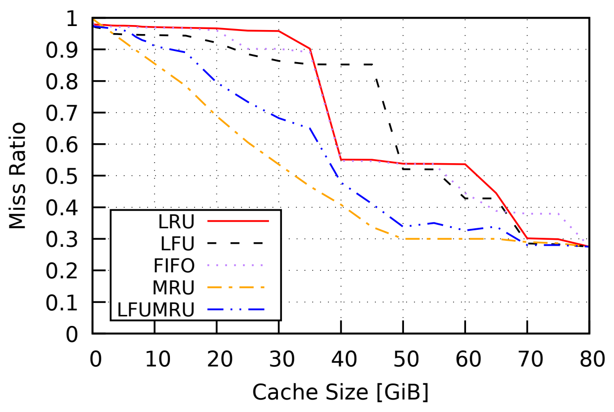


Figure 2.6: MRCs for different eviction policies for the MSR `web` workload, which includes a cyclical access pattern.

Since optimal eviction policies, such as OPT, have been shown to require knowing the future and are thus not practical [149], a large number of developed eviction policies are striving to reach the miss ratios of optimal eviction. However, as evidenced by the large number of available policies, it is clear that there is no single “best” eviction policy; typically, the performance of eviction policies depends on the workload’s access pattern. For instance, it has been shown that MRU works best for cyclical workloads [204–206]: see Fig. 2.6 for an example. Similarly, LFU has been shown to be close to Belady’s MIN for some workloads (primarily workloads with Zipfian power law distributions and uniform object sizes) [152, 153, 207]. FIFO has been shown to provide performance close to that of LRU for modern in-memory cache workloads from IBM and Twitter [22, 125]. In those cases, FIFO is a better-performing eviction policy because it has lower operational overheads than LRU.

Despite the large number of proposed eviction policies, the preponderance of MRC-generation studies has focused on a set of policies called stack policies, primarily LRU [55, 61, 62, 64, 86, 113]. Some potential reasons why this might have been the case:

1. LRU is a *de facto* standard and the default eviction policy for most caches [22–24] (e.g., Redis [208] and Memcached [31] are LRU-based).
2. LRU is very simple to implement in software, while other policies require more sophisticated designs (e.g., ARC [150] and 2Q [181]).
3. The only method to obtain *exact* MRCs for non-stack-based policies is to run a separate simulation for each cache size on the MRC, a process that is space- and time-inefficient.

Miniature Simulations

In 2017, Waldspurger et al. introduced a breakthrough in MRC-generation for non-stack-based policies with *Miniature Simulations*. Waldspurger et al. observed that SHARDS sampling (described earlier in Section 2.2.4) can be applied to simulations in order to make them more tractable [88]. Thus, their proposed algorithm, Miniature Simulation, processes a subset of the accesses in a workload’s access trace to generate the MRC with good accuracy (with a typical mean absolute error of less than 2% [88]), yet it can model any eviction policy.

Miniature Simulations works as follows. The number of simulated cache instances needs to be specified before the MRC-generation process starts. This number determines the granularity of the generated MRC. In their evaluation, Waldspurger et al. used 100 equally distributed cache sizes to be simulated with a predefined maximum cache size. Hence, 100 cache simulator instances are created using the desired eviction policy. Each access in the workload is then processed by each of the cache simulation instances. For each instance, a counter is maintained to record the number of misses, and this number is updated whenever an access to a simulated cache results in a miss. To generate the MRC, at any point in time, the miss ratio for each cache simulation instance is recorded on the MRC with the corresponding simulated cache size.

Instead of processing all the accesses in a workload's access trace, Miniature Simulations utilizes SHARDS sampling, making it an approximate algorithm. Given a fixed sampling rate R, accesses to only $R\%$ of the objects are sampled. The simulated cache sizes are scaled down by $1/R$, thus significantly reducing the space requirements for each simulation instance.

Recent developments. In 2021, Yu et al. identified challenges with Miniature Simulations, such as the need to determine the number and size of simulated caches before beginning the process of generating the MRC [91]. Simulating too many sizes could negatively impact the performance of Miniature Simulations, while the selected sizes affect the granularity and utility of the MRCs. To address these challenges, they proposed DFSHARDS to dynamically adjust the configurations of Miniature Simulations in real time based on the behavior and access patterns of the workload being simulated. Their evaluation showed that DFSHARDS saves up to 47% of the memory space required for MRC-generation compared to Miniature Simulations. However, adjusting the number of simulated caches has significant effect on the throughput of DFSHARDS, thus making it impractical to generate MRCs online [124].

Lastly, in 2024, our research group presented another improvement for non-stack policy MRC generation [124]. This work is based on the premise that although non-stack policies violate the inclusion property, this violation does not significantly affect the generated MRC for the policies that were tested. The developed algorithm, Kosmo, simulates different cache sizes and uses SHARDS sampling similar to Miniature Simulations and DFSHARDS but generates the MRC differently. Mattson's approach, where the MRC is generated from a stack distance histogram, is used, but with the adjustment for the meaning of a stack distance. For non-stack policies, they define the stack distance of an access to an object as the smallest simulated cache size where the object exists. Kosmo has been shown to be more efficient than Miniature Simulation [124].

2.3 Working Set Size (WSS)

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 [11, 14, 60, 68, 93–106, 209–211]. WSS has practical applications in memory-aware load balancing, improving database performance, virtual machines failure recovery, and it has been a guiding metric for memory allocations [68, 102–104, 106, 212]. Its origins can be traced back to Denning, who introduced the working set model for understanding program behavior, emphasizing its significance in the dynamic management of paged memories and resource allocation [11]. A comprehensive background on the origins and applications of the Working Set is presented in [14, 60].

In the context of in-memory cache management, the WSS refers to the aggregate size of distinct objects accessed in a workload over a specified interval of time. The WSS identifies the minimum cache size needed to achieve the minimal miss ratio, which is the *compulsory miss ratio*. The compulsory miss ratio is due to the fact that accesses to an object must result in at least one miss, which is for the first access to that object; subsequent object cache misses are capacity (or expiry) misses, which happen because the object does not exist in a smaller cache size due to evictions. Given a cache size equal to the WSS, there will be no capacity misses.

There are mixed opinions as to the utility of the WSS. Some consider it *"not enough to guide memory allocation"* and find MRCs to offer more utility [55, 213]. The primary reason behind such claims is that the WSS does not offer insights into the performance of the cache, i.e., the miss ratio at a cache size equal to the WSS [213]. We have observed that it is relatively straightforward to find the performance for a cache size equal to the WSS by dividing the number of distinct objects accessed over the total number of accesses, thus obtaining the compulsory miss ratio. Other researchers consider the WSS to be sufficient to guide memory allocations without the need for MRCs [106].

We argue that the WSS and MRCs are both valuable and have different use cases. For small WSS values, directly setting the cache size to the WSS will minimize the miss ratio. Conversely, for larger WSS values, the MRC provides insights into cache size versus miss ratio trade-offs, which are not available when using the WSS. In Section 3.5, we provide some surprising insights, which show that many real-world workloads have very small WSSes, thus underscoring the utility of the WSS.

Effect of Object Expiry on the WSS.

Section 2.1.1 highlighted the importance of TTL expiry in modern caches. TTL attributes directly affect the WSS because within the interval the WSS is measured, some of the objects might expire. This complicates the meaning and measurement of the WSS.

We differentiate between the original WSS and the WSS when objects expire based on TTL by introducing the term **"unexpired WSS"**. The original WSS measures the aggregate size of distinct objects accessed in a workload over a specified interval, while the unexpired WSS measures the aggregate size of unexpired distinct objects accessed within the interval.

We illustrate the difference between the two WSS variants using the example shown in Fig. 2.7. The dotted line in the figure shows the original WSS measured as the aggregate size of distinct objects accessed since the beginning of the workload's access trace up to the point the WSS is measured. The WSS keeps increasing monotonically, ultimately reaching 210GiB.

Taking TTLs into consideration leads to objects expiring. In the figure, the solid line shows the unexpired WSS, measured as the aggregate size of unexpired distinct objects accessed since the beginning of the workload's access trace. The unexpired WSS oscillates between 2.31GiB and 22.75GiB. When using the unexpired WSS for cache allocation, utilizing the high watermark value allows for accommodating the worst-case scenario within the interval in which the unexpired WSS is measured. Hereinafter, we use unexpired WSS and WSS_{ttl} interchangeably.

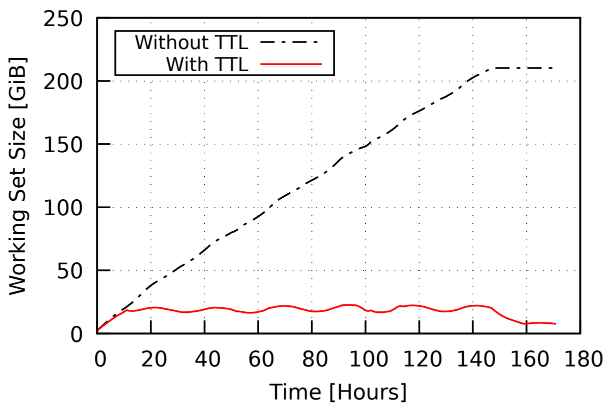


Figure 2.7: Working Set Size for the Twitter workload #46, both with and without TTLs. Each point at time t represents the WSS from $[0, t)$. The dotted line shows the original WSS, while the solid line shows the unexpired WSS when taking TTLs into consideration.

2.4 Cardinality Estimation

Cardinality estimation (also known as probabilistic counting) refers to approximating the number of distinct objects accessed in an access stream or trace. Consider the following example, which motivates the need for cardinality estimation. A web service administrator is interested in finding the number of unique sessions connected to their web service. Assuming each session has a unique identifier, this task can be easily performed in linear space. However, if the number of sessions is large, then significant space would be needed for this purpose. For instance, assume that we have one billion sessions, and each session is identified with a 64-bit hash, then approximately 8GiB of memory would be required. Cardinality estimation reduces this space requirement significantly while being highly accurate. For instance, one of the most widely used cardinality estimators, the HLL counter, achieves 98.4% and 99.6% accuracy while using ˜4KiB and ˜64KiB of space, respectively [166, 167].

We are interested in using a cardinality estimator (also known as probabilistic counter and F_0 estimators [162–172]) to determine the number of distinct objects accessed in a stream of cache accesses. The HLL counter can efficiently approximate the number of distinct elements in a multiset [166, 167]. For our particular application, the multiset being considered contains one element for each access in the target workload.[8] More specifically, each element is a hash of the key used to access an object in the cache. Below, we give a high-level overview of the HLL algorithm, which is needed to understand our extended HLL to accommodate TTLs in Chapter 3 and our extended HLL to accommodate heterogeneous object sizes in Chapter 4.

[8]A multiset (a.k.a. bag [214]) is a generalization of a set: while a set contains only one occurrence of any given object, a multiset may contain multiple occurrences of the same object. We use standard set notation throughout the dissertation for multisets. The multiset could also refer to a stream of accesses to cached objects; each object is identified by a key, typically an integer hash value.

```
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 $\alpha$ 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 2.8: HLL operations

Assuming multiset $\mathcal{M}$ contains 64-bit integers, the HLL 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 bias correction factor [166]:

$$\alpha = \begin{cases} 0.673 & \text{if } b = 4 \\ 0.697 & \text{if } b = 5 \\ 0.709 & \text{if } b = 6 \\ \frac{0.7213}{1+1.079/2^b} & \text{if } b \geq 7 \end{cases} \tag{2.2}$$

To improve accuracy, the algorithm first partitions the elements of $\mathcal{M}$ into 2^b buckets. The b-bit prefix of element 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`$[0 : 2^b - 1]$. The overall count estimate is then $\alpha \cdot 2^{\overline{n+1}}$, where $\overline{n+1}$ is the harmonic mean[9] of the bucket maxima in the bucket array.

The estimation error of an HLL counter is $1.04/\sqrt{2^b}$, so b is effectively a precision parameter [166]; 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 $b = 12$, 3KiB of space is needed for the bucket array.[10]

There are three main operations on HLL counters: `Insert`, `Count`, and `Merge`. Their implementations are surprisingly simple and shown in Fig. 2.8. `Insert` updates an HLL to reflect a new element x being added to the 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.

[9]Flajolet et al.'s HyperLogLog cardinality estimator [166] improved upon its predecessor, the LogLog algorithm [165], by using a harmonic mean instead of a geometric mean. This reduced the effect of outliers and improved estimation accuracy from $1.30/sqrt(2^b)$ to $1.04/sqrt(2^b)$.

[10]In practice, using a byte instead of 6 bits per bucket simplifies the implementation, but increases the space required from 3KiB to 4KiB. In our implementation, we use Heule et al.'s HLL^{++} implementation [167], which supports 64-bit hashes and has a sparse implementation. Throughout the dissertation, we refer to Heule's HLL^{++} as HLL.

Chapter 3

Accommodating TTLs in MRC Generation and WSS Estimation

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) attributes to define the lifespan of cached objects, a feature essential for ensuring data freshness and adhering to regulations like the General Data Protection Regulation (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 attributes into consideration allows a 69% lower memory footprint for in-memory caches on average (and up to 99%) without a degradation in the miss ratio.

This chapter 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 performance levels comparable to those of the original algorithms. All our extended approximate algorithms are efficient, run in constant space, and enable more resource-efficient and cost-effective cache management.

3.1 Introduction

In this chapter, 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 trade-offs [55, 56, 61, 62, 64, 75, 86–90, 105, 113, 143, 144, 215], it is surprising that none of these tools are able to support modern workloads. More specifically, none of these tools are able to take TTL attributes of cached objects into account, even though many modern workloads use TTLs to limit the lifespan of cached objects [39, 109, 125, 134, 135]. This chapter rectifies this situation for the most important state-of-the-art MRC generation and WSS estimation algorithms.

MRC. The MRC is the most effective tool for evaluating cache size trade-offs. It plots the cache miss ratio as a function of the cache size for a given workload under a specific eviction policy. Recall our example from Fig. 1.2 which showed the MRC for cache workload #079 from IBM, and provided details on how cache sizing decision could be guided by the MRC.

Mattson [61] and Olken [62] are 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 [55, 64, 113]. Consequently, a number of algorithms have been introduced that generate *approximate* MRCs with significantly lower computation and memory overheads. Examples include Counterstacks [64], Shards [55], and AET [113]. Despite being approximate, these algorithms generally produce MRCs with acceptable errors, e.g., $<2\%$.

WSS. The 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 ratio. With TTLs, the WSS becomes the aggregate size of the unexpired distinct objects accessed, which we refer to as unexpired WSS (or WSS_{ttl}), as discussed in Section 2.3.

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 unique 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 [104, 105, 215].

Finally, a WSS estimate can be obtained by using a cardinality estimator (§2.4) to identify the number of distinct objects accessed. A popular cardinality estimator is the HLL counter [166, 167], which is also used by the CounterStacks MRC-generation algorithm. The 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.

TTLs Matter. For many modern workloads, TTLs play a critical role in cache management by providing a mechanism to expire cached objects based on their age [39, 109, 114, 125–135]. TTLs are used, for example, to limit stale, inconsistent data in the cache or to implement GDPR-mandated restrictions [39, 109, 125, 133, 138]. In the Twitter cache workloads that have been made public [39], each cached object has an associated TTL attribute. Fig. 3.1 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 attributes can significantly impact MRCs and WSSes. For example, Fig. 3.2 shows that neglecting TTLs when generating MRCs can result in substantially inaccurate MRCs. The figure shows two MRCs for Twitter's recommended workload #50: one taking TTLs into account and the other not. The cache size needed to achieve the minimal miss ratio is 7.3GiB when TTLs are taken into account, but it increases to 123GiB when they are not.

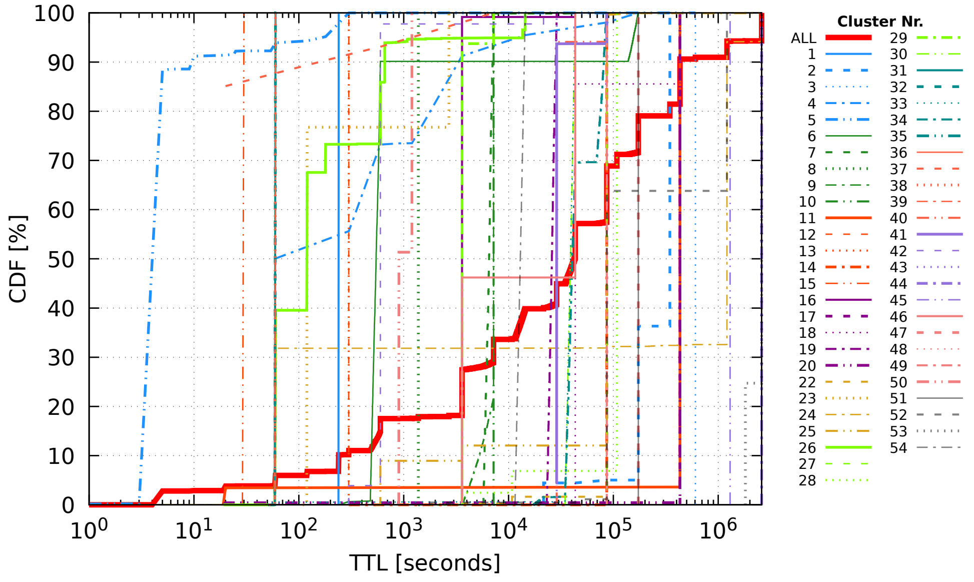


Figure 3.1: Cumulative distribution function (CDF) of the TTL attributes for the workloads in the Twitter collection. The red curve shows the CDF of TTL attributes across all accesses.

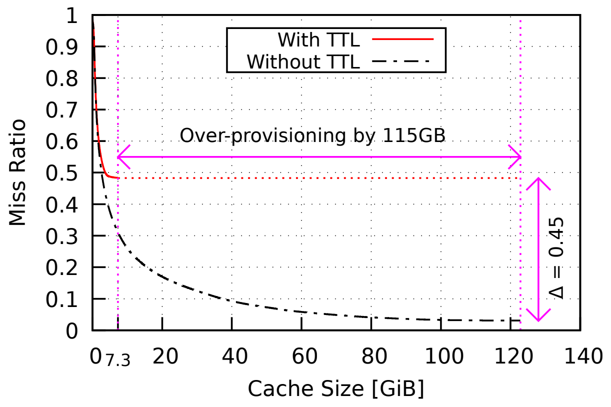


Figure 3.2: MRCs for the Twitter workload #50: With TTL, it reaches steady-state at 7.3GiB; without TTL, at 123GiB.

Similarly, neglecting TTLs when generating WSSes can also be highly misleading. Fig. 3.3 shows that for Twitter's recommended workload #19, the unexpired WSS never exceeds 5GiB when taking TTLs into account but reaches 40GiB when neglecting TTLs. Given that some objects may expire, the WSS can fluctuate, as illustrated. Therefore, cache sizing should be based on the high watermark of the unexpired WSS within an interval, rather than the instantaneous unexpired WSS.

Benefits of streaming. Streaming refers to the periodic saving of HLL counters [64]. 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 granular level, and it enables analysis of the effects of combining multiple workloads to use a single cache.

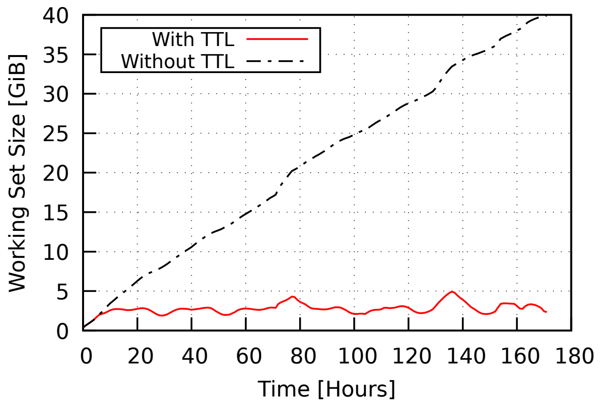


Figure 3.3: WSS for the Twitter workload #`19`, both with and without TTL. Each point at time t represents WSS from $[0, t)$. The solid line shows the unexpired WSS.

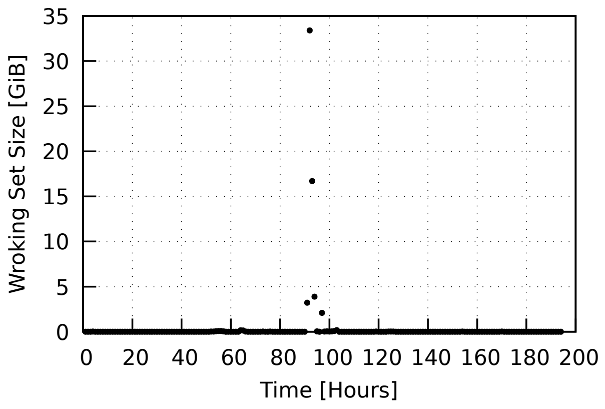


Figure 3.4: WSSes for the MSR `src2` workload. Each point captures the WSS over a one hour time period. For most hours, the size of the WSS is under 200MiB. In hours 91–97 the WSS of the workload is orders of magnitude larger.

As an example, consider Fig. 3.4 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.

If these one-hour WSSes are saved as HLL counters, then the HLL `Merge` operator (§2.4) 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. 3.5 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 interval $[0, t)$. 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 [64]. For example, Fig. 3.6 shows the MRCs for three different publicly available workloads from MSR [115], IBM [22], and

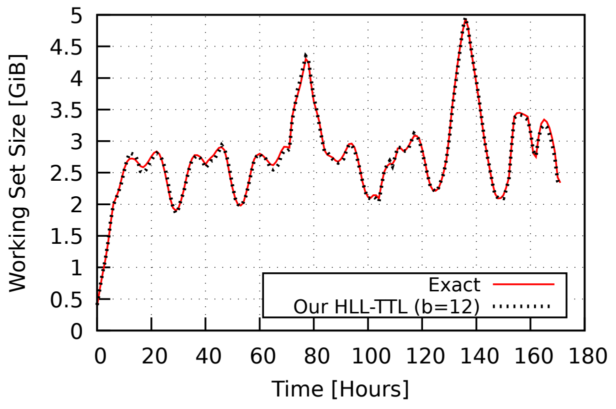


Figure 3.5: Twitter workload #19 unexpired WSS both with TTL. The solid line was obtained from exact WSS calculations using a hash table; each point at time t identifies the unexpired WSS over the interval $[0, t)$. The dotted curve was obtained using our extended HLL; each point at time t is the result of merging the individual one-hour WSS estimates for hours 0 to t.

Twitter [39]. 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 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 rest of the chapter, 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 (§3.2). We refer to these extended algorithms as Mattson^{++} and Olken^{++}. The necessary 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. We refer to the extended SHARDS algorithm as SHARDS^{++}.

Second, we show how HLL counters can be extended to accommodate TTLs (§3.3). 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. We refer to the extended HLL algorithm as HLL-TTL.

Third, we show how our extended HLLs can be used to extend the CounterStacks MRC-generation algorithm to accommodate TTLs (§3.4). We found that a straightforward integration of our extended HLL counters with CounterStacks led to a significant negative impact on accuracy. This is the case because CounterStacks processes accesses in batches [64], which may result in inaccuracies when handling TTLs, as some accesses could expire within the batch before the timestamp of the last access in the batch. 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 critical optimizations, we ultimately developed an algorithm that is far more efficient than CounterStacks and requires only a constant amount of space. We refer to the

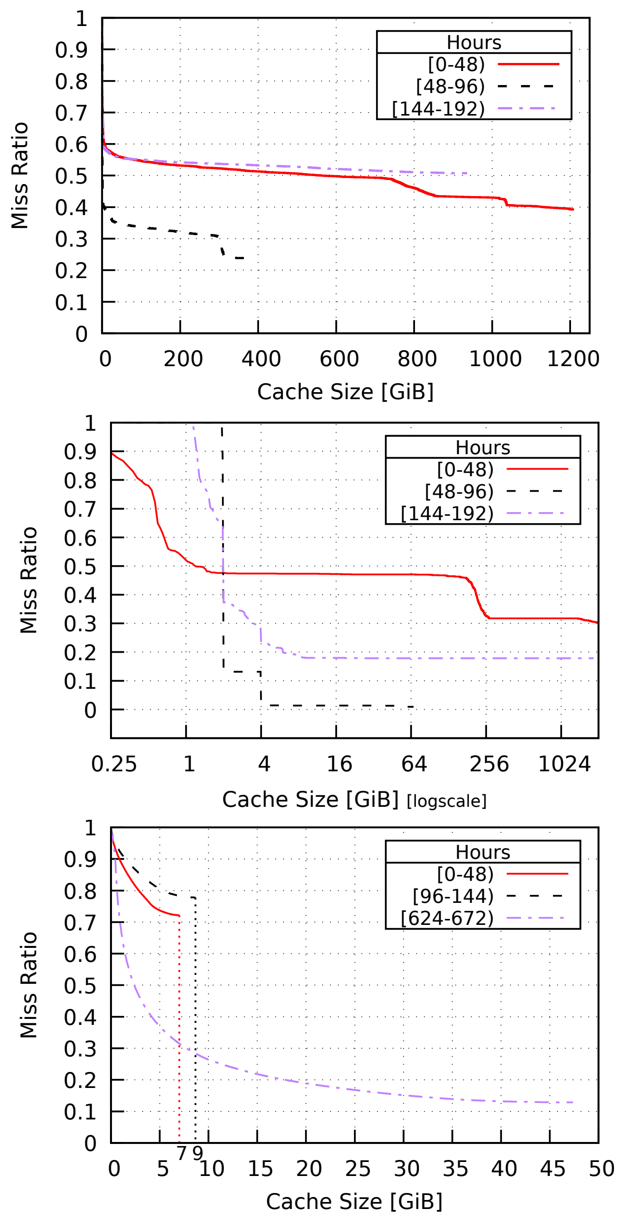


Figure 3.6: MRCs for 3 different publicly available workloads. **Top:** MSR combined workload. **Middle:** IBM workload #027. **Bottom:** Twitter workload #50. Each figure shows the MRCs for 3 different 48-hour time periods to demonstrate how caching requirements change in these periods.

extended and optimized CounterStacks algorithm as CounterStacks^{++}. As another application of our extended HLLs, we show how they can be utilized to estimate the unexpired WSS of TTL-endowed workloads in constant space (§3.5).

Finally, our experimental evaluation (§3.6) demonstrates the efficacy of our algorithms across 28 workloads from Twitter [39], totaling 113 billion accesses. The results indicate that accommodating TTLs leads to memory savings of 69% on average, and up to 99%. Our algorithms achieve over 99% accuracy on average when incorporating TTLs. The throughput levels of our extended MRC-generation algorithms are comparable to their original counterparts, maintaining their performance despite the accommodation of TTLs.

3.2 Mattson^{++}, Olken^{++}, and SHARDS^{++}

In this section, we extend the seminal Mattson (§3.2.1), Olken (§3.2.2), and SHARDS (§3.2.3) MRC-generation algorithms to accommodate TTLs.

3.2.1 Mattson^{++}

Mattson's algorithm was the first to generate the MRC of an access trace in one pass. It maintains the uniquely accessed objects in a stack ordered by access recency. For each access to an object, it calculates the stack distance as described in Section 2.2.1. Subsequently, a histogram of these stack distances is used to generate the MRC.

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 (i.e., TTL + access timestamp). 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.[1] (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.

3.2.2 Olken^{++}

Recall from Section 2.2.2 that Olken optimized Mattson's algorithm by using a balanced binary search tree instead of a stack, thus significantly improving the efficiency of MRC generation.

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 of the current access. Olken^{++} maintains the same time and space complexities as the original algorithm, although it increases space by maintaining an extra copy for the metadata of unexpired objects in *ET-PQ*.

We note that recording the eviction times in the tree nodes and evicting expired objects during tree traversal, similar to the strategy used 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 currently 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.

[1]We determine if an object is expired based on the timestamp of the current access, which corresponds to the current time of the simulation.

3.2.3 SHARDS^{++}

Background on the SHARDS algorithm was provided earlier in Section 2.2.4. In this section, we show how the two variants of SHARDS can be extended to support TTL attributes.

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 our Olken^{++} algorithm 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}. 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 $\langle \texttt{object}, F_i \rangle$ 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 F-PQ and Olken's data structures, then 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 (§3.2.2). 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 [55]. For example, we observed that FR-SHARDS performed poorly on most workloads from the SEC EDGAR dataset [116, 117], 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$_{adj}$ [55]. This modification estimates the number of accesses expected to be sampled for a given sampling rate and then adjusts the first bin 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 bin in the stack distance histogram addresses this issue. With the SEC workloads, we found that 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.

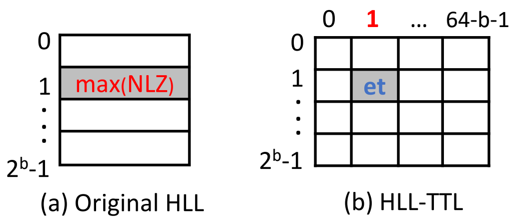


Figure 3.7: Bucket arrays of HLL and HLL-TTL (et: eviction time).

3.3 Extending HyperLogLog (HLL)

HLL is a cardinality estimation algorithm that efficiently approximates the number of distinct elements in a multiset [166, 167]. 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. Background on the HLL was presented in Section 2.4. Below, we show how the HLL can be extended to accommodate TTLs.

The original HLL does not provide the functionality to delete expired objects. We found that extending HLLs to support deleting expired objects to be non-trivial. The idea underlying our approach is to exclude 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. 3.7 (a) is extended to a 2-dimensional array as shown in Fig. 3.7 (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, which we call **HLL-TTL**. Their implementations are shown in Fig. 3.8. 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 array. 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. The reason behind keeping the larger eviction time is to ensure that a bucket is not reset until all objects that map to that bucket have expired.

With HLL-TTL, cardinality estimation (`Count`) is 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.[2] That is, assuming the NLZ increases as one moves right in the columns, we scan each row from right to left and identify the rightmost bucket with an unexpired value; the column index of that bucket is used to compute the harmonic mean.[3] The fundamental concept

[2]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 [166].

[3]As an optimization to eliminate the need for scanning, an extra array of 2^b buckets (bytes) can be used to track the rightmost valid index.

```
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): return counter estimate
    return $\alpha \cdot 2^{\overline{n+1}}$ where $\overline{n+1}$ = harmonic mean of $n_0..n_{2^b-1}$
    where $\alpha$ is a constant and
    $n_i = \max\{j \in [0, 64-b-1]\}$: 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-$b$-1
        HLL[i, j] = max(HLL$_1$[i, j], HLL$_2$[i, j])
```

Figure 3.8: HLL-TTL operations

behind this approach is that the largest NLZ will be utilized for cardinality estimation, similar to the original HLL. Maintaining an expiry time for each possible unexpired NLZ ensures that when the current largest NLZ expires, it will be replaced with the next largest NLZ that has not yet expired.

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 of 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 4KiB to 832KiB (using a byte per bucket for the original HLL instead of 6 bits). 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 [167]. Each row is replaced with a linked list, and buckets are allocated as needed.[4] Each bucket contains the tuple ⟨`NLZ, eviction time`⟩, and the list is ordered by NLZ. Whenever a bucket has an eviction time less than that of the next bucket in the list, it can be removed because it is guaranteed not to ever contribute to the results of the `Count` operator. While this reduces space usage in practice, the downside is that it incurs extra processing overhead for allocating and freeing buckets as well as for traversing the linked lists.

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 encountered. Each entry contains a ⟨`Hash(key), expiry time`⟩ tuple. This is similar to Microsoft's HLL implementation's *direct counting* approach [216]. 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.

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

3.4 CounterStacks^{++}

Background on CounterStacks was provided in Section 2.2.3. In this section, we extend CounterStacks to accommodate TTLs (§3.4.1), describe optimizations (§3.4.2), and discuss streaming (§3.4.3). We refer to the extended CounterStacks version that accommodate TTLs as CounterStacks^{++}.

3.4.1 TTLs Support in CounterStacks^{++}

To take TTLs into account, we replace CounterStacks's basic HLL with our HLL-TTL (§3.3). Simply integrating the extended HLLs into CounterStacks causes an issue related to downsampling. With TTL treatment, processing d accesses in a batch often introduces inaccuracies because some of the accesses in the batch 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 occurs at time 9 and is processed, it should be considered a cache hit. But if we complete processing the d accesses of the batch 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 then prematurely terminate processing accesses for the current batch whenever an object expires, thus processing fewer than d accesses in the batch. 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 CounterStacks^{++} 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 $8K$ unexpired eviction times are recorded in the priority queue; in our evaluation, we have found that limiting the recorded expiry times to $8K$ does not affect accuracy.

3.4.2 Optimizations

We optimized CounterStacks^{++} significantly, and these improvements also apply to the original CounterStacks algorithm for workloads without TTLs. This section details these optimizations and their effects on the CounterStacks algorithm using all the MSR workloads used in the CounterStacks paper (and other studies) [55, 64, 113]. Fig. 3.9 shows the speedups obtained using our optimizations, using HLL precision $b = 16$ and the HiFi setup. Our 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, respectively. 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 CounterStacks uses HLLs (§3.3), 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 and Microsoft, use `for loop` and `shift` operations to compute the NLZ.[5] We observed that a hardware supported instruction, `LZCNT` (leading zeros count), is more efficient. Using this instruction leads to an average speedup of 2× for CounterStacks.

[5]Redis (method hllPatLen()): https://git.io/JDBtu. Microsoft (method GetSigma()): https://git.io/JDBtR.

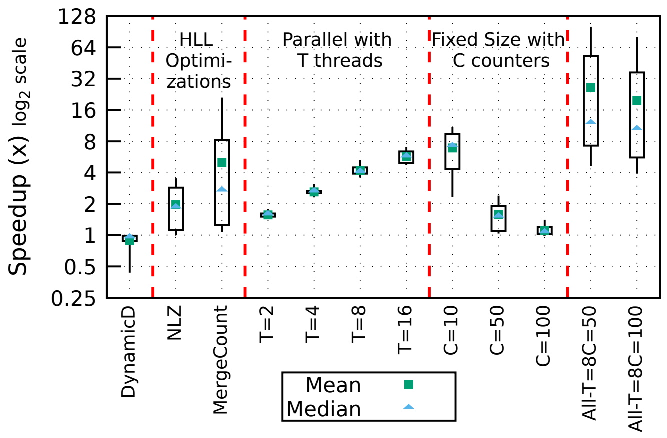


Figure 3.9: Speedup from each optimization against the original CounterStacks algorithm using all 14 MSR workloads, including the `combined` workload used in the CounterStacks paper [64]. Line ends show maximum and minimum speedups, while the box marks the 75th and 25th percentiles.

O2. New Specialized HLL Operation (`MergeCount`). As detailed in Section 2.2.3, after instantiating a new counter, up to d accesses are `Insert`ed into all existing counters. The count of each counter is then calculated using the `Count` operation (§2.4), 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 performance of CounterStacks by a factor of 5, on average.

O3. Fixed Size Overhead. We found that we can prune counters much more aggressively than what was originally proposed for CounterStacks and do so with marginal impact on accuracy. Recall that CounterStacks prunes a counter whenever its value is at least $(1 - \delta)$ times the older counter (§2.2.3). We found we can limit the number of counters to a constant, ***making CounterStacks/CounterStacks^{++} an MRC-generation algorithm with constant space overhead***, without serious impact on accuracy.

The strategy used to limit the amount of memory used is that 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 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 CounterStacks 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 of the resulting MRCs by an average of 6.34%, 0.26%, and 0.05%, respectively, but leads to an average speedup of 6.89×, 1.58×, and 1.11×, respectively.

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

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

3.4.3 Streaming

A major feature of CounterStacks and CounterStacks^{++} is their ability to stream intermediate counter information; e.g., by persistently storing the most recent counter values after the processing of each 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 computing the stack distances using the finite differencing scheme (§2.2.3) 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 complicates the above method. After processing a batch, some objects previously added to the existing counters might have expired. Because CounterStacks^{++} 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 CounterStacks^{++} streams counter values because they are static, which means that objects which expired during the processing of the batch are not accounted for. As a result, and based on the workloads we analyzed, the generated MRCs become so inaccurate that they become effectively 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 and 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. [64]) 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.

Table 3.1: Workloads used in this chapter.

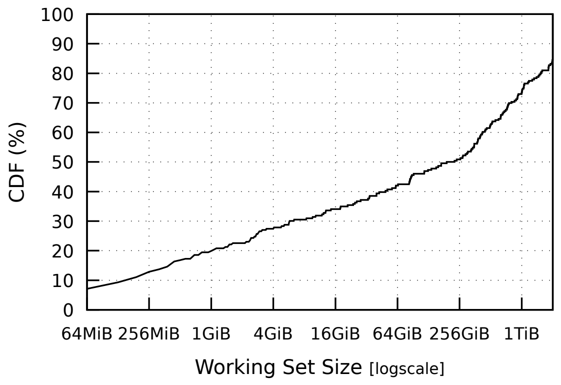
	# workloads	**# accesses**
MSR Cambridge [115]	13	434M
Twitter [39]	54	247B
Wikipedia [118]	1	2.5B
SEC EDGAR [116, 117]	58	25B
IBM [22]	98	1.6B
Tencent [56]	40 (5,584 traces)	30B

3.5 Exploiting HLL-TTL for WSS Estimation

Recall from Section 2.3 that opinions differ on the utility of WSSes and MRCs. 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 ratio. Conversely, for larger WSS values, the MRC provides insights into cache size versus miss ratio trade-offs, which are not available when using WSS. Fig. 3.10 (left) shows the WSS CDF across the 264 workloads from 6 different collections shown in Table 3.1 (without considering TTLs). Nearly 20% of these workloads have a WSS of less than 1GiB. By allocating 1GiB of memory to each cache serving these workloads, we achieve the lowest possible miss ratio. Fig. 3.10 (right) shows the WSS CDF across 28 workloads (described in §3.6) from the Twitter collection. Taking TTLs into account drastically reduces the largest unexpired WSS from over 2TiB to 64GiB. The effect of TTLs is substantial: 80% of the analyzed workloads have an unexpired WSS of 16GiB at most, compared to a WSS of 256GiB when disregarding TTLs. Furthermore, 60% of the workloads have an unexpired WSS of less than 6GiB when considering TTLs, compared to a WSS of 32GiB when disregarding TTLs. In cases like these, WSS alone often suffices for allocation guidance without needing to generate MRCs.

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 [166, 167]. In contrast, tools tailored for object membership testing (like Bloom and Cuckoo filters) are less efficient than HLL due to their broader scope [64].

To the best of our knowledge, our extended HLL-TTL (§3.3) is the first to accurately estimate unexpired WSS (taking TTLs into account) in constant space. Its worst-case memory usage for precision $b = 12$ is 832KiB (140KiB 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 GiB of memory to compute the exact WSS. Our HLL-TTL-based estimate is up to *five orders of magnitude* more memory-efficient.



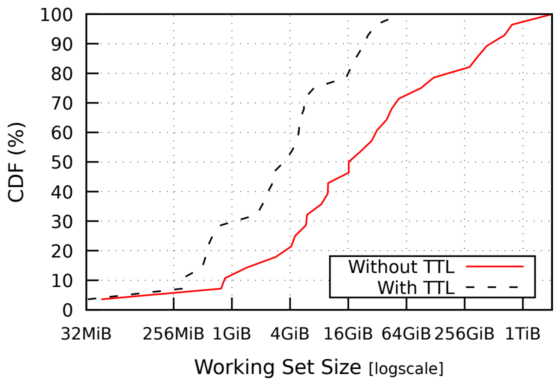


Figure 3.10: The left figure shows the WSS CDF for workloads listed in Table 3.1. The right shows the WSS CDF for 28 TTL-related Twitter workloads (with and without TTLs).

3.6 Evaluation

In this section, we show that:

1. significant memory savings can be achieved when sizing caches using TTL-aware WSSes 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 is comparable to that of the existing TTL-agnostic algorithms, and
4. our extended algorithms maintain consistent accuracy across varied configuration parameters.

We first consider WSS results (§3.6.1) and then MRC results (§3.6.2).

Experimental Setup and Workloads. Our experiments were conducted on a server equipped with an Intel 13900KS CPU and 128GiB of DDR5-4800MHz DRAM. Workloads were read from a Corsair PCIe 4.0 MP600 PRO 8TiB NVME SSD after they were formatted into a binary format. We use an in-house system that we developed to process the access workloads and generate the WSS and MRC using each of the algorithms discussed in this work. In all our generated MRCs, the stack distance histogram is divided into 64K bins, each representing 32MiB, supporting a cache size up to 2TiB; a similar approach was used in earlier studies [55, 86, 113].[6]

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 [110],[7] 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.[8] We only used the GET requests from the workloads, as in previous studies [64, 113, 144]. We did not consider SET requests because the SET requests in the workloads were captured when running a specific (undisclosed) cache size, and the

[6]AET uses logarithmic ranges [113].

[7]The recommended TTL clusters are: {6, 7, 11, 18, 19, 22, 25, 52} [110].

[8]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 [106, 109].

number and placement of SET requests would be different for different cache sizes. However, 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. An underlying assumption behind this approach is that a cache miss would be followed by a SET operation, which we found to be the case for most of the access traces we tested.

3.6.1 WSS Results

To evaluate the impact of TTLs on the WSS, we compared the exact unexpired WSS accommodating TTLs, WSS_{ttl}, and the exact WSS neglecting TTLs, WSS_{nottl}, across all 28 workloads. Notably, WSS_{ttl} fluctuates over time, as previously illustrated in Fig. 3.3 (see also §2.3). 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* (RS):

$$RS = \frac{WSS_{nottl} - WSS_{ttl}}{WSS_{nottl}} \quad . \tag{3.1}$$

Using $\mathbf{WSS_{ttl}}$ instead of $\mathbf{WSS_{nottl}}$ to size caches results in 69% memory savings, on average. Fig. 3.11 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} (242.3GiB) is 96% lower than the aggregate WSS_{nottl} (7.8TiB).

The estimation error of our HLL-TTL algorithm (when applied to workloads with TTLs) is in line with the expected error of the original HLL algorithm (when applied to workloads without TTLs). To evaluate the accuracy of HLL-TTL, we used the *Absolute Relative Error* (ARE):

$$ARE = \frac{|Exact\,WSS_{ttl} - Estimated\,WSS_{ttl}|}{Exact\,WSS_{ttl}} \quad . \tag{3.2}$$

We tested using different values for HLL precision parameter b within the range $[8-16]$. Fig. 3.12 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, $\sigma = \frac{1.04}{\sqrt{2^b}}$ [166]. The error of our HLL-TTL is in line with that expected error.

For throughput, Fig. 3.13 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 is increased is poorer cache locality. Given that in-memory caches typically operate at a throughput of less than a million access per second, these achieved throughput values are practical in real-world online scenarios.

For memory usage, the dense implementation of our HLL-TTL uses 832KiB, 1.59MiB, and 3.12MiB for precisions $b = 12, 13$, and 14, respectively. The dynamic implementation reduces this by over 80% but reduces throughput by a factor of 2; for precisions $b = 12, 13$, and 14, it uses 140KiB, 260KiB, and 479KiB, respectively.

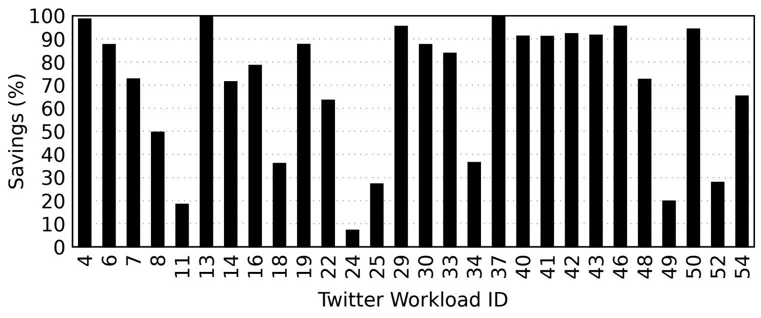


Figure 3.11: Relative memory savings across workloads when sizing memory with WSS_{ttl} instead of WSS_{nottl}. Savings range from 7.20% to 99.93%, with an average of 69.38%.

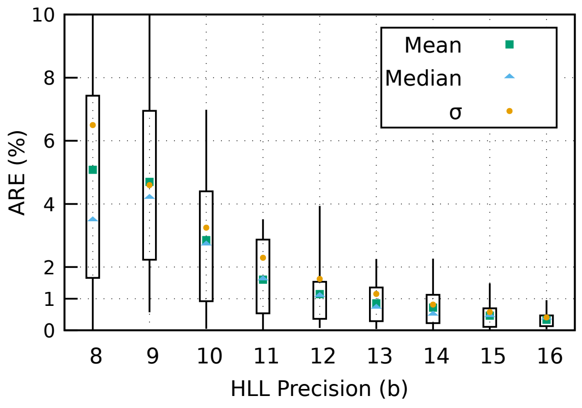


Figure 3.12: Sensitivity of the precision parameter, b, for HLL-TTL. σ is the theoretical Standard Error of the original HLL-NoTTL (when applied to workloads without TTL). In Figures 3.12, 3.13, 3.14, and 3.15, each line represents the range of results across the 28 workloads analyzed. The line's top and bottom represent maximum and minimum results. The box's top and bottom represent the 75th and 25th percentiles.

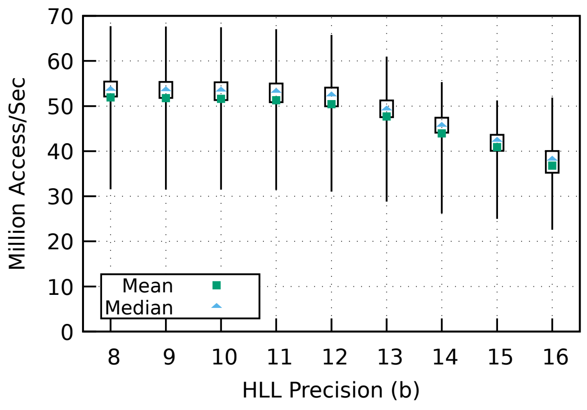


Figure 3.13: Throughput of HLL-TTL for different precisions.

3.6.2 MRC Results

MRCs are more complex to evaluate than WSSes because each MRC identifies a trade-off between the cache size and miss ratio. 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 ratio (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}, and we then quantified the savings using the same relative saving metric presented earlier, RS (Equation 3.1). 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 counterintuitive, 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. 3.2), making an increase in t, typically, less beneficial.[9] 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 4.49TiB to 243.66GiB), 93% (from 2.50TiB to 173.81GiB), 88% (from 1.08TiB to 131.13GiB), 83% (from 618.28GiB to 104.94GiB) for t = 0%, 0.1%, 0.5%, and 1%, respectively.

Existing MRC-generation algorithms which do not accommodate TTLs can misreport miss ratios 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* (MAD):

$$MAD = \frac{\sum_{1}^{B} |MRC_{ttl}[i] - MRC_{nottl}[i]|}{B} \tag{3.3}$$

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 32MiB) up to the point where both MRCs cease to change, extending the shorter MRC as necessary.[10] MRC_{nottl} deviates by up to 38% from MRC_{ttl}, with an average MAD of 5%, as shown in Fig. 3.14 (left). This shows that the vertical deviation in the MRC can be high when neglecting TTLs. Combined with the horizontal deviation presented earlier, where neglecting TTLs introduces a deviation of 69% in terms of the WSS of the workload, this illustrates that existing MRC-generation algorithms deviate significantly from the exact MRCs taking TTLs into consideration.

[9]For example, in cluster #41, with a tolerance of $t = 0\%$, the MRC_{nottl} and MRC_{ttl} show cache sizes of 286GiB and 25.6GiB, respectively, representing a 91% savings. With $t = 0.1\%$, the sizes of MRC_{nottl} and MRC_{ttl} become 143.6GiB and 17.9GiB, respectively, representing an 87.5% savings. Increasing t to 0.5% results in MRC_{nottl} and MRC_{ttl} sizes of 63.4GiB and 14.2GiB, respectively, representing a 77% savings. Finally, with $t = 1\%$, the sizes of MRC_{nottl} and MRC_{ttl} are 39GiB and 11.6GiB, respectively, representing a 70% savings.

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

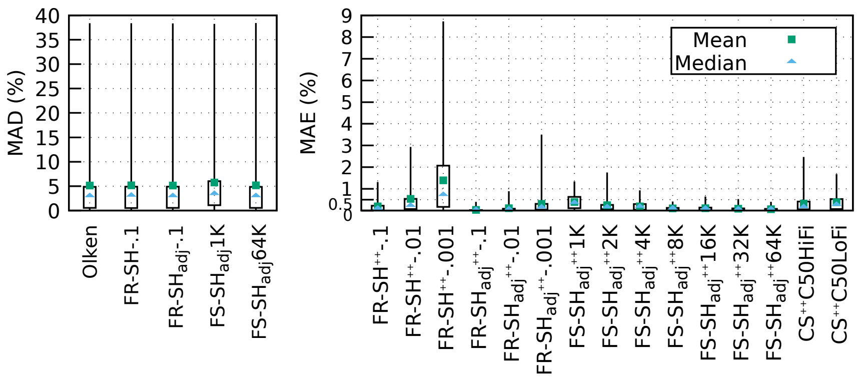


Figure 3.14: **Left:** Deviation of exact and approximate MRC_{nottl} from exact MRC_{ttl} in terms of MAD. **Right:** MAE between approximate MRC_{ttl} and exact MRC_{ttl} in terms of MAE. FR-SH refers to the fixed-rate SHARDS variant followed by the sampling rate used. FS-SH refers to the fixed-size SHARDS variant followed by the value of S_{max}. CS++ refers to CounterStacks++.

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*:

$$MAE = \frac{\sum_1^B |MRC_{ttl}[i] - MRC_{nottl}[i]|}{B} \tag{3.4}$$

between approximate MRC_{ttl} and the exact MRC_{ttl} obtained using our Olken^{++} algorithm. The MAE is widely used to quantify MRC errors [55, 64, 69, 84, 88, 90, 106, 113, 143]. Fig. 3.14 (right) shows the MAEs for different configuration parameters and algorithms 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^{++} with a sampling rate of 0.1%, 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 $64K$ reduces the average MAE to 0.09% and 0.06%, respectively, but increases memory usage by a factor of 8× and 64×, respectively. Third, CounterStacks^{++}, running with only 50 counters, has an average MAE of 0.32% and 0.39% for the HiFi and LoFi variants, respectively.[11]

The throughput of our extended algorithms is comparable to those of their original counterparts. Fig. 3.15 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 than the original FS-SHARDS$_{adj}$ for $S_{max} = 1K$, $8K$, and $64K$, respectively, due to the extra processing overhead introduced by evictions from two

[11]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 $2K$ increases the worst-case MAE from 1.3% to 1.7%. The reason for this is to be investigated. Nonetheless, average MAE decreases with higher precision as anticipated.

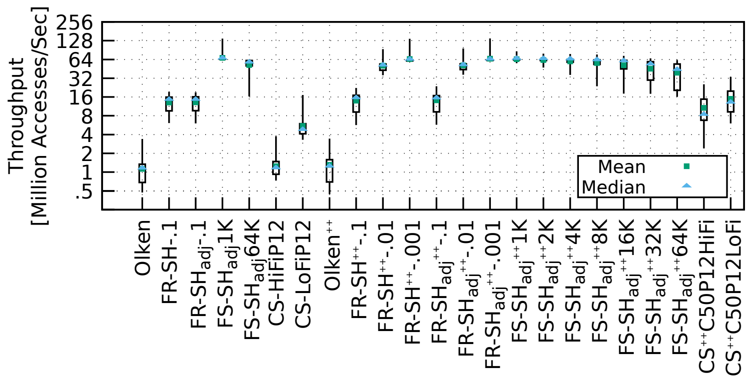


Figure 3.15: Throughput of all tested algorithms.

priority queues *F-PQ* and *ET-PQ* (§3.2.3). CounterStacks^{++} is 762% and 170% faster than the original CounterStacks for the HiFi and LoFi variants, respectively. This is due to the higher frequency of counter updates in the HiFi variant (every 60 seconds) compared to the LoFi variant (every 3,600 seconds), which makes the effect of parallelism more pronounced.

As a side observation, our study yielded a surprising result: the SHARDS^{++} variants exhibited a significantly higher accuracy compared to CounterStacks^{++}. This outcome was unanticipated as CounterStacks has generally been perceived as being more accurate than SHARDS, as even suggested by the authors of SHARDS [55].[12] To validate this unexpected finding, we undertook a comparative analysis between SHARDS and CounterStacks (*both without our extensions*), using the 264 workloads listed in Table. 3.1. This comparison confirmed that, without TTLs, SHARDS$_{adj}$ indeed outperforms CounterStacks in all aspects (accuracy, throughput, and memory usage). The primary advantage of CounterStacks over SHARDS, however, is streaming.

3.7 Related Work

MRCs have long served as a foundational tool for analyzing the relationship between cache size and miss ratio. As noted in Section 2.2, over the last five decades, a wealth of MRC-generation algorithms have been proposed [55, 56, 61–64, 66, 69, 71, 75–77, 82–92, 106]. In this chapter, we covered the seminal algorithms: Mattson, Olken, SHARDS, and Counterstacks. MRCs are widely utilized to optimize cache resources [9, 56, 65–78].

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 [11, 14, 60, 68, 93–106]. WSS has practical applications in cache management and has been a guiding metric for memory allocations [68, 102–104, 106, 212]. WSS estimation is related to cardinality estimation, which has been the subject of extensive research [162–172, 217]; both aim to measure statistics concerning the number of distinct objects [64].

[12]The reason behind this is that Waldspurger et al. compared SHARDS and CounterStacks using a single workload (i.e., the combined MSR workload).

Below, we present some other recent developments related to MRC-generation.

Other MRC-generation algorithms and applications. Zhang et al. presented OSCA, a system that utilizes MRCs generated for multiple caches in order to guide sizing decisions with the aim of maximizing the global hit rate [56]. To generate the MRCs, they introduced a new algorithm called RAR-CM. The primary downside of RAR-CM is that it requires $\mathcal{O}(M)$ space. Li et al. presented an approximate MRC-generation algorithm called APAC and used the MRCs to guide cache configurations [76]. Similar to RAR-CM, the main downside that limits APAC's practicality in online use cases is that it uses $\mathcal{O}(M)$ space.

MRCs are widely utilized to optimize cache resources [9, 56, 65–78, 218]. Dynacache and LAMA use MRCs to improve Memcached's slab allocation [9, 71]. They generate the MRC for each slab category and use it for optimization. MRCs are also utilized for performance cliff removal and cache partitioning: Talus and eMRC are two examples [75, 78, 87]. Talus utilizes the MRC to split the cache into two in order to remove performance cliffs [87]. Centaur uses MRCs to manage cache allocations for virtual machines [72]. Similar to the aforementioned OSCA and APAC, mPart and ORCA utilize MRCs to optimize cache allocations [56, 74–76].

Extensions to stack processing. Thompson [83] extended stack eviction policies to support write-back, and showed that the MIN policy is non-optimal if writes are considered. Similarly, Gecesi [219] introduced *joint stack processing*, an extension of stack processing to model multilevel storage hierarchies called staging hierarchies. Silberman [220] extended stack eviction policies to Delayed-Staging hierarchies, which introduce extra paths for parallel data flow between storage and the CPU. O'Krafka [221] extends stack eviction policies to support cache coherence schemes. Sugumar [222] studied different stack policies and how they could be parallelized.

Combining MRCs. Shedler and Slutz [223] proposed a technique to merge different MRCs, assuming the different MRCs are for workloads that access independent objects. A similar assumption was used later by CounterStacks [64]. To the best of our knowledge, we are the first to enable combining workloads that access common objects to generate a combined MRC (§3.4.3).

Metrics other than the stack distance and the use of sampling. Mattson, Olken, PARDA, CounterStacks, and SHARDS all use *stack distance* as their primary metric to generate the MRC using a histogram of encountered stack distances (see Section 2.2.1). Despite the established significance of stack distance as a critical metric in the study of program locality, its measurement incurs considerable overhead [55, 64, 113, 224]. Hence, other researchers used different metrics to generate the MRC. Below, we discuss a few examples. The primary theme behind these ideas is to introduce an extra layer of complexity that aims to approximate the stack distance and then use the same method to generate the MRC from an approximated stack distance histogram.

In 2016, Hu et al. utilized the accessed object's *reuse time* to generate MRCs with an algorithm called Average Eviction Time (AET) [113, 225, 226]. This approach comprises two primary techniques. First, it builds on Denning and Schwartz's work from 1972 [227], wherein they proposed a method to approximate the WSS from reuse time. Hu et al. used this to approximate the stack distances from a histogram of reuse times. Second, similar to SHARDS, AET utilized sampling to reduce the space overhead to a constant. Sampling has been widely used in the literature to improve the efficiency of

MRC-generation [55, 85, 88, 113, 173, 228–230]. Some of the major types of sampling used in the area include:

- *Address sampling:* This type of sampling is also called Spatial Sampling [55] and Set Sampling [231]. The main idea behind this type of sampling is to *sample a subset of the address space*. This approach has been widely used in program locality analysis, such as stack distance and reuse time approximation [55, 84, 88, 91, 224, 232–234]. Hu et al. observed that this approach could lead to imprecise MRCs based on their experience with AET because it might miss sampling highly popular objects. Although this observation is accurate, we found that Waldspurger et al.'s adjustment to SHARDS ingeniously addresses this sampling bias and generates highly accurate MRCs (as we discussed in Section 2.2.4 — see SHARDS$_{adj}$).

- *Access sampling:* This type is also known as fixed-interval sampling or time sampling [231]. The primary idea behind this approach is to sample a subset of the accesses (not the address space). For example, one might choose to sample every k-th access or sample the first y accesses every s seconds. The effectiveness of this approach is, however, highly dependent on the access pattern of the workload. Kessler et al. [231] studied set sampling and time sampling to predict the number of misses and concluded that set sampling is more effective than time sampling. They provide valuable insight into the drawbacks of both address sampling and access sampling, where the former has a high sensitivity to unrepresentative sampled sets (similar to SHARDS without Waldspurger et al.'s adjustment), and the latter needs long warmup periods [228, 231].

- *Random sampling:* Employs a random strategy to sample accesses [85, 113]. Hu et al. claimed that this approach performed better than access sampling and address sampling [113] — but again SHARDS$_{adj}$ significantly improves address sampling.

- *Reservoir sampling:* The primary goal behind reservoir sampling [235] is to randomly sample up to a constant number of objects, thus keeping the space requirement constant. This approach was used by several stack distance approximation studies [113, 234]. AET used this type of sampling by default, primarily to reduce the space requirement of their algorithm from $\mathcal{O}(M)$ to $\mathcal{O}(1)$ and improve the efficiency of their algorithm. Waldspurger et al. showed how address sampling can be adjusted such that the number of sampled objects remain constant, thus also making FS-SHARDS's space complexity $\mathcal{O}(1)$. Since AET has two levels of approximations, SHARDS provides more accurate results, and since both have the same memory usage requirements and performance, we only used SHARDS herein.

- *Non-statistical sampling:* Statistical sampling techniques (e.g., [236]) sample accesses or objects based on a predefined set of parameters such as the addresses to be sample, a sampling rate, or randomly selecting samples, all previously discussed. A more formal definition of sampling is provided in [237], and they used two statistical sampling methods in combination with one-pass algorithms to improve the efficiency of cache simulations. On the other hand, non-statistical sampling takes into consideration the behavior of objects to decide which accesses to sample. Non-statistical sampling was used by our research group [238], but SHARDS's address sampling has shown superior performance to it (in particular due to SHARDS$_{adj}$).

Coarse stack distance granularities and bucketing. Mattson's algorithm maintains a stack of accessed objects, ordered by access recency. Each stack element contains only one object, which enables identifying the exact stack distance as the position of the accessed stack element. To improve the efficiency of MRC-generation, Kim et al. [239] (1991) extended each stack element to become a bucket that contains several objects instead of one. They only allow identifying the stack distance at coarser granularity (i.e., the size of the bucket — they used powers of 2). In 2014, Saemundsson et al. extended this technique to group objects into variable-sized buckets with MIMIR [69]. Nonetheless, the space complexity of these approaches remains $\mathcal{O}(M)$. In the same year, Wires et al. used the same idea of buckets but ingeniously used HLL counters, which allowed them to reduce the space complexity to $\mathcal{O}(\log M)$, introducing a breakthrough in MRC-generation with CounterStacks (as described in Section 2.2.3). The same concept of MIMIR was later improved upon with QuickMRC [240], where a per-task stack distance histogram is generated and later used to generate a combined MRC for multiple tasks by combining the per-task histograms.

Hardware related. Fang et al. presented a method for approximating the stack distance for hardware caches [241]. MRCs are occasionally utilized in studies related to hardware caches; however, an alternative metric known as Misses Per Thousand Instructions (MPKI) is also frequently employed [67, 231]. These metrics can be used interchangeably in some contexts [84]. Qureshi et al. [67] utilized MPKI and proposed a utility-based cache partitioning scheme to distribute resources among different applications.

Conte et al. claimed that one-pass algorithms are limited to fully associative caches [242]. They presented a model to extend one-pass algorithms to support direct-mapped caches. Shi et al. claimed that most MRC generation methods (e.g., Mattson) target uniprocessors [243]. They presented a new stack processing method for multiprocessors. Wu et al. presented a stack algorithm for a specific type of virtual memory caches called virtually indexed caches [244]. Hwu et al. presented a stack algorithm that takes context switching into consideration [245]. Zhou et al. [66] used hardware monitors to generate MRCs at fine granularity with negligible overhead and presented another method for operating systems that has 7% overhead. They used the MRC for different applications: (i) MRC-directed memory management for multi-programmed systems and (ii) MRC-directed memory energy management. Wu et al. also presented stack algorithms for shared memory multiprocessors [246]. He et al. presented FractalMRC to generate MRCs for multiprocessors based on the Fractal model [247].

3.8 Concluding Remarks

This chapter addressed the critical issue of extending tools to support the sizing of in-memory caches in modern cloud environments. The key issue we focused on was incorporating TTL handling into the state-of-the-art MRC-generation and WSS estimation algorithms. The importance of modeling TTLs for in-memory caches was demonstrated. We adapted the Mattson, Olken, and SHARDS algorithms to handle TTLs, while 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 showed the effectiveness of our algorithms.

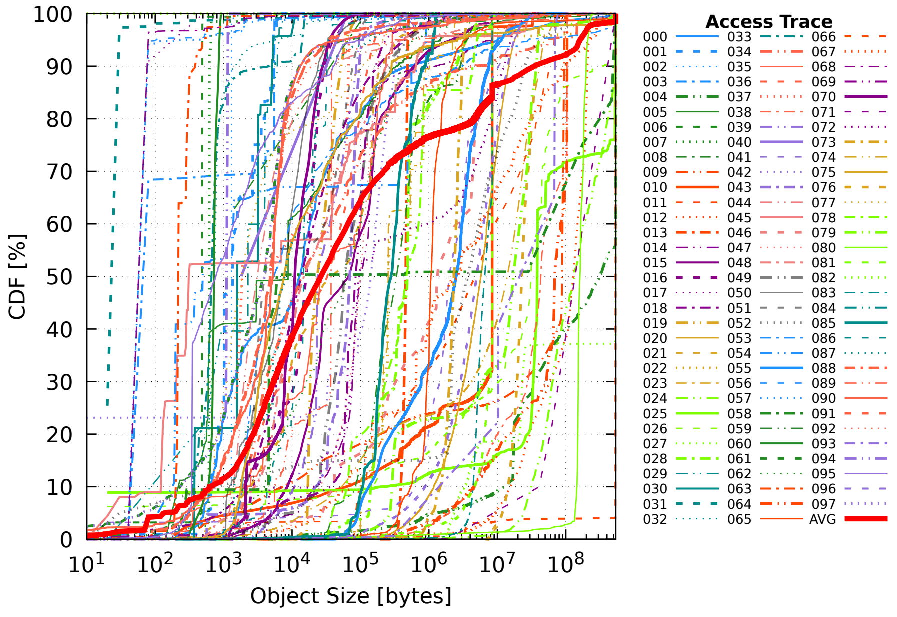
Chapter 4

Accommodating Heterogeneous Object Sizes in MRCs and WSSes

The prior chapters have underscored the critical role of in-memory caches and the central challenge they present in terms of optimal sizing. With the widespread deployment of these caches and the significant amount of DRAM dedicated to in-memory caching, identifying an optimal cache size is paramount for reducing operational costs. Prior chapters also introduced the MRC and the WSS as the primary tools for guiding cache sizing decisions. The MRC provides insights into the trade-off between cache size and miss ratio, while the WSS determines the necessary cache size for minimizing the miss ratio.

Historically, the assumption of uniform object sizes has dominated research on online eviction policies [61, 62, 64, 113, 129, 173, 248–261]. This focus was initially due to policies being designed for caches that operate on uniformly sized disk blocks. However, the landscape has shifted with in-memory caches now often storing key-value pairs of heterogeneous sizes, as highlighted by real-world workloads [22, 39, 56, 115–118]. In-memory caches such as Memcached and Redis allocate memory for each object in proportion to the size of the object being cached. Earlier in the Introduction, we showed the object size distribution for the MSR and Twitter datasets in Figures 1.5 and 1.6. In addition to these, Figures 4.1 and 4.2 show the object size distribution for additional datasets from IBM, Tencent, SEC, and Wikipedia.

The inclusion of heterogeneous object sizes introduces significant complexity into the analysis of eviction policies and MRC generation. Regarding eviction policies, for instance, although the OPT eviction policy [61], introduced in the 1970s, was proven to be optimal, it is based on the premise that a uniform object size is used; adapting it to heterogeneous sizes has proven to be an NP-complete problem and thus insolvable within polynomial time [24, 262].



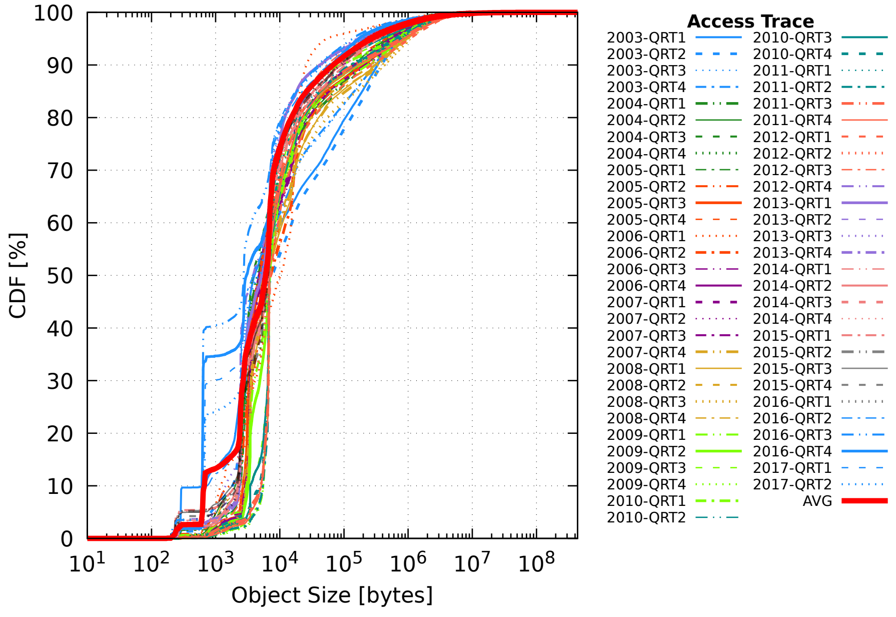
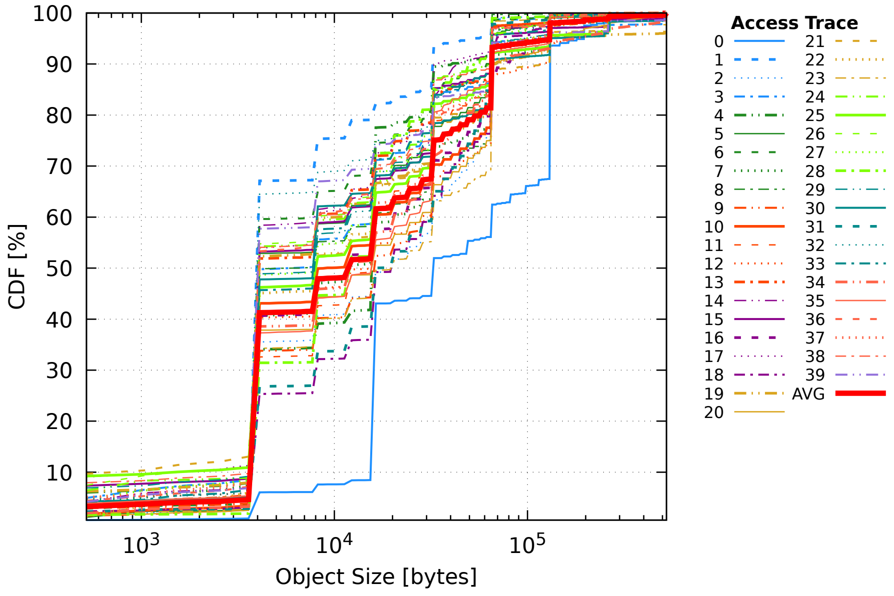


Figure 4.1: Object size CDFs for the access traces from IBM (top) and SEC (bottom).



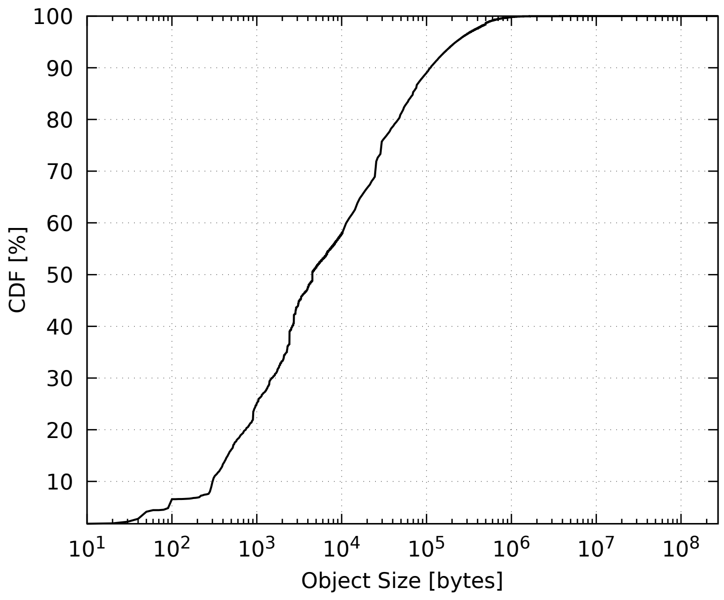


Figure 4.2: Object size CDFs for the access traces from Tencent (top) and Wikipedia (bottom).

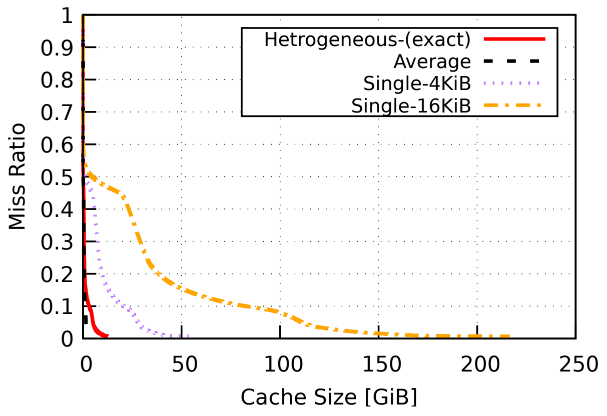


Figure 4.3: Effect of object size treatment on MRCs for the Twitter workload #34. The figure shows four different object size treatment strategies. The exact curve shows the MRC when generated using the object size as provided in the access trace. The Average curve shows the MRC when generated using the running average of the object sizes in the access trace. Lastly, we show two uniform object size treatments, one using 4KiB per object and the other using 16KiB per object.

Taking heterogeneous object sizes into account matters when generating MRCs. Figures 4.3 and 4.4 show that not taking heterogeneous object sizes into account can lead to inaccurate MRCs for the workloads from the Twitter and IBM datasets. In addition, Fig. 4.5 shows that previous simplistic attempts to take heterogeneous object sizes into account also lead to substantial errors. Two simple strategies are prevalent in prior work for accommodating heterogeneous object sizes. The first approach simply assumes all objects have the same size. For instance, SHARDS assumes uniform 16KiB objects [55] (other researchers have assumed 4KiB objects). The second approach converts each read request into as many uniformly sized objects as needed to store the data belonging to the reference; e.g., a reference to an 11KiB object would be converted to three 4KiB access references. A similar approach is used by CounterStacks [64, 119].[1] Although it has not been used in earlier studies, we have also added to the aforementioned figures the MRC generated using the average object size, which we found can be computed online efficiently using only an extra 16 bytes (as we describe in Section 4.2).

The prevalence of objects with heterogeneous sizes in caching has become increasingly evident, but has been largely overlooked by prior research [55, 64, 90, 113, 144]. Recently, and specifically after 2018, some studies explored accommodating heterogeneous object sizes in MRC-generation, with the common claim that past MRC-generation algorithms only supported uniform sizes [90, 114, 144]. For instance, Pan et al. claimed that *"Past cache modeling techniques are typically limited to a cache system with a fixed cache line/block size"*, and presented an extension to support heterogeneous object sizes in an approximate MRC-generation algorithm called AET [114, 144]. Similarly, Carra et al. claimed that *"We observe that it is possible to extend Olken's approach to MRC computation to the case of heterogeneous size contents maintaining $\mathcal{O}(\log M)$ complexity per request. [...] We suspect that this approach may be known, but we were not able to find it described elsewhere."* [114]. However, some studies in the literature, such as the SHARDS paper [55], claimed in 2015 that Traiger and Slutz [263] extended Mattson's algorithm to support heterogeneous object sizes in 1971. One

[1]The CounterStack paper does not mention this explicitly, but Drudi's Master's thesis [120] does, and other researchers have claimed that CounterStacks uses 4KiB objects as well [55, 113].

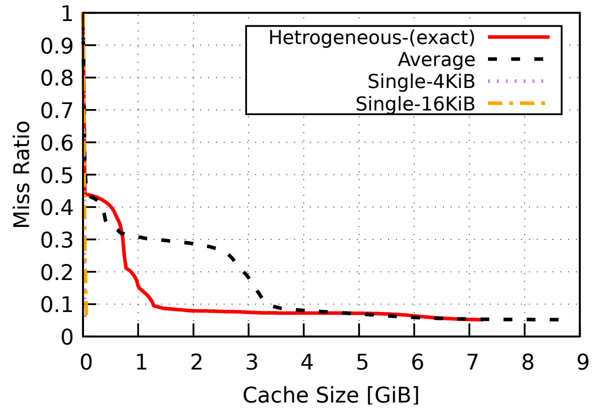


Figure 4.4: Effect of object size treatment on MRCs for the IBM workload #59. We utilize the same object size treatments as described earlier in the caption of Fig. 4.3.

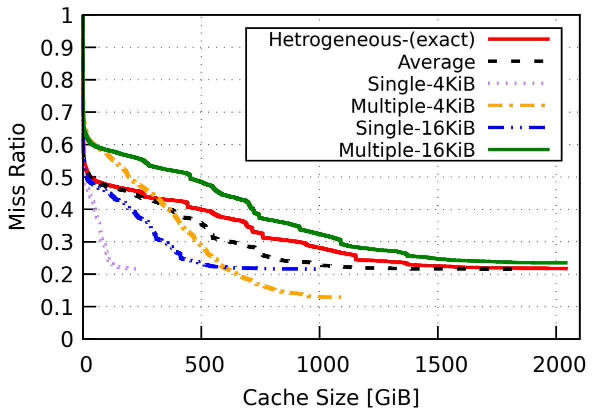


Figure 4.5: Effect of object size treatment on MRCs for the combined MSR workload.

of the potential reasons behind this discrepancy is that a copy of Traiger and Slutz's paper is not available online despite being cited many times.[2] Given that the preponderance of eviction policies and MRC-generation studies also assume uniform object sizes, accommodations for heterogeneous objects could be easily overlooked.

This chapter describes how Mattson's and Olken's exact algorithms can be extended to support heterogeneous object sizes (§4.1). Further, we describe the straightforward adaptation of the SHARDS algorithm to accommodate heterogeneous object sizes. Lastly, we describe how HLL counters can be extended to support heterogeneous object sizes, where they are extended to count bytes instead of the count of objects (§4.2). This extended HLL is then utilized to extend CounterStacks which to date only supports uniform object sizes (§4.3).

4.1 Heterogeneous Objects with Mattson, Olken, and SHARDS

This section describes accommodating heterogeneous object sizes in Mattson's algorithm (§4.1.1), Olken's algorithm (§4.1.2), and the SHARDS algorithm (§4.1.3).

[2]Surprisingly, even the publisher (IBM) did not have a copy when we contacted them.

4.1.1 Mattson

Our comprehensive analysis of papers that cited Traiger and Slutz, specifically [55, 63, 65, 82, 83, 93, 148, 219–222, 232, 236, 237, 242, 244–246, 264–278], revealed that Traiger and Slutz adapted Mattson's algorithm to support heterogeneous object sizes. We describe how to accommodate heterogeneous object sizes in Mattson's algorithm.

Recall from the Background Section 2.2.1 that Mattson's algorithm maintains a stack of unique objects accessed. In order to accommodate accesses to heterogeneous object sizes, the stack elements are extended to also record the size of the object they are referencing. This increases the amount of space required by a constant factor. Then, instead of counting the number of elements during stack traversal to obtain the stack distance, the object sizes of the elements being traversed in the stack are summed to obtain the stack distance. Thus, instead of the stack distance identifying the number of distinct objects accessed since the current object was last accessed, the stack distance now identifies the aggregate size of the distinct objects considered. The compute complexity is the same as that of the original algorithm.

Accommodating heterogeneous objects violates the inclusion property. All the aforementioned studies failed to recognize that processing heterogeneous object sizes violates Mattson's inclusion property for any cache size smaller than the largest object size used in the workload. We provide the simple access sequence of $\{A(4\text{KiB}), B(8\text{KiB}), A(4\text{KiB})\}$ to demonstrate the inclusion property violation. After the second access (to B), an 11KiB cache will only contain object B, while any cache smaller than 8KiB and larger than 4KiB will contain A only. For workloads with small objects, this would not significantly affect the results, but for workloads with large objects such as IBM workloads where objects can be in the gigabytes [22], this could be problematic, as LRU could experience Belady's anomaly [154] as FIFO does. Typically, in-memory caches limit the largest object sizes that can be stored. For instance, the default largest size in Memcached is 1MiB, in Redis is 512MiB, and in Eytan et al.'s study, they limited the largest object size to 4MiB [22].

4.1.2 Olken

Recall from the Background Section 2.2.2 that Olken optimized Mattson's algorithm by using a balanced binary search tree instead of a stack. In 2020, Carra et al. proposed that Olken's algorithm can be extended to support heterogeneous object sizes, which requires changing its internal data structure [90, 114]. Our investigation revealed a small section that can be easily overlooked in Olken's Master thesis [62], which introduced an extension to support heterogeneous object sizes without changing the original data structure (i.e., AVL tree). The extension is relatively simple, and it works as follows.

The weight of each node in the tree is redefined to be equal to the size of its object plus the sum of the weights of its child nodes. The stack distance is calculated as the sum of the sizes of all objects referenced by nodes with a greater timestamp than the time of the current access. This modification leaves the compute complexity of the algorithm intact, and can be applied as is to our TTL-aware variant, Olken^{++}, described in Section 3.2.2.

4.1.3 Shards

Recall from the Background Section 2.2.4 that Shards was proposed to improve the efficiency of MRC generation by employing sampling. The only modification required to accommodate heterogeneous object sizes in Shards (and our TTL-aware Shards^{++}) is to use the extended version of Olken that supports heterogeneous object sizes as described in the previous section.

4.2 Accommodating Heterogeneous Sizes in HyperLogLog

Recall from the Background Section 2.4 that the HLL is one of the most widely used cardinality estimators that approximates the number of unique objects in a multiset. In contrast to exact algorithms that require memory proportional to the cardinality, HLL estimates the cardinality with high accuracy using practically constant space. We are interested in HLLs because they enable a number of useful applications. First, they effectively capture a workload's WSS if a reference to each object being accessed by the workload is added to the multiset. Given the constant space requirement of HLLs, this becomes one of the most space-efficient methods to approximate a workload's WSS.

Second, HLL counters can be merged to estimate the number of unique objects in the union of two multisets, given the HLL values of the two multisets. For example, two HLLs can capture the WSS of two workloads, which can then be combined to obtain the WSS of the combined workload. As another example of the merge capabilities of HLLs, they can be used to capture the WSS of a workload during, e.g., each 1 hour period; the individual HLLs obtained from consecutive hours can then be merged to identify the WSS of the same workload over any longer time period (e.g., 6 hours, 12 hours, 1 day, 1 week, 1 month, etc.), with little computational overhead.

Finally, HLLs can also be used to generate MRCs, as demonstrated by CounterStacks [64]. When MRCs are generated from HLLs, then combined MRCs can also be generated from the HLLs. For example, MRCs can be constructed for each (non-overlapping) hour to obtain more granular insight, and MRCs for longer time periods can also be generated. Alternatively, for shared caches, a per-tenant MRC can be constructed to understand the behavior attributed to each tenant, and these MRCs can be combined to understand the overall behavior of the shared cache.

When using a uniform object size, the WSS can be estimated as the cardinality reported by the HLL times the uniform object size used. However, if each element in the multiset has a numeric parameter associated with it (e.g., weight, size, etc.), then one may be interested in the sum of the parameter values of the distinct elements in the multiset instead of their distinct count. For our application, we are interested in the sum of the object sizes — the cache size needed to store the objects. In Section 4.2.1, we describe how to achieve this without TTLs, and in Section 4.2.2, we describe it with TTLs.

4.2.1 Accommodating Heterogeneous Object Sizes

Recall from Section 2.4 that the internal data structure of the HLL is an array of bytes (Fig. 4.6-a). Each of these bytes records the maximum NLZ in the hash of the accessed object keys for the objects assigned to the bucket. To accommodate heterogeneous object sizes, we could instantiate an HLL for each distinct object size encountered. But since this would significantly increase the space overhead, a simple, effective estimation is to maintain an HLL for each power of 2 (or a factor f) size only. The resulting extended HLL is shown in Fig. 4.6-b.

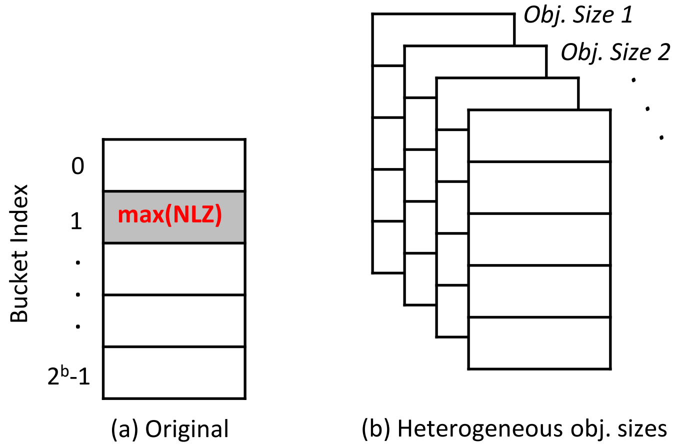


Figure 4.6: Comparison between the structure of the original HLL and our extended HLL that supports heterogeneous object sizes.

If an HLL is kept for each power of 2, then we need $(l - s + 1)$ HLLs to accurately estimate the WSS of an access trace that has object sizes that range between 2^s to 2^l bytes. For example, to cover object sizes between 2 bytes and 4MiB, we need 22 HLLs with a total size of 88KiB to achieve an accuracy of over 98% (precision $b = 12$).

Recall from Section 2.4 that the HLL has three primary operations: `Insert`, `Merge`, and `Count`. Now, we explain how these operations are extended. To `Insert` an object into the extended HLL, we use the object size to determine which HLL the access should be inserted into. Once identified, we follow the exact same `Insert` process as of the original HLL. `Merge`ing two extended HLLs, should result in an extended HLL that includes the objects sizes maintained in both source HLLs.

For the `Count` operation, we iterate through the maintained HLLs and accumulate each HLL `Count` times the object size it represents. Since each HLL represents an object size of a power of 2, this might increase the variance since object sizes could have significant variability. To reduce the variance, we track the average object size of the `Insert`ed accesses in each HLL. Tracking the average object size can be efficiently computed online using Welford's method [279], as shown in Eq. 4.1, where B represents the currently accessed object size and n represents the number of accesses (which is incremented after each access). This requires only an extra 16 bytes: a `double` for the average and a `uint64` for the count, n.

$$Average = \frac{B - Average}{n + 1} + Average \tag{4.1}$$

To find the WSS, the reported count of each HLL is multiplied by the average object size of its corresponding HLL, as shown in Eq. 4.2.

$$WSS = \sum_{i=1}^{l-s+1} \texttt{Count}(HLL[i]) \times HLL[i].Average \tag{4.2}$$

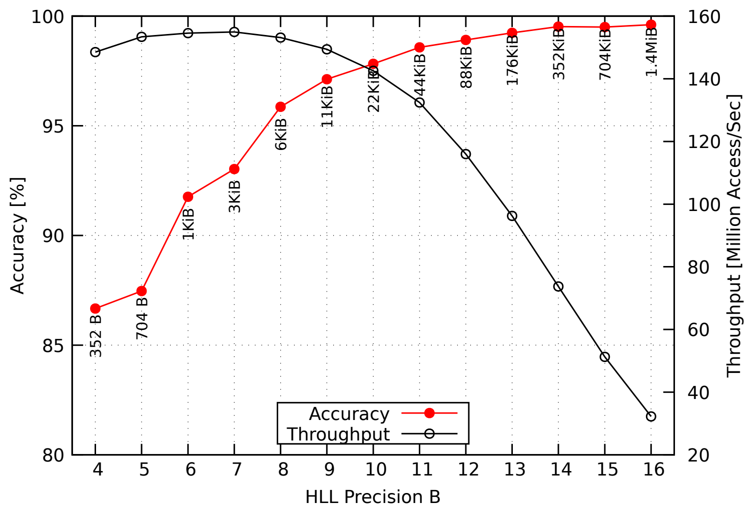


Figure 4.7: WSS estimation accuracy and throughput for our extended HLL that supports heterogeneous object sizes based on the MSR workloads.

Fig. 4.7 shows the accuracy, throughput, and needed space to estimate the WSS while accommodating heterogeneous object sizes using the MSR workloads. The accuracy is measured by comparing the estimated WSS to the exact WSS measured using a hash table. The figure shows that 352KiB is needed to estimate the WSS with an accuracy of over 99% (i.e., b=14). In practice, precision 12 is sufficient, which uses 88KiB of memory and achieves over 98% accuracy. The reason behind the decrease in throughput as the precision increases is due to poorer cache locality.

In contrast, finding the exact WSS requires space linear in the number of distinct objects, which is impractical for workloads that access many objects. For example, the combined Twitter dataset includes 247 billion accesses to 25 billion distinct objects, which requires 25B $\times$ 12 bytes $=$ 279GiB to compute the exact WSS (assuming each object can be tracked using 12 bytes: 4 for the object size and 8 for the hash of the key). Thus, the estimate based on our extended HLL uses up to 5 orders of magnitude less space while being highly accurate.

4.2.2 Accommodating Heterogeneous Object Sizes and TTLs

In Section 3.3, we presented our extended HLL-TTL that supports TTLs by automatically removing objects from the multiset when their TTLs expire. This allows our algorithm to handle modern workloads where TTLs are often used. In this section, we show how we extend HLLs to support both heterogeneous object sizes and TTLs. The HLL-TTL is shown in Fig. 4.8-b. To add heterogeneous object sizes with HLL-TTL, we add a third dimension to the HLL in order to capture different object sizes as described earlier in Section 4.2.1, which would result in the structure shown in Fig. 4.8-c.

The HLL operations on the 3-dimensional array are straightforward adaptations of those for the 2D operations. However, the `Count` operation is adjusted to report the sum of the average object size times the cardinality estimate for that specific object size. Welford's running average is used to compute the average per object size plane as described earlier (Eq. 4.1).

Fig. 4.9 compares the accuracy and throughput for the basic HLL (that supports uniform object size without TTLs) versus our extended HLL (that supports heterogeneous object sizes and TTLs).

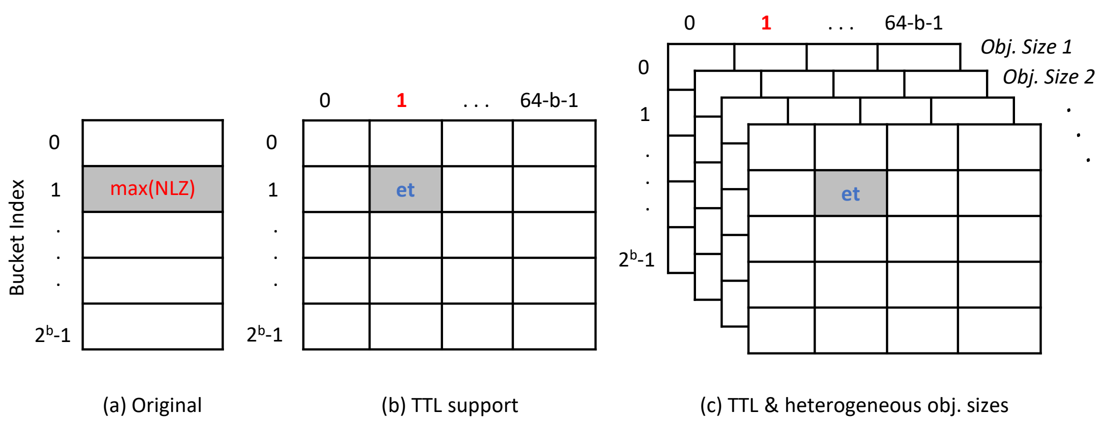


Figure 4.8: Comparison among the structures of the original HLL, HLL-TTL, and HLL-TTL with support for heterogeneous object sizes. (et: eviction time)

The results shown are for the largest recommended workload from Twitter (cluster52), which contains 11.6 billion GET accesses. The accuracy is measured by comparing the estimated WSS and the exact WSS obtained using a hash table (based on the configuration as shown in the figure). The accuracy of the basic HLL is shown compared to the exact WSS using a uniform object size and not taking TTLs into consideration to show how our extensions compare to the original HLL's accuracy.

The results show that precision 12 is sufficient to achieve over 98% accuracy. The dense implementation of the extended HLL requires 11.5MiB, while the dynamic implementation (described in Section 3.3) requires 1.06MiB. In contrast, calculating the exact WSS without TTLs and uniform object sizes requires 1.35GiB of memory just to maintain each distinct object's 64-bit key, and the exact variant accommodating TTLs and heterogeneous object sizes requires 2.4GiB memory to maintain each unexpired distinct object's timestamp, key, size, and TTL attribute. The decrease in throughput shown in Fig. 4.9 is because increasing the precision for the extended HLL results in poorer cache locality. However, the throughput of the basic HLL is consistent and does not decrease when increasing the precision because the size of the basic HLL is less than 64KiB for all precisions, thus having good cache locality.

4.3 Extended CounterStacks

In this section, we first describe how to adapt CounterStacks to accommodate heterogeneous object sizes (§4.3.1). We refer to this extended version as CounterStacks$_{HOS}$. Then, we describe how to accommodate both heterogeneous object sizes and TTLs (§4.3.2). We refer to this extended version as CounterStacks$^{++}_{HOS}$.

4.3.1 Accommodating Heterogeneous Object Sizes

To accommodate heterogeneous object sizes in CounterStacks, we replace the traditional HLL counter used in CounterStacks with our extended HLL that supports heterogeneous object sizes shown in Fig. 4.6(b). Each extended HLL counter contains several object size planes, where each plane is essentially a traditional HLL with buckets used to count the number of objects of a specific

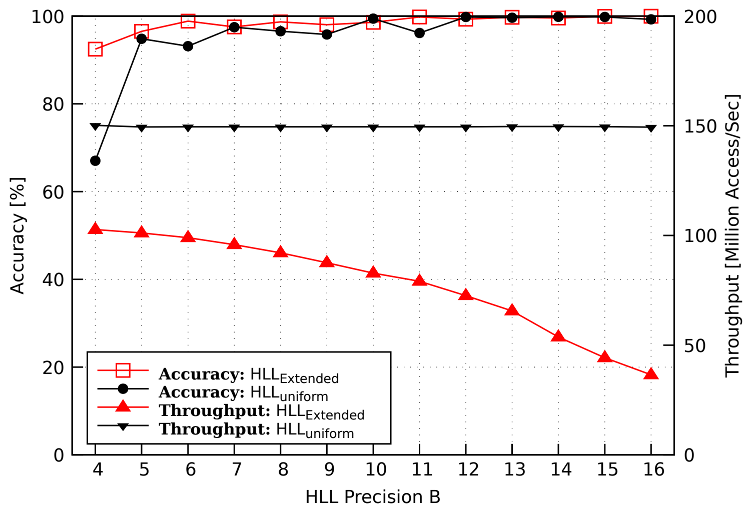


Figure 4.9: HLL precision vs. the estimated WSS accuracy and throughput using an HLL with uniform object sizes without TTLs and our extended HLL with heterogeneous object sizes and TTLs. The largest recommended Twitter workload (`cluster52`) was used for these results.

size. Since this will significantly increase the number of HLLs used, the algorithm will become slow. We optimize the extended CounterStacks algorithm to accommodate heterogeneous object sizes while remaining more efficient than the original algorithm and using constant space.

Accesses in a workload's access trace are processed in the following manner. For each access, a new counter is instantiated. It is initialized to contain one object size plane for each object size previously encountered, and each of those object size planes is initialized to 0. Then, if the access is to an object size not yet encountered, a new object size plane for that object size is added to each counter in the stack. Finally, the object size planes for the target object size in each counter are updated to reflect the distinct number of objects of that size that have been encountered since the object size plane was instantiated.

Fig. 4.10 shows an example of how heterogeneous object sizes are treated in CounterStacks$_{HOS}$. For the first access in the trace, $A(4K)$, a counter c_1 is added. An object size plane for 4KiB objects, c_1^{4KiB}, is added to c_1 and set to $|\{A\}| = 1$. This object size plane will maintain the number of accesses to distinct objects of size 4KiB encountered since c_1 was initialized. For the next access, $t(\text{2KiB})$, a new counter, c_2, is added with an object size plane for 4KiB initialized to 0. Then, an object size plane for 2KiB objects is added to c_1 and c_2, initialized to $|\{t\}| = 1$. Hence, counter c_1 will have object size plane c_1^{4KiB} equal to $|\{A\}|$ and object size plane c_1^{2KiB} equal to $|\{t\}|$.

To calculate the stack distance for an access to an object of size s, if the value of object size plane c_i^s did not increase while the value of the next object size plane, c_{i+1}^s, increased, then the stack distance is equal to the sum of all object size planes of counter c_i times their respective object sizes; i.e., $\sum_{\forall c \in c_i}(c \cdot s)$. For example, for the second access to B(4k), shown in bold in Fig. 4.10, object size plane c_3^{4KiB} did not increase while the next object size plane c_4^{4KiB} increased from 2 to 3. Hence, the stack distance is equal to the value of c_3^{4KiB} (i.e., 3) multiplied by 4KiB plus the corresponding 2KiB object size plane value, c_3^{2KiB} (i.e., 2) multiplied by 2KiB for a total of 16KiB ($= 3 \times \text{4KiB} + 2 \times \text{2KiB}$). This is recorded in the stack distance histogram. This process is the same finite differencing scheme described in Section 2.2.3 but adjusted for heterogeneous object sizes.

$\longrightarrow$ time (j) $\longrightarrow$

	A (4k)	t (2k)	B (4k)	C (4k)	u (2k)	A (4k)	t (2k)	u (2k)	**B (4k)**	B (4k)
c_1^{4KiB}	1	1	2	3	3	3	3	3	3	3
c_1^{2KiB}	-	1	1	1	2	2	2	2	2	2
c_2^{4KiB}		0	1	2	2	3	3	3	3	3
c_2^{2KiB}		1	1	1	2	2	2	2	2	2
c_3^{4KiB}			1	2	2	3	3	3	$\boxed{\mathbf{3}}$	3
c_3^{2KiB}			0	0	1	1	2	2	**2**	2
c_4^{4KiB}				1	1	2	2	2	$\boxed{3}$	3
c_4^{2KiB}				0	1	1	2	2	2	2
c_5^{4KiB}					0	1	1	1	2	2
c_5^{2KiB}					1	1	2	2	2	2
c_6^{4KiB}						1	1	1	2	2
c_6^{2KiB}						0	1	2	2	2
c_7^{4KiB}							0	0	1	1
c_7^{2KiB}							1	2	2	2
c_8^{4KiB}								0	1	1
c_8^{2KiB}								1	1	1
c_9^{4KiB}									1	1
c_9^{2KiB}									0	0
c_{10}^{4KiB}										1
c_{10}^{2KiB}										0

Figure 4.10: Example of CounterStacks$_{HOS}$ with heterogeneous object sizes.

Optimizations

The heterogeneous object size implementation described above increases the number of HLLs by a factor equal to the number of object sizes encountered. This makes the implementation impractical, given that each HLL consumes $2^4 - 2^{16}$ bytes (depending on precision). We introduce a number of optimizations to significantly reduce the number of HLLs needed.

O1. Limiting the number of object sizes

As a first optimization, we reduce the number of object size planes by grouping objects based on their sizes, similar to the technique used in our extended HLL (§4.2.1). By default, we use the next power of 2. With this, each object size plane now tracks accesses for objects that have a size which rounds to the same next power of 2 to group accesses. To improve accuracy, we track the average object size within each object size plane and use that value to represent the object size plane. This is computed as described earlier in the extended HLL section using Welford's method (Eq. 4.1).

Table 4.1: Example to illustrate different pruning techniques used in *CounterStacks*$_{HOS}$.

Object Size	c_1	c_2	c_3
4KiB	$c_1^{4KiB} = 100$	$c_2^{4KiB} = 99$	$c_3^{4KiB} = 99$
2KiB	$c_1^{2KiB} = 1,000$	$c_2^{2KiB} = 100$	$c_3^{2KiB} = 100$

Table 4.2: Effect of different types of pruning when processing the `combined` MSR workload.

Pruning Type	**# Counters**	**# Size Planes**
No Pruning	689	4,202
Counter	138	1,014
Counter and Size Plane	72	172

O2. Counter pruning

In CounterStacks$_{HOS}$, a counter is pruned (along with all its object size planes) when each of its object size planes' values is at least $(1 - \delta)$ times the value of the corresponding object size plane of the next older counter. As an example, consider the three counters depicted in Table 4.1, each with two object size planes. With pruning delta $\delta = 0.01$, counter c_3 can be pruned entirely because $c_3^{\text{4KiB}} \geq (1 - \delta) \cdot c_2^{\text{4KiB}}$ and $c_3^{\text{2KiB}} \geq (1 - \delta) \cdot c_2^{\text{2KiB}}$.

O3. Object size plane pruning

Further efficiencies can be obtained by pruning individual object size planes. An object size plane can be pruned whenever its value is at least $(1 - \delta)$ times the corresponding object size plane in the next older counter. For example, if we consider counter c_2 in Table 4.1, its object size plane c_2^{4KiB} is equal to $(1 - \delta)$ times c_1^{4KiB} and hence is a candidate for pruning. However, it cannot be removed because counter c_2 depends on it for correct results. Instead, we replace object size plane c_2^{4KiB} with a pointer to c_1^{4KiB}, the next older corresponding object size plane. As a result, c_2^{4KiB} consumes less space and no longer needs to be updated when processing new accesses, yet c_2 contains approximate values of each of its object size planes, albeit indirectly. Whenever an object size plane in a counter, $c_i^{y\text{KiB}}$, is pruned and replaced with a pointer to $c_k^{y\text{KiB}}$, then all object size planes pointing to the pruned counter must be updated to now point to the same object size plane $c_k^{y\text{KiB}}$.

Because object size planes now may have pointers pointing to them, care must be taken not to free an object size plane that other object size planes are pointing to. For this reason, we include a *reference count* within each object size plane to keep track of the number of pointers pointing to it. Therefore, an object size plane can only be freed when the reference count is 0. A counter can be pruned when all of its object size planes are pointers to object size planes in older counters.

Counter and object size plane pruning reduces the number of HLL counters maintained. Table 4.2 shows the effect of different types of pruning used in CounterStacks$_{HOS}$ for the `combined` MSR workload. Furthermore, since object size pruning makes two consecutive object size planes point to the same object size plane if they satisfy the pruning condition, the counter pruning algorithm can be optimized only to run when object size plane pruning removes at least one object size plane.

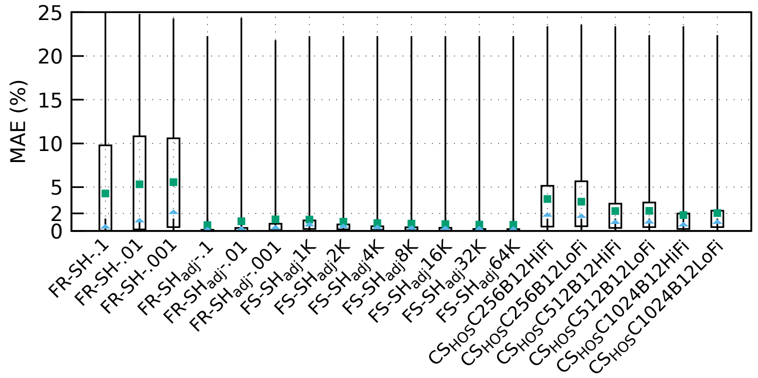


Figure 4.11: MAEs for the algorithms tested with heterogeneous object size support (without TTLs)

O4. New object size optimization

Whenever a new object size is encountered for the first time, new object size planes need to be added to each of the existing counters, initialized to 0. All but one of these object size planes will be pruned by the next object size plane pruning operation. Hence, as an optimization, when a new object size is encountered for the first time, a new object size plane is added to the oldest counter only, and a pointer to that object size plane is added to all other counters.

Performance analysis

We evaluated CounterStacks$_{HOS}$ using the access traces described in Table 3.1 (page 47), and compared it to the extended SHARDS variant. We measured the accuracy of the algorithms we tested using the MAE by comparing the approximate MRC to the exact MRC generated by Olken's algorithm that we extended to support heterogeneous object sizes. We used the same server specifications described in Chapter 3. Each line in Figures 4.11 and 4.12 represents the results for 264 workloads.

Fig. 4.11 shows the accuracy of the algorithms tested. We make the following observations. First, SHARDS$_{adj}$ should always be used instead of SHARDS as it has far better accuracy. For instance, FR-SHARDS with a sampling rate $R = 0.001$ has an MAE of 5.57% on average and a median of 1.99%. Conversely, FR-SHARDS$_{adj}$ with the same sampling rate reduced the MAE to 1.33% on average and the median to 0.22%. Our second observation is that SHARDS has significantly higher accuracy than CounterStacks$_{HOS}$. For example, when using up to 1,000 HLLs in CounterStacks$_{HOS}$, the achieved MAE is 1.81% and 2.05% on average for the HiFi and LoFi variants, respectively. The median MAE is 0.57% and 0.85% for the HiFi and LoFi variants, respectively. However, using FS-SHARDS$_{adj}$ with S_{max} equal to 1,000 objects achieves a significantly lower MAE of 1.31% and a median of 0.57%. Increasing S_{max} to 4,000 objects reduces the MAE to 0.9% on average and the median to 0.25%.

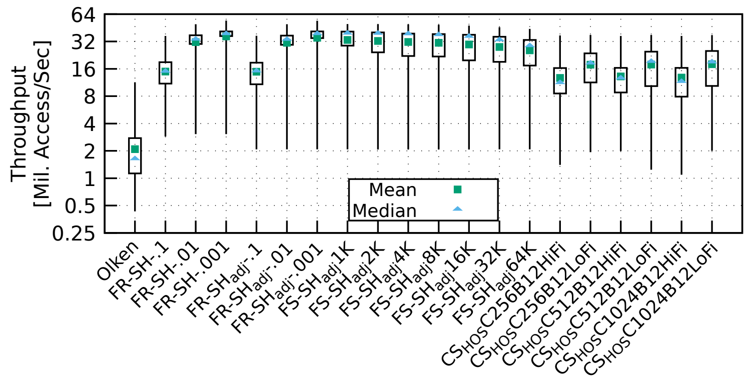


Figure 4.12: Throughput for the algorithms tested with heterogeneous object sizes (without TTLs)

Fig. 4.12 shows the throughput of the algorithms tested. All these algorithms are the extended ones that support heterogeneous object sizes. The exact Olken algorithm can process 2 million accesses per second, on average. For FS-SHARDS, increasing the value of S_{max} reduces performance but improves accuracy. For example, for $S_{max} = \{1K, 4K, 16K, 64K\}$, it achieves a throughput of $33.3M$, 31.7M, 29.6M, and 25.5M, on average, respectively. Similarly, increasing the number of HLLs used in CounterStacks$_{HOS}$ has less of an effect on performance than with SHARDS. For example, for the LoFi variant of CounterStacks$_{HOS}$, using 256, 512, and 1,024 HLLs achieves a throughput of 17.8M, 17.9M, and 18M. The reason behind this counterintuitive slight variation could be attributed to the fact that with a smaller number of HLLs, more pruning iterations could occur to enforce the constant space overhead, thus making the algorithm slightly slower with a smaller number of HLLs.

4.3.2 Accommodating Heterogeneous Object Sizes and TTLs

In Section 3.4, we presented CounterStacks^{++}, which accommodates TTL attributes. To extend CounterStacks^{++} to support heterogeneous object sizes, we follow the same approach described earlier for accommodating heterogeneous object sizes in CounterStacks$_{HOS}$. This introduces multiple object size planes by switching the HLL-TTL to become as shown in Fig. 4.8(c). We refer to the extended algorithm that supports heterogeneous object sizes and TTLs as CounterStacks$^{++}_{HOS}$.

We evaluated CounterStacks$^{++}_{HOS}$ and compared the results with those of Olken^{++} and SHARDS^{++}; all extended to support heterogeneous object sizes and TTLs. The MAE was used to measure the errors between the exact MRCs generated by Olken^{++} and the approximate MRCs generated by CounterStacks$^{++}_{HOS}$ and SHARDS^{++}. We used the same 28 workloads used in the evaluation of Chapter 3, so each line in Figures 4.13 and 4.14 represents the results for 28 access traces.

Fig. 4.13 shows the MAE for the evaluated algorithms. One observation is that SHARDS$_{adj}$ significantly affects the results. For example, for FR-SHARDS^{++} with a sampling rate of $R = 0.001$, FR-SHARDS$^{++}_{adj}$ reduces the MAE from 1.5% to 0.55% on average. Similarly, the median MAE is reduced from 0.7% to 0.2% on average. All FS-SHARDS$^{++}_{adj}$ variants achieve an MAE of less than 1%. For instance, using S_{max} = 1,000 achieves an MAE of 0.66% on average. Increasing S_{max} to 64,000 reduces the MAE to 0.09% on average but increases space usage by a factor of 64×.

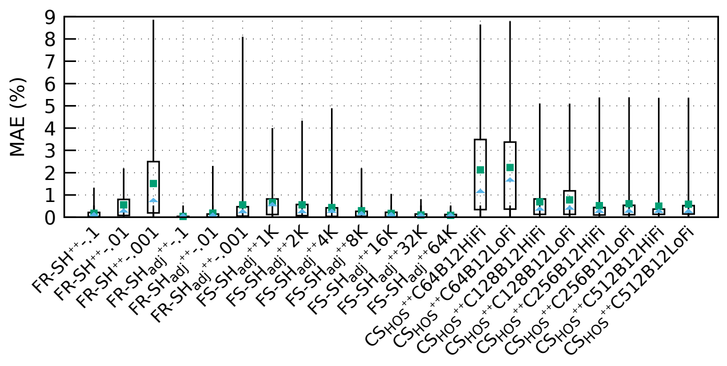


Figure 4.13: MAE for the extended algorithms with heterogeneous object sizes and TTL support

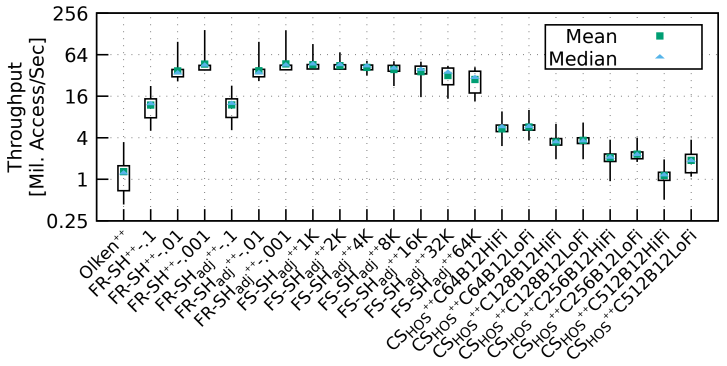


Figure 4.14: Throughput for the extended algorithms with heterogeneous object sizes and TTL support

For CounterStacks$^{++}_{HOS}$, we tested the HiFi and LoFi variants using various configurations based on the number of HLL-TTL counters used, ranging from 64 to 512 counters.[3] With 64 counters, the HiFi and LoFi variants have an MAE of 2.12% and 2.23%, on average, respectively. Doubling the number of HLL-TTL counters to 128 reduces the MAE significantly to 0.69% and 0.78% for the HiFi and LoFi variants, respectively. The reason for this improved performance is that, as the number of HLL-TTL counters decreases, our algorithm performs pruning with increased δ values to maintain constant space usage, which increases the stack distance estimation error. The best MAE achieved for the HiFi and LoFi variants is 0.5% and 0.59%, on average, respectively, when using 512 counters.

Fig. 4.14 shows the throughput for the evaluated algorithms. All these algorithms are extended to support heterogeneous object sizes and TTLs. The exact Olken algorithm can process 1.3 million accesses per second, on average. For FS-Shards^{++}, increasing the value of S_{max} reduces the throughput but improves accuracy. For example, for $S_{max} = \{1K, 4K, 16K, 64K\}$, it achieves throughputs of $45.8M$, $41M$, $36.2M$, and $27.8M$, on average, respectively. Similarly, increasing the number of HLL-TTL counters used in CounterStacks$^{++}_{HOS}$ has a significant effect on throughput. Using only 64 counters, the HiFi and LoFi variants achieve average throughputs of $5.3M$ and $5.7M$, respectively, with an MAE of 2.12% and 2.23%, respectively. Increasing the number of counters to 128 reduces the throughput to $3.4M$ and $3.6M$, respectively, with an MAE of 0.69% and 0.78%, respectively.

[3]Each object size plane is equivalent to a single HLL-TTL counter.

Chapter 5

Interval-based Historical Analysis of Cache Workloads with HistoChron

In-memory caches such as Memcached and Redis have played a pivotal role for decades in improving the performance of distributed systems. However, configuring these caches to operate efficiently remains a challenging task, especially when considering the dynamic nature of modern workloads, where caching requirements can change significantly over time.

This chapter introduces HistoChron, a novel methodology with an associated GUI that enables efficient interval-based historical analysis of caching workloads. HistoChron generates a separate stack distance histogram and HyperLogLog (HLL) counter each minute and persists them for historical analysis. HistoChron needs just 24MiB of storage space weekly, showcasing its space efficiency. Histograms and HLL counters over any time interval can be combined to obtain the Miss Ratio Curves (MRCs), the Working Set Size (WSS), as well as statistics on the number of cache requests and number of accesses to distinct objects for the target interval. The information generated, in turn, enables the identification of seasonal or diurnal patterns, and it enables detailed postmortem analysis of unexpected cache behaviors.

We evaluated HistoChron on over 5,000 cache access traces from six real-world datasets encompassing more than 300 billion accesses and, when combined, span a total duration of 18 years of cache operations. Our results show that HistoChron produces exact MRCs with overheads that are on par with state-of-the-art exact MRC-generation algorithms. We also present a lower overhead variant of HistoChron that generates approximate results with a mean error of less than 1%. Finally, we show how to adapt Miniature Simulations to enable HistoChron to provide interval-based MRCs, working set sizes, and cache request statistics for any eviction policy.

5.1 Introduction

Despite the fact that in-memory caches have been used productively for decades, many organizations do not understand how best to configure them for their workloads and when to reconfigure them. In effect, they often treat in-memory caches as black boxes and either apply coarse trial-and-error methods or simply configure the cache with a large amount of memory, hoping for the best [27, 50, 53–58]. Worse, assessing whether in-memory caches are operating as expected is particularly challenging. When a perceived issue occurs, it is difficult for operators to diagnose whether the issue is real and, if so, what might have caused it. Consider the following questions that operators of in-memory caches might reasonably pose:

- An operator observes significantly increased response times of their service for a period of time and after analyzing system logs, the operator deduces the cause was a surge of requests to the back-end storage service. They suspect the surge was due to a sudden increase in the cache miss rate. Was this actually the case, and if so, what might have caused the increased miss rate?
- A service operator recalls that their operations struggled to handle the high volume of incoming requests just before the holiday season of the previous year. Would increasing cache sizes this year help mitigate such behavior, and if so, by how much should the cache size be increased?
- To save on costs, a service operator is considering using caches with significantly less memory over the weekend when traffic tends to be lower. How much can the cache size be reduced, if at all, without significantly affecting the cache hit rate?
- An operations team is considering co-locating two in-memory cache servers currently running on separate physical servers onto the same physical server. Are performance issues expected to arise?
- Are there time periods where it would be beneficial to switch to an alternative eviction policy such as FIFO, which incurs lower overheads?

Existing tools and methods are ill-suited to help answer these and similar questions. One methodology is to collect and store cache access traces for postmortem simulations and analysis. However, storing such traces would consume significant storage space. For example, Twitter's 153 Memcached access traces, spanning 1-4 weeks, consumed 80TiB of storage space [125]. It would be difficult to justify keeping traces for a prolonged period, especially when operating many caches. This is further exacerbated by the exponential growth of in-memory datasets [25–27].

Another well-known tool is the MRC, which depicts the expected cache miss ratio as a function of the cache size [55, 61]. However, MRCs are limited in the information they convey. For example, an MRC generated for cache accesses that occurred over a period of a month provides no information on the trade-off between cache size and miss rate for a specific set of consecutive hours or days, and it provides no insight into the operation of the cache when an issue is suspected to have occurred. Conversely, MRCs generated, say, for each day separately, cannot be combined to understand the trade-off between cache size and miss rate for a week's worth of cache traffic. While it is possible to simultaneously and continuously generate daily, weekly, and monthly MRCs, the memory and compute overheads would likely negatively impact the cache's operation when done online, especially if these MRCs need to be generated for multiple eviction policies.

Finally, the WSS is a tool that measures the aggregate size of distinct objects accessed over a given period of time [11, 14, 60]. However, the WSS suffers from limitations similar to those of MRCs in that WSS information for smaller time intervals cannot be extracted from a WSS generated over a longer period of time, and WSS information generated over smaller intervals cannot be combined to obtain the WSS of caches over a longer time period.

In this chapter, we introduce ***HistoChron***, a new methodology that can help answer the types of questions listed earlier. Despite its simplicity, HistoChron's novel approach offers surprising power. Our work on HistoChron was motivated by two observations. First, for key eviction policies such as LRU, the MRC of a workload is generally constructed from a histogram of stack distances obtained from processing the workload's cache accesses (using an algorithm such as Olken [62]) after which the histogram is typically discarded. We argue that the histogram should be retained (instead of the MRC) as it contains additional useful information, including a count of the number of accesses, a count of objects accessed the first time, as well as the information needed to construct the MRC.

The second observation is that histograms obtained over multiple non-overlapping time periods can be usefully combined. For example, if we have a separate histogram of a workload's stack distances for each successive minute, then 60 consecutive histograms can be combined to obtain a histogram that accurately represents the cache accesses over the corresponding hour. (As noted above, this is not the case for MRCs: multiple MRCs cannot be usefully combined.)

The key idea behind HistoChron is to separately collect cache access statistics over uniform, consecutive time intervals and then persist that information for historical analysis. The default time interval we have selected is one minute, so cache access statistics are collected and retained for each consecutive one-minute interval (but this can be overridden through a configuration parameter).

As such, HistoChron's chronological sequence of cache access statistics enables ***interval-based analysis***; i.e., it enables us to retroactively analyze the workload for any arbitrary time interval at one minute granularity. For example, HistoChron enables us to zero in and analyze the behavior of the cache over the previous few weekends or holiday seasons, and it allows us to zero in on the specific hour in which anomalous cache behavior is suspected to have occurred. Histogram sequences of different workloads can also be combined, which allows us to analyze the effects of combining multiple workloads (assuming the workloads' keys are disjoint).

The access information collected and retained each minute by HistoChron is comprised of:

1. a histogram of stack distances encountered and
2. a cardinality estimator such as the HLL, which counts the number of distinct objects accessed.

Together, this information allows us to obtain for each minute interval:

1. the number of access requests,
2. the number of objects accessed the very first time,
3. the number of distinct objects accessed,
4. the MRC, and
5. the WSS.

Section 5.3 details how the collected information can be combined to obtain these metrics without needing the entire trace. The reason we use an HLL counter is that the number of distinct objects accessed within a time interval cannot be extracted from the stack distance histograms, yet it is needed to identify the WSS of the workload in that interval. HLL counters, like stack distance histograms, have the attractive property that they can be combined; e.g., the HLL counter of each minute can be combined to obtain a count of the distinct objects accessed over a period of one hour.

An important question is whether HistoChron can be practically realized with acceptable overheads. With respect to storage overhead, by implementing a number of optimizations (§5.3), we were able to achieve a storage requirement of 3.4MiB per day and 24MiB per week while maintaining one minute epoch granularity (98.7KiB per day and 691.1KiB per week for one hour granularity). This space overhead is orders of magnitude less than storing the access traces directly. HistoChron's memory overheads are on par with those of the state-of-the-art MRC-generation algorithms such as Olken [62] that identify the stack distance for each cache access and record them in a histogram, in addition to 2.4KiB for each HistoChron's entry that has not yet been persisted. Finally, HistoChron's compute overhead is comparable to that of Olken's algorithm.

We developed a system with a Graphical User Interface (GUI) that uses HistoChron's chronology of access statistics (§5.4). The GUI allows the user to zoom in/out to any time interval (at minute granularity), for which the GUI graphically displays the MRC, the WSS, the number of cache requests per minute, and the number of distinct objects accessed per minute. It also displays the aggregate number of accesses and the number of distinct objects accessed during the selected interval. This information is obtained by combining the corresponding information from the chronology.

The access counts and MRCs obtained from combining consecutive HistoChron histograms are *exact*, and correspondingly, HistoChron's memory and computation overheads are in line with exact MRC-generation algorithms such as Olken. However, Olken is considered to have memory and computational overheads that might be too high for online use (where the stack distance of each cache access is identified when the access occurs). As a result, a number of *approximate* MRC-generation algorithms have been proposed that are more efficient [55, 64, 113]. In Section 5.5, we show how HistoChron can be adapted to work in conjunction with SHARDS, thus significantly decreasing HistoChron's memory and computational overheads. While SHARDS makes HistoChron's MRCs and metrics approximate, we show in Section 5.7 that mean errors are low, i.e., <1% on average.

Another limitation of most MRC-generation algorithms is that they can only model eviction policies that are referred to as *stack-based eviction policies* (e.g., LRU and OPT) [55, 61, 88], which have the property that if an object exists in a cache of size s, then it must also exist in all larger sizes. To model non-stack-based policies such as FIFO, MRU, ARC [150], and S3-FIFO [184], it is necessary to simulate caches of different sizes. However, such simulations have high computational and memory overheads. Waldspurger et al. introduced *Miniature Simulations* that applies SHARDS to cache simulations to make them tractable [88]. In Section 5.6, we show how Miniature Simulations can be adapted for use with HistoChron, thereby accommodating non-stack-based policies.

Organization. We start by further motivating the need for interval-based analysis. In Section 5.3, we present HistoChron. Our HistoChron GUI is presented in Section 5.4. We show how SHARDS can be incorporated with HistoChron in Section 5.5. In Section 5.6, we show how non-stack-based policies and heterogeneous object sizes are accommodated. Section 5.7 presents the experimental evaluation, highlighting the exactness of our MRCs generated by HistoChron and its storage space efficiency. We also show the high throughput of our approximate HistoChron variant that uses SHARDS (capable of processing over 100 million accesses per second) while maintaining high accuracy (with a mean error <1%). We close with related work and concluding remarks.

5.2 Motivation

The compelling case to be made for HistoChron and interval-based analysis becomes apparent when examining real-world workloads. The fact that cache workloads are often highly variable over time [9, 64, 88, 106, 136] makes interval-based analysis particularly relevant.

Consider IBM's workload #`079` [22]. The MRC obtained from a week's worth of accesses is depicted in Fig. 5.1, top left. The figure shows that a cache size of 250GiB is needed to achieve the minimal miss rate of 12%. This information is useful for a "configure and forget" strategy. However, if resource efficiency is the objective, then zooming in on individual days can be helpful, as shown in Fig. 5.2. The figure shows significant day-to-day variance in cache size requirements for this workload. On the first day (Wednesday), the cache size required to achieve the minimal miss rate is roughly 60GiB. For Thursday, Friday, and Saturday, the cache size needed to achieve the minimal miss rate for each day is significantly higher: 120GiB, 250GiB, and 120GiB, respectively. The following three days each require a cache size of approximately 60GiB again to achieve the minimal miss rate.

To better understand why Thursday and Friday require larger cache sizes, we turn our attention to the number of accesses and number of distinct objects accessed, as shown in Fig. 5.1, top right. The pattern of cache requests begins to change late Wednesday when the number of distinct objects accessed increases significantly. This could indicate a scanning pattern that might be due to an analytics job being run at that time. This increase in accesses to distinct objects increases the WSS from 100GiB to 250GiB in a few hours, as shown in Fig. 5.1, bottom. The effect of this suspected scanning pattern subsides on Sunday, and for the rest of the week, the cache size needed to achieve the optimal miss rate goes back to roughly 60GiB. (Note that this more detailed information is not extractable from the weekly MRC.) By performing interval-based analysis over previous weeks, one could identify whether this is a regularly occurring pattern and, if so, consider potential remediation actions.

It is not uncommon for workloads to exhibit significant variability in their WSSes and MRCs when considering shorter time intervals. Recall the example of the `prn` workload from MSR, previously shown in the Introduction (Fig. 1.8 and Fig. 1.9), to underscore the significant variability in workload behaviors over short time intervals. This case illustrated how the perceived cache size requirement for minimizing miss ratios could dramatically fluctuate, from needing less than 15GiB during weekends to a surge up to 70GiB on weekdays, particularly after a potential sudden scanning pattern that drastically increased the workload WSS. Such findings emphasize the critical role of interval-based analysis tools like HistoChron in accurately determining cache size requirements amidst the dynamic nature of modern workloads.

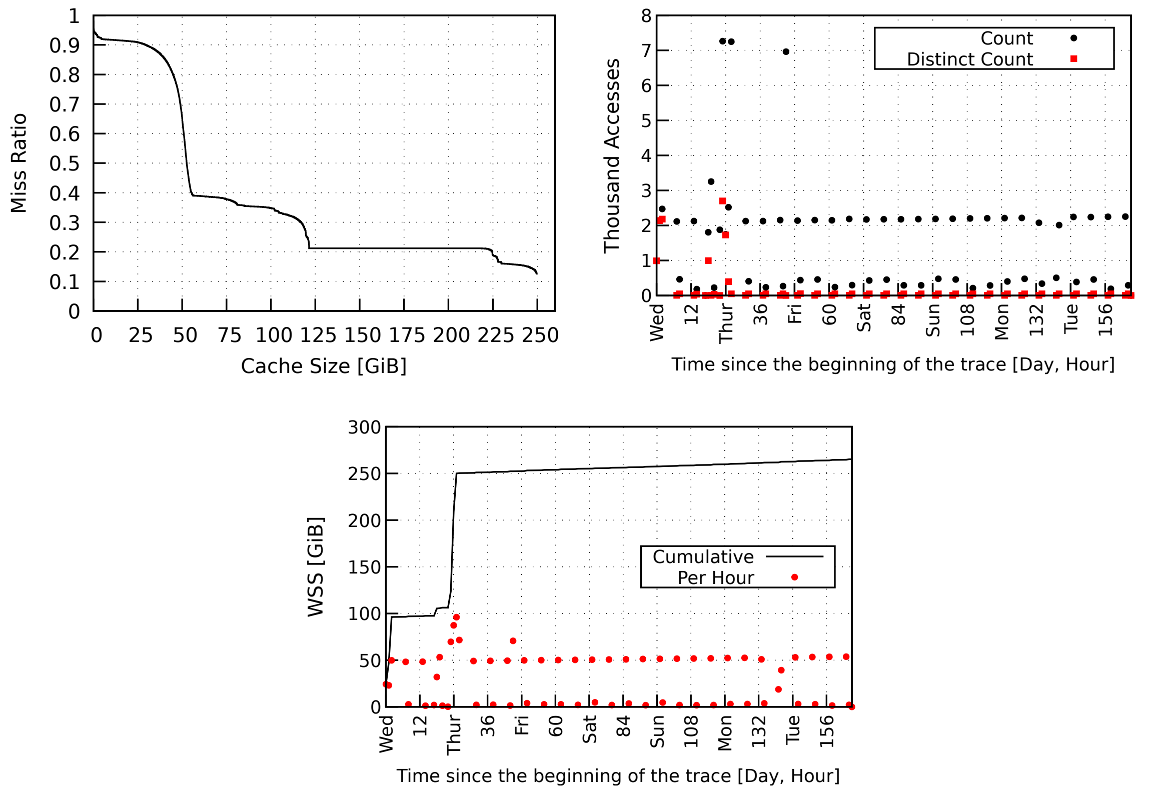


Figure 5.1: IBM workload #079. **Top Left**: MRC for the week. **Top Right**: Hourly access count statistics. **Bottom**: WSS.

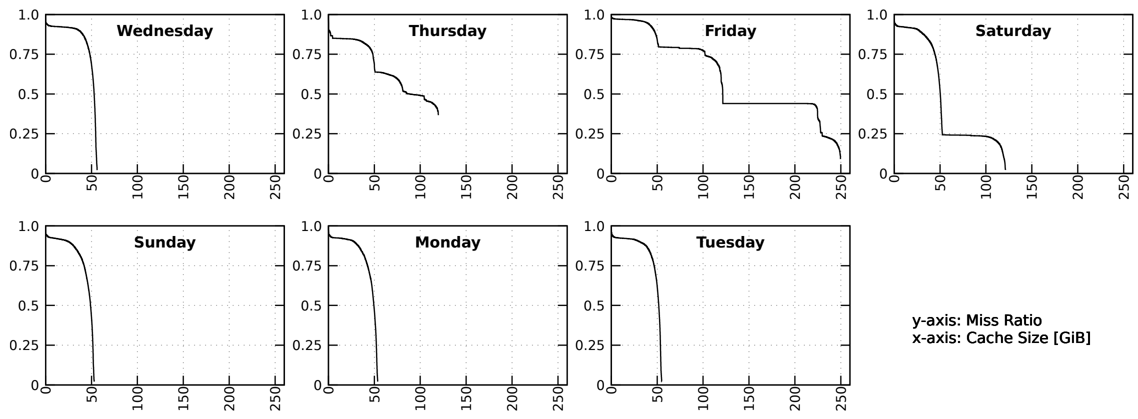


Figure 5.2: IBM workload #079: per-day MRCs. The curves end when increasing the cache size no longer reduces the miss ratio.

5.3 HistoChron Design and Implementation

This section describes the essential background needed to understand HistoChron (§5.3.1), HistoChron's conceptual design (§5.3.2), and its practical implementation (§5.3.3).

5.3.1 Background

Stack Distance Calculation. Recall from the Background Section 2.2.1, Mattson's algorithm records the stack distance of each cache access in a histogram, which is then utilized to generate the MRC. This approach is used by virtually all one-pass MRC-generation algorithms. Following we describe how the stack distance histogram is used to generate the MRC.

Mattson's histogram of stack distances, H, is shown in Fig. 5.3. After processing N accesses, the sum of all histogram frequencies must be equal to N, as shown in Eq. 5.1. Given that every access invariably results in a hit or a miss, and each miss increments the ∞ bin by 1 while a hit increments a non-∞ bin by 1, it follows that the sum of these frequencies must be exactly equal to N. That is,

$$N = \sum_{i=1}^{\infty} H[i] \quad . \tag{5.1}$$

The *Hit Ratio Curve* (HRC) defines the relationship between cache size and its associated hit ratio. Each histogram bin (excluding ∞) corresponds to a cache size on the HRC. For simplicity, assuming the cached objects have a Uniform Object Size, UOS, (e.g., 4KiB), then each bin in the histogram corresponds to a cache size $s = UOS \times$ stack distance represented by that bin.

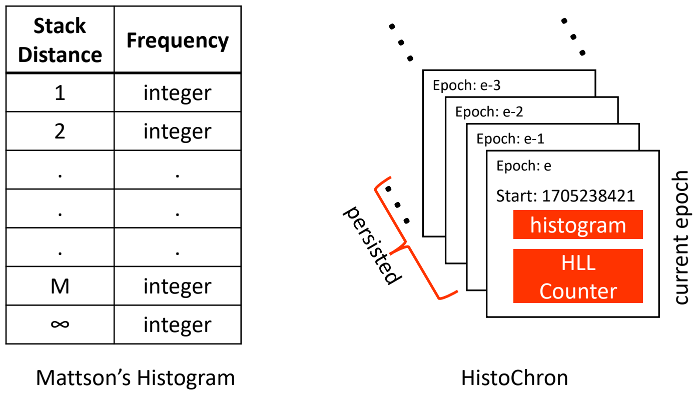
Thus, the histogram supports generating an HRC with cache sizes that range from the minimum stack distance, $min(sd)$, to the maximum stack distance, $max(sd)$, recorded in the histogram. Mattson et al. showed that the hit ratio at cache size s is equivalent to the cumulative sum of frequencies from that bin and all smaller bins, divided by the total number of accesses N; see Eq. 5.2. Considering all stack distances in the histogram (without ∞), effectively the HRC is the histogram's CDF.

$$HRC(s) = \frac{1}{N} \cdot \sum_{i=min(sd)}^{s} H[i] \tag{5.2}$$

The MRC is the inverse of the HRC, which is effectively the inverse-CDF of the histogram: $MRC(s) = 1.0 - HRC(s)$.

Mattson's algorithm cannot generate the MRC over arbitrary time points because the histogram does not record timing information. As a result, it can only generate the MRC from the time of the first access up to the time t_y when the MRC is requested: $MRC(t_1, t_y)$. Once the algorithm proceeds to t_{y+1}, it can no longer generate $MRC(t_1, t_y)$, but is limited to generating $MRC(t_1, t_{y+1})$. In Section 5.3.2, we discuss how we address this gap with HistoChron.

HyperLogLog (HLL). Background on the HLL was provided in Section 2.4. Recall that the HLL operations include `Insert` (adding elements), `Count` (estimating count), and `Merge` (combining multisets). `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 HLLs powerful for interval-based analysis.



Stack Distance	Frequency
1	integer
2	integer
.	.
.	.
.	.
M	integer
∞	integer

Figure 5.3: Mattson's histogram and HistoChron.

5.3.2 Conceptual Design of HistoChron

We divide time into a continuous sequence of uniform, non-overlapping intervals, referred to as *epochs*. Our goal is to have a stack distance histogram and HLL counter per epoch, initialized when the epoch starts. On each access to an object in the cache, HistoChron determines whether a new epoch begins based on the time of the access. If so, it retires the previous histogram and HLL, persisting them lazily for historical analysis. (See Fig. 5.3).

Throughout the lifetime of the cache or while processing an access trace, the data structure of the underlying MRC-generation algorithm is maintained and updated (e.g., Olken's tree of accessed objects, ordered by access recency). On each access to an object in the cache, HistoChron

- determines the stack distance of the object and updates Olken's tree,
- records the stack distance in the current epoch's histogram, and
- inserts the hash of the accessed object's key into the HLL of the current epoch.

Utilizing the information from HistoChron, we describe how to query HistoChron to report the interval-based access statistics, and generate the WSS, and MRC. For simplicity, this explanation covers a widely used scenario in the literature [55, 64, 113, 120] where objects are of uniform size; we show in Section 5.6 how to support heterogeneous sizes.

A. Count(e_x, e_y) The number of accesses over a time-interval, $N(e_x, e_y)$, is obtained by iterating through all stack distance histograms in HistoChron where $e \in [e_x, e_y)$, and for each stack distance bin, accumulate their corresponding frequencies:

$$N(e_x, e_y) = \sum_{\forall e \in [e_x - e_y)} \sum_{sd=min(sd)}^{\infty} \text{HistoChron}[e].H[sd] \quad . \tag{5.3}$$

B. Newly Accessed Objects NAO(e_x, e_y) The number of newly accessed objects in epoch e is equal to the frequency of e's ∞ bin. Thus, the number of newly accessed objects over an interval, $NAO(e_x, e_y)$, is obtained by iterating over the stack distance histograms in HistoChron where $e \in [e_x, e_y)$, and accumulating the frequency of the ∞ bin:

$$NAO(e_x, e_y) = \sum_{\forall e \in [e_x - e_y)} \text{HistoChron}[e].H[\infty] \quad . \tag{5.4}$$

C. Distinct-Count(e_x, e_y) The number of distinct objects accessed in an epoch is captured by e's HLL counter. The number of distinct objects accessed in an interval, $M(e_x, e_y)$, can be calculated by `Merging` the HLLs of the relevant epochs and then performing the `Count` operation on the HLL that results from the `Merge`():

$$M(e_x, e_y) = \texttt{Count}\big(\texttt{Merge}(\text{HistoChron}[e_x].HLL, \cdots, \text{HistoChron}[e_{y-1}].HLL)\big) \quad . \tag{5.5}$$

D. WSS(e_x, e_y) Computing the WSS is straightforward once we know the number of distinct objects accessed in an interval:

$$WSS(e_x, e_y) = UOS \cdot M(e_x, e_y)[Eq. 5.5] \quad . \tag{5.6}$$

where UOS is the uniform object size.

E. MRC(e_x, e_y) For simplicity, we first derive the HRC and then describe how to generate the MRC. Recall from Eq. 5.2 that the hit ratio at cache size s, HRC(s), is equal to the number of hits for cache sizes $\leq s$ divided by the total number of accesses, N. Adjustments to Eq. 5.2 are needed to generate the HRC for cache size s over the interval $[e_x, e_y)$, denoted HRC(s, e_x, e_y). The hit ratio at cache size s over the interval $[e_x, e_y)$ is the number of cache hits in the interval divided by the total number of accesses in the interval. Thus, to obtain the number of hits for cache size s in the interval, we sum the frequencies of all bins for cache sizes $\leq s$, as shown in Eq. 5.7. HRC(s, e_x, e_y) is then obtained using Eq. 5.8.

$$N_{hits}(s, e_x, e_y) = \sum_{\forall e \in [e_x - e_y)} \sum_{\forall sd \leq s} \text{HistoChron}[e].H[sd] \tag{5.7}$$

$$HRC(s, e_x, e_y) = \frac{N_{hits}(s, e_x, e_y)[Eq.\ 5.7]}{N(e_x, e_y)[Eq.\ 5.3]} \tag{5.8}$$

From this, the MRC can be directly derived as $1.0 - HRC(s, e_x, e_y)$.

5.3.3 Practical HistoChron

Following our conceptual design of HistoChron, we now delve into the practical aspects of its implementation. In this section, we describe the data structures used as well as various optimizations to conserve storage space.

In-Memory Design

HistoChron is compatible with any exact stack distance calculation algorithm such as Mattson and Olken. We describe integrating HistoChron with Olken because it is more efficient than Mattson.

For each epoch, HistoChron employs a stack distance histogram and an HLL counter. The histogram is implemented as an array, indexed by stack distance, with values representing frequencies.[1] Upon each access, if the timestamp of the access no longer belongs to the current epoch, we continuously archive the current epoch's timestamp alongside its histogram and HLL, advance the epoch marker, and instantiate a new histogram and HLL. The access is then processed as per Olken's algorithm, and the stack distance of the accessed object is recorded in the (now new) current epoch's histogram. Lastly, the hash of the accessed object's key is `Insert`ed into the epoch's HLL.

Storage Space Considerations

HistoChron's storage space usage depends on:

1. the number of bins in the histogram,
2. the precision of the HLL,
3. the epoch granularity, and
4. how this information is stored.

A. Histogram Bin Granularity The bin granularity primarily affects the details visible in the generated MRC but also affects the amount of space the histogram consumes. Clearly, having each histogram bin represent a byte is impractical, considering that a workload with a WSS of 1GiB would potentially require 4GiB space (assuming each histogram bin stores a `uint32`). Fortunately, that level of detail in the MRC is not needed. In practice, researchers have adopted coarser bin granularities. SHARDS, for instance, uses bins that each represents 64MiB [55]. AET uses logarithmic scaling, where the size of each bin is twice as large as the previous bin [113]. Miniature Simulations uses 100 simulated cache sizes, so the size of each simulated cache is equal to the maximum cache size being simulated divided by 100 [88].

HistoChron introduces a ***NonUniform*** granularity approach that lies between the ***Uniform*** granularity of SHARDS and the logarithmic scaling used by AET. The size of each bin depends on the cache size represented by the bin, where

$$Size(c) = \begin{cases} 32\,\text{MiB} & \text{if } c \leq 1\,\text{GiB} \\ 1\ \text{GiB} & \text{if } 1\,\text{GiB} < c \leq 100\,\text{GiB} \\ 5\ \text{GiB} & \text{if } 100\,\text{GiB} < c \leq 2\,\text{TiB} \end{cases} \tag{5.9}$$

This NonUniform granularity represents cache sizes of up to 2TiB using 520 bins, yet provides sufficient detail for workloads with small WSS. As we will show in Section 5.7, the Uniform implementation consumes 2.8KiB storage space per histogram on average, in practice, while the NonUniform implementation reduces this to 403 bytes on average for a per-minute epoch granularity.

[1] This could also be implemented as a hash table indexed by the stack distance and the value is the frequency of that stack distance.

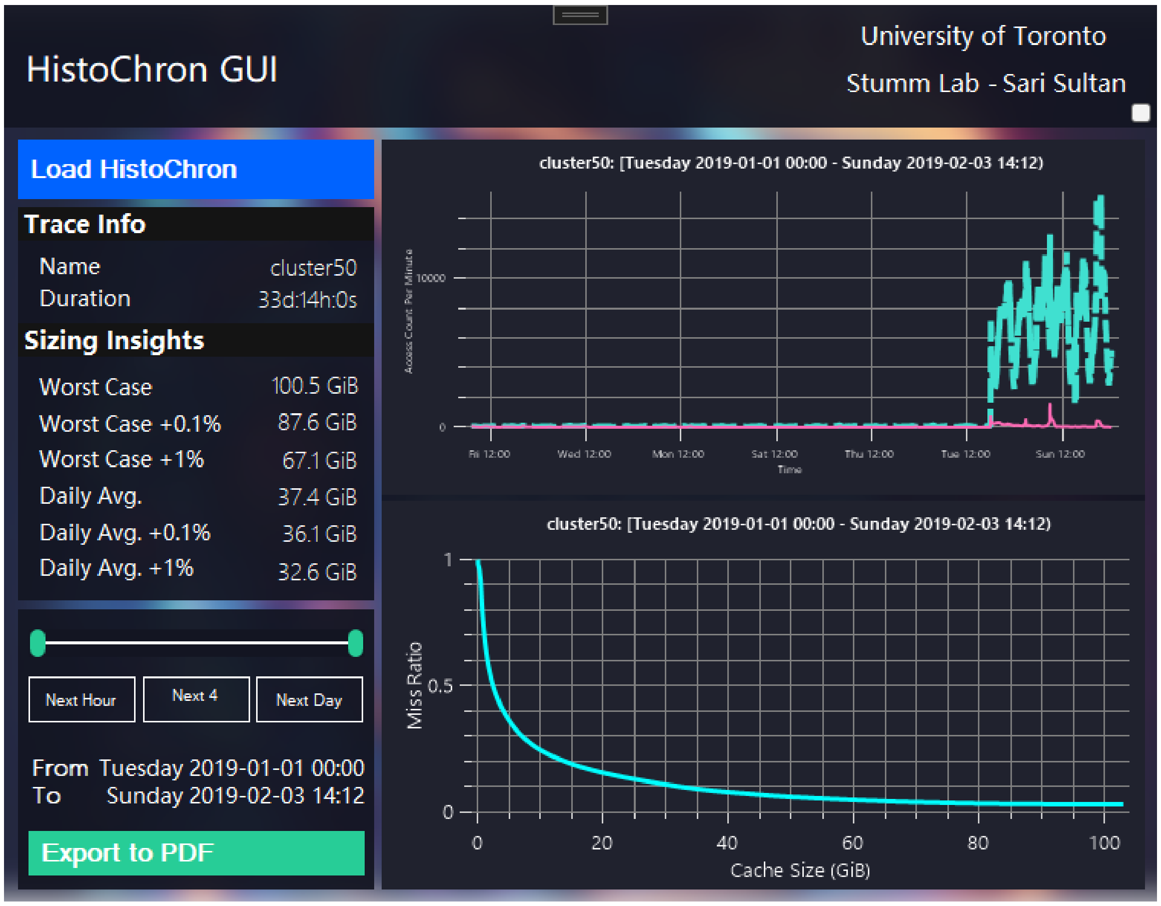
For a per-hour epoch granularity, the Uniform implementation uses 8.4KiB per histogram on average, while the NonUniform consumes 482 bytes on average. The difference in space consumption might seem counterintuitive, as one might think the space usage per histogram does not depend on the epoch granularity. However, we only output non-zero entries, as we describe further below, and the population of the histogram becomes more dense with larger epoch granularities. As such, although the histogram for larger epoch granularity consumes more storage space, the number of recorded histograms is lower, leading to an overall smaller storage space requirement, as we will see below.

B. Precision of the HLL The precision used for the HLL directly affects the storage space consumed per epoch as we output an HLL per epoch. Recall from Section 2.4, each HLL contains an array of buckets, and each bucket can be represented using a byte. The number of buckets is controlled by the precision parameter b, such that the number of buckets equals 2^b. Thus, for an HLL precision of 12, which yields over 98% accuracy in practice, the HLL consumes 4KiB of space.

C. Epoch Granularity Epoch granularity determines the number of stack distance histograms and HLL counters that are generated and need to be stored. The smallest practical granularity is 1 second because timestamps in most publicly available access traces are recorded in units of 1 second. However, 1 second granularity HistoChron information (i.e., histograms and HLLs) would consume over 700MiB of space each week per workload. Moreover, it is questionable whether 1 second epochs provide any additional useful information over, say, 1 minute epochs. Using 1 minute epochs is more meaningful and requires, on average, 48MiB of space per week for the Uniform implementation and 24MiB for the NonUniform implementation (as we show in §5.7). Per-hour epochs require, on average, 2MiB of space per week for the Uniform and 691KiB for the NonUniform implementation. HistoChron uses 1 minute epoch granularity by default, in part because it allows identifying the start of, say, a spike in the number of distinct objects accessed down to the minute.

D. Marshaling and Querying We marshal HistoChron's epoch information into a binary format as follows. We first record the start time of the epoch (`uint32`). We then count the number of non-zero histogram bins and record this count (`uint32`). Because the histogram can be sparse where few bins are utilized, marshalling only non-zero bins could save significant space. For each non-zero stack distance bin, we record the ⟨stack distance, frequency⟩ pair (⟨`uint32`,`uint32`⟩). Finally, we record the HLL counter using either a dense or sparse format based on which one uses less space. For that, we first count the number of non-zero buckets in the HLL, and if they are few, we use the sparse implementation where we record the number of non-zero buckets, followed by a sequence of ⟨bucket index, value⟩ pairs (⟨`uint16`,`byte`⟩). If the number of non-zero buckets is large, then we marshal the dense format, which is a sequence of bytes, one per HLL bucket. A flag (`byte`) is recorded to identify which format is used.

It is straightforward to unmarshal this format into an array structure, which is efficient for querying. For instance, to answer the query $N(e_x, e_y)$ (Eq. 5.3), we first use *binary search* to determine the start and end indices in the unmarshaled array because the timestamps are sorted and then iterate over the entries in the range to compute the sum. This method is equally effective for all queries.



Trace Info

Name	cluster50
Duration	33d:14h:0s

Sizing Insights

Worst Case	100.5 GiB
Worst Case +0.1%	87.6 GiB
Worst Case +1%	67.1 GiB
Daily Avg.	37.4 GiB
Daily Avg. +0.1%	36.1 GiB
Daily Avg. +1%	32.6 GiB

From Tuesday 2019-01-01 00:00
To Sunday 2019-02-03 14:12

Figure 5.4: A screenshot of the HistoChron GUI system initialized with the HistoChron of Twitter's recommended workload #`50`.

5.4 HistoChron GUI

HistoChron's GUI is a tool that uses HistoChron to enable interval-based analysis of caching workloads. It can assist system administrators in comprehensively understanding caching requirements and conducting in-depth performance debugging. Fig. 5.4 shows a screenshot of HistoChron GUI.

Operation. Upon initialization, a workload's HistoChron can be loaded into the GUI via the `Load HistoChron` button. For instance, consider the HistoChron from Twitter's recommended workload #`50`,[2] as illustrated in Fig. 5.4. The system displays the trace information, including the name and duration, and it generates two main graphs: one depicts the access count statistics, which includes the access count and distinct access count per epoch, while the other shows the MRC.

The user can zoom into specific time intervals using an intuitive slider (on the bottom left), thereby adjusting the graphs and enabling the user to swiftly focus on periods of interest. For instance, a preliminary observation of the Twitter workload over a 33-day span reveals low activity during the first three weeks (roughly 100 accesses per minute).

[2]Recommended to users on the discussion channel [110].

Cache Sizing Insights. Determining an optimal cache size often depends on the system administrator's budget considerations. We observed that the MRCs of many 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 consider a miss rate increase of $\delta = 0.1\%$ or $\delta = 1\%$ to be acceptable in return for some memory savings. Hence, we show such cache sizing insights (see Fig. 5.4 left panel). The figure shows that the cache size required to achieve the minimal miss rate is 100.5GiB. If the operator is willing to accept an increase of up to 1% in miss rate, then the cache size required becomes 67.1%GiB, saving 33% of memory.

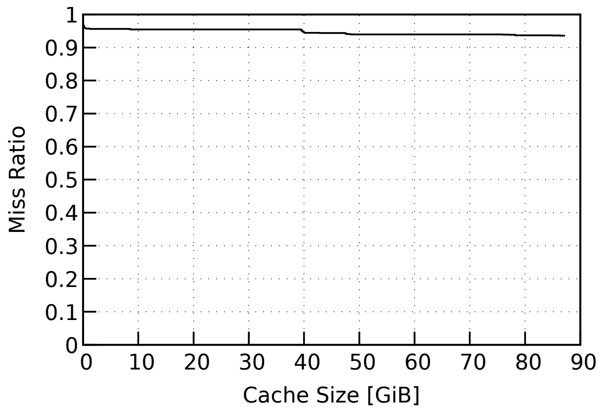
Other metrics (e.g., daily standard deviation) can easily be incorporated into the GUI, and as an example we included the per-day cache requirement analysis which could help identify whether the cache requirements are stable or differs significantly for different periods. Day-to-day analysis shown in the figure reveals that average daily cache size requirement is significantly lower, at 37.4GiB (compared to 100.5GiB for the 33 day period). This shows a significant variability in the workload's caching requirements.

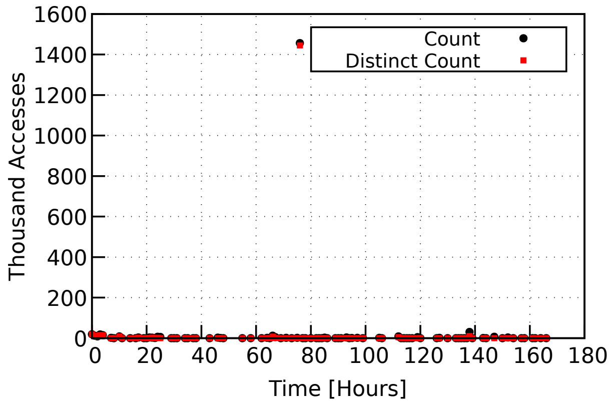
Performance Debugging. Operators might observe a sudden increase in back-end data store queries with a concomitant increase in cache miss ratios. The GUI provides valuable insights to understand such anomalies. For example, after analyzing the behavior of the MSR `mds` workload, we observed a significant increase in the access rate within a single hour of the week, along with a spike in the number of unique objects accessed, suggesting an extensive scanning operation: 1.45M accesses to 1.44M unique objects in a single hour (Fig. 5.5-middle). This significantly altered the resultant MRC.

Insights such as these can be helpful for optimizing cache utilization and for identifying potential remediations, e.g., temporarily suspending the caching of new objects during intensive scanning operations, switching to a different eviction policy, or making the application use a separate cache. This is graphically demonstrated in Fig. 5.5, showcasing the stark contrast in MRC with and without the problematic one hour that included the scanning pattern. The MRC over the entire period indicates that this workload is not a good candidate for caching as evidenced by a miss rate of over 90%. However, the MRC, excluding that single-hour scan, shows that this workload can significantly benefit from caching where the miss rate is as low as 25%.

5.5 HistoChron with Shards

The last decade witnessed several advancements in *approximate* stack distance calculation, enabling online MRC-generation. In 2015, a breakthrough was introduced with Shards, where the space complexity was reduced from $\mathcal{O}(M)$ to a constant $\mathcal{O}(1)$ [55] (for fixed-sized Shards). Background on Shards was presented in Section 2.2.4. In this section, we describe how to incorporate FR-Shards (§5.5.1) and FS-Shards (§5.5.2) with HistoChron.





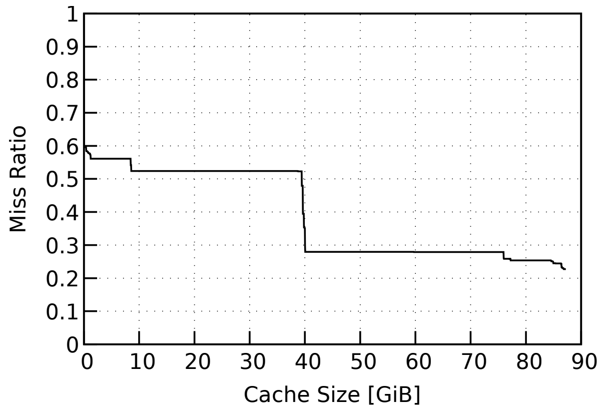


Figure 5.5: MSR `mds` workload. **Top**: MRC for the entire week. **Middle**: Hourly access rate revealing a significant anomaly within one hour of the week. **Bottom**: MRC for the duration post-anomaly until the end of the week.

5.5.1 FR-Shards

FR-Shards uses a fixed sampling rate, R, when processing a trace using Olken but scales each stack distance by R because each sampled access now represents $1/R$ accesses [55]. Due to sampling, the number of accesses in HistoChron's histogram only represents the sampled accesses. To support FR-Shards$_{adj}$ in tandem with HistoChron, we also record the number of unsampled accesses for each epoch, adding `uint32` overhead per epoch entry. FR-Shards$_{adj}$ can be applied to interval-based MRC-generation as follows. First, we calculate the total number of accesses within an interval. Second, we calculate the expected number of sampled accesses by multiplying the total by the sampling rate R. Third, we combine the stack distance histograms within the interval and calculate the number of sampled objects from the recorded frequencies. Lastly, we adjust the first bin of the combined histogram by the difference between the expected number of sampled objects and the actual number of sampled accesses. (Note that accesses `Insert`ed into HistoChron's HLLs are not sampled.)

5.5.2 FS-Shards

FS-Shards samples accesses to objects such that the number of distinct sampled objects does not exceed a constant, S_{max}. To incorporate FS-Shards with HistoChron, we introduce a new variable to track the sampling threshold per epoch, T_e. This is necessary because generating the $MRC(e_x, e_y)$ requires knowing the sampling rate at e_{y-1} in order to do the stack distance scaling: when generating the MRC, each of the recorded frequencies in the HistoChron is scaled by $T_{bin}/T_{e_{y-1}}$. To support FS-Shards$_{adj}$, we use the same approach as described for FR-Shards, but instead of using a fixed sampling rate, R, as in FR-Shards, we use the last epoch's sampling rate, $R_{y-1} = T_{y-1}/P$.

5.6 Non-Stack-Based Policies and Heterogeneous Sizes

Interval-Based Analysis for Non-Stack Policies Mattson, Olken, Shards, CounterStacks, and virtually all other one-pass algorithms only support stack policies (e.g., LRU, OPT) [61, 88]. For non-stack policies (e.g., FIFO, ARC [150], 2Q [181], LIRS [194], LHD [196], S3-FIFO [184]), the state-of-the-art exact MRC-generation algorithm is running a simulation for each cache size on the MRC, and the state-of-the-art in approximate MRC-generation is Miniature Simulations (MiniSim) [88].[3] We describe here how simulations and MiniSim can be extended to support interval-based analysis.

Assume we are interested in generating an MRC with cache sizes at granularity of s bytes, up to a maximum cache size X, so the sizes to be simulated are $\{s, 2s, 3s, \cdots, X-s, X\}$. For each size, we run a separate simulation. For each epoch, we track the number of accesses, N_e, which is the same for all simulations. At the end of each epoch, we record the number of misses for each of the simulated sizes, $K[e][s]$. This information enables generating the interval-based MRC:

$$MRC(s, e_x, e_y) = \frac{\sum_{e=e_x}^{e_{y-1}} K[e][s]}{\sum_{e=e_x}^{e_{y-1}} N[e]} \quad . \tag{5.10}$$

[3]Kosmo also models non-stack-based eviction policies like MiniSim, and both use Shards [124]. But since Kosmo also produces histograms similar to Mattson, it can be adapted to work with HistoChron by generating a histogram per epoch, then use the same approach we use with HistoChron (§5.3).

MiniSim [88] uses SHARDS sampling on top of the simulation approach, which simulates each cache size on the MRC, but instead of running full simulations, SHARDS is used to run a sampled subset of the cache accesses. To support interval-based analysis with MiniSim, we use the exact same approach described above but scale the recorded statistics by the fixed sampling rate R, as described by Waldspurger et al. [88]. In Section 5.7, we show that our implementation incurs minimal overhead beyond that of the original MiniSim algorithm.

Because HistoChron records a single HLL per epoch, we can identify the distinct number of objects accessed in an interval (see Eq. 5.5) and the interval WSS (see Eq. 5.6). The number of newly accessed objects for a specific epoch, $NAO(e_x)$, can be found as the difference between the count of the merged HLLs from the first epoch up to e_x inclusive and the count of the merged HLLs from the first epoch up to e_{x-1} inclusive. The same concept can be applied to find the NAO in an interval.

Heterogeneous Object Sizes Support To accommodate heterogeneous object sizes in HistoChron, which is based on Olken (§5.3), the weight of each node in Olken's tree is redefined to be equal to the aggregate size of its child nodes, as described in Section 4.1.2. In addition, we introduce a new variable for each epoch that is tracked by HistoChron. It records the aggregate size of distinct objects per time epoch, which allows computing the interval-based aggregate size of the NAO with heterogeneous sizes. For the WSS, we can track the average object size per epoch, so the WSS is calculated based on the average object size instead of a uniform size. As noted earlier, this can be computed efficiently using Welford's running average method, which requires two variables only: a `double` for the average and an `integer` for the count [279]. However, we observed if there is significant variability in the size of objects, the WSS can become inaccurate. To address this, we can introduce multiple HLLs, one for each object size group (grouped as powers of 2), and track the average object size per HLL to reduce the variance; this has shown to be within 2% of the exact WSS when using HLL $b = 12$.

5.7 Evaluation

In this section, we show that:

1. HistoChron generates exact MRCs,
2. HistoChron has minimal computational overhead over Olken,
3. our interval-based MiniSim has minimal computational overhead over MiniSim while maintaining the same accuracy,
4. HistoChron with SHARDS demonstrates high throughput across various configurations,
5. HistoChron with SHARDS demonstrates high accuracy across various configurations,
6. HistoChron needs just 24MiB weekly for 1 minute epochs and 691KiB for 1 hour epochs,
7. HistoChron significantly reduces the storage footprint by 99% compared to access traces, efficiently offering detailed cache analysis without storing access traces.

5.7.1 Experimental Setup

The experimental setup is identical to that of the previous two chapters (see Section 3.6).

We evaluated HistoChron on 5,808 cache access traces from six real-world datasets encompassing more than 300 billion accesses and, when combined, span a total duration of 18 years (Table 3.1). For the cloud virtual volumes (CVVs) access traces from Tencent, the original study mentioned they were mapped to 40 cache instances, but the mapping was not provided [56]; thus, we employed modulo-based assignment to create 40 workloads (i.e., CVV-id%40). For MSR's workloads, we included a combined workload of all 13 traces, as done in prior studies [55, 64, 113]. We used GET/READ accesses from all workloads, and used a uniform object size of 4KiB to be able to compare with related work, which used the same configuration and only supported uniform object sizes [64]. The *Mean Absolute Error* (MAE) was used to quantify the accuracy of the generated MRCs by our algorithms, which is widely used in the literature [55, 64, 69, 84, 88, 90, 106, 113, 143]. The aforementioned configurations result in 264 different workloads. We use HLL precision $b = 12$ by default.

For Figures 5.6, 5.7, 5.8, and 5.9, each candlestick represents the range of results described therein: The line's top and bottom represent maximum and minimum results. The box's top and bottom represent the 75th and 25th percentiles. SHC refers to SHARDS with HistoChron. U refers to the Uniform, and NU to the NonUniform histogram implementation (§5.3.3).

5.7.2 Results and Discussion

(1) HistoChron generates exact MRCs. We evaluated HistoChron's accuracy by generating exact MRCs for the MSR workloads using Olken's algorithm for different randomly selected 1,000 time intervals for each workload at 1 minute granularity (using the default Random function generator seeded with the value 2023). HistoChron was then used to generate corresponding MRCs for the same periods. Direct comparison of the two sets of MRCs yielded an MAE of 0, demonstrating HistoChron's exactness.

(2) HistoChron incurs minimal computational overhead over Olken. Computational overhead of HistoChron was evaluated by comparing its throughput to that of Olken across all MSR workloads, using 1 minute epoch granularity, with each experiment repeated 10 times. On average, HistoChron exhibited a slowdown of 4.5%, with the 25^{th} and 75^{th} percentiles at 2.2% and 4.6%.

(3) Our interval-based MiniSim incurs minimal computational overhead compared to MiniSim while having the same accuracy. The computational overhead of our interval-based MiniSim (§5.6) was evaluated by comparing its throughput to that of MiniSim across all MSR workloads, with each experiment repeated 10 times, using a sampling rate $R = 0.1$, 1 minute granularity, and using the `FIFO` policy. On average, our interval-based MiniSim exhibited a slowdown of 8%, with the 25^{th} and 75^{th} percentiles at 0.8% and 10.87%, respectively. Similarly, we verified our implementation's accuracy by computing the MAE between the interval-based MiniSim and the original MiniSim, similar to the way we evaluated HistoChron, and the resulting MRCs were exactly the same, with an MAE of 0. By comparing directly to MiniSim, which has been shown to be an accurate approximation by Waldspurger et al. [88], we demonstrate that our extension maintains the original algorithm's accuracy while adapting it to support interval-based analysis.

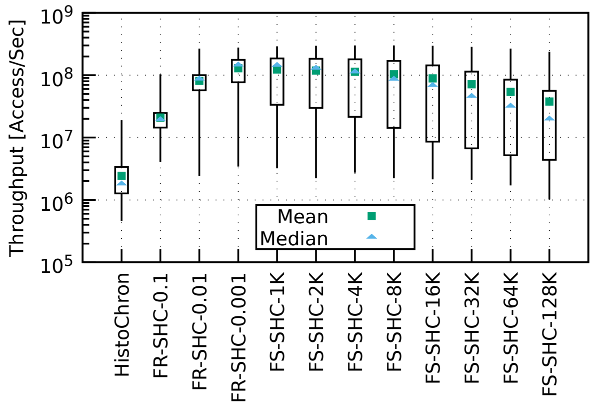


Figure 5.6: Throughput of HistoChron for all workloads. HistoChron is the exact variant. FR-SHC is the FR-SHARDS variant adapted to work with HistoChron, followed by the fixed sampling rate used. FS-SHC is the FS-SHARDS variant adapted to work with HistoChron, followed by the S_{max} parameter.

Table 5.1: Throughput for different S_{max} values for HistoChron with FS-SHARDS

S_{max}	Throughput (M)
1K	123.5
2K	119
4K	113.1
8K	103.2
16K	88.9
32K	71.7
64K	54.1
128K	37.8

(4) HistoChron with SHARDS demonstrates high throughput across various configurations. We measured the throughput of HistoChron in terms of the number of accesses per second, excluding the IO time required to read traces from disk and parsing them, thus allowing a more direct comparison across a wide range of configuration parameters. Fig. 5.6 shows that HistoChron has an average throughput of $2.4M$. The throughput of HistoChron with FR-SHARDS is, on average, $20.9M, 81.3M$, and $129M$ for sampling rates $R = \{0.1, 0.01, 0.001\}$, respectively. Similarly, as shown in Table 5.1, the throughput varies with different S_{max} values for HistoChron with FS-SHARDS and ranges between 123M and 37M for S_{max} equal to 1K and 128K, respectively.

(5) HistoChron with SHARDS demonstrates high accuracy across various configurations. Evaluating the accuracy of MRC-generation is more challenging because there are many different time-interval ranges for each workload. Thus, for each workload considered, we generated 1,000 randomly selected time ranges within the duration of the workload using the default C# Random function generator seeded with the value 2023. The exact MRC for each period was generated using our HistoChron algorithm, which we evaluated earlier and showed that it is exact. Fig. 5.7 shows the MAE for the tested algorithms, where each candlestick represents the MAEs over 264,000 MRCs. For FR-SHARDS and FS-SHARDS, we show the results with and without SHARDS$_{adj}$. SHARDS$_{adj}$ should be used as a best practice as it significantly reduces the MAE. For example, for HistoChron with FS-SHARDS using $S_{max} = 8K$, SHARDS$_{adj}$ reduced the MAE from 3.89% to 0.46%, on average.

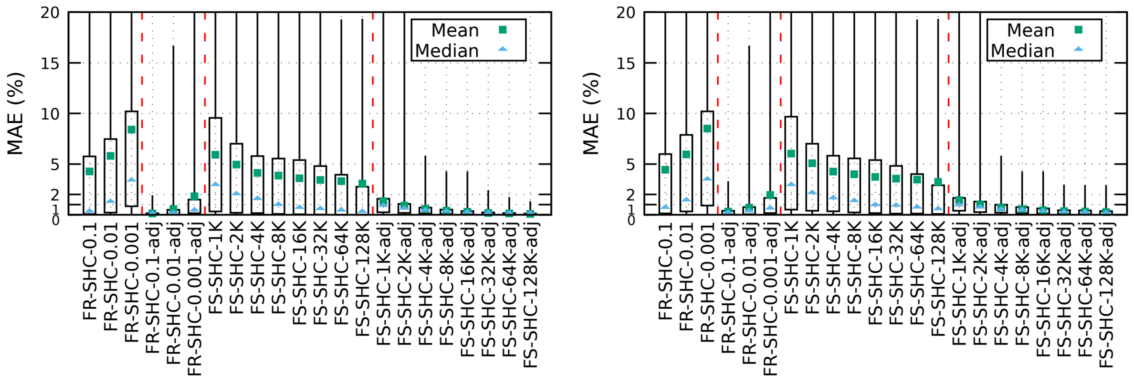


Figure 5.7: HistoChron's accuracy in terms of MAE. **(Left)** 1 minute epochs **(Right)** 1 hour epochs. FR-SHC is the FR-SHARDS variant adapted to work with HistoChron, followed by the fixed sampling rate used. FS-SHC is the FS-SHARDS variant adapted to work with HistoChron, followed by the S_{max} parameter.

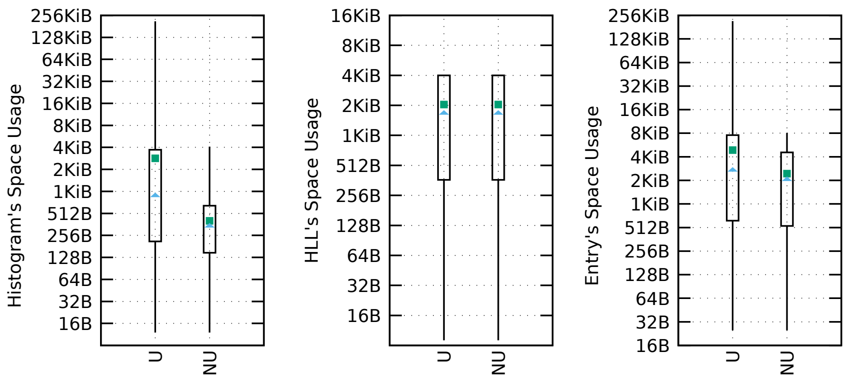


Figure 5.8: HistoChron's space requirement using 1 **minute** epochs **(Left)** for the histograms, **(Center)** HLLs, and **(Right)** combined.

For the 1 minute epoch granularity, the MAE of HistoChron with FS-SHARDS$_{adj}$ range from 0.10% to 1.44%. Conversely, for the 1 hour epoch granularity, the MAEs range from 0.24% to 1.35%. These values are detailed for each S_{max} setting in Table 5.2. We posit that employing $S_{max} = 8K$ is judicious, as it achieves a high throughput exceeding $100M$ accesses per second, with a concomitant MAE of ˜0.5% across various configurations, on average.

(6) HistoChron needs just 24MiB weekly for 1 minute epochs and 691.1KiB for 1 hour epochs. Recall from §5.3, HistoChron epochs comprise a histogram and an HLL. We analyzed their space requirements. Fig. 5.8 details 1 minute epoch statistics from our tested workloads that contain 9 million distinct minutes, showing average histogram sizes of 2.8KiB (U) and 403 bytes (NU), and a consistent HLL size of 2KiB, totaling an average of 4.87KiB (U) and 2.44KiB (NU) consumed per epoch. For each 1 hour epoch (Fig. 5.9), the histogram uses 8.47KiB (U) and 482 bytes (NU), and the HLL 3.6KiB, for a total average of 12.11KiB (U) and 4.11KiB (NU) per epoch. Overall space requirements are larger for 1 hour epochs than for 1 minute epochs due to increased density in both the histogram and HLL over longer durations (§5.3.3). Table 5.3 shows the total space requirements to store HistoChron's information for different durations. For example, the space consumed weekly using the NU implementation is 24MiB with 1 minute epochs and 691KiB with 1 hour epochs.

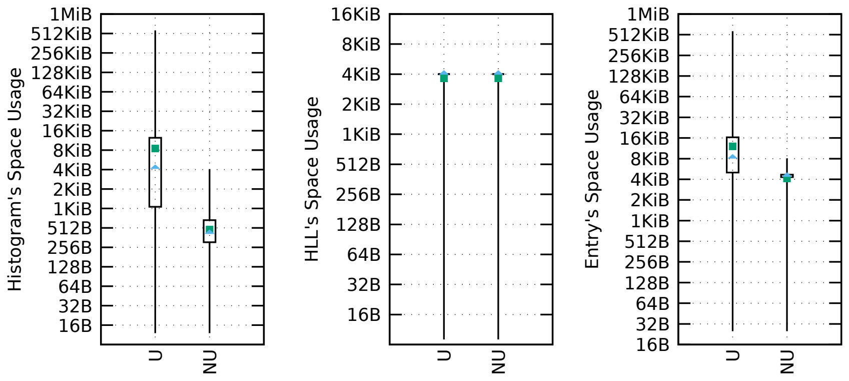


Figure 5.9: HistoChron's space requirement using 1 **hour** epochs **(Left)** for the histograms, **(Center)** HLLs, and **(Right)** combined.

Table 5.2: MAEs for HistoChron with FS-SHARDS$_{adj}$ at different S_{max} values for 1 minute and 1 hour epoch granularities.

	MAE for	
S_{max}	1 minute epoch	1 hour epoch
1K	1.44%	1.35%
2K	0.98%	0.95%
4K	0.65%	0.67%
8K	0.46%	0.51%
16K	0.32%	0.40%
32K	0.21%	0.32%
64K	0.13%	0.27%
128K	0.10%	0.24%

Table 5.3: HistoChron's storage requirements for both the Uniform and NonUniform stack distance histogram configurations (using the averages per entry from Fig. 5.8 and Fig. 5.9).

Granularity	1 minute		1 hour	
Duration	Uniform	NonUniform	Uniform	NonUniform
Minute	4.9KiB	2.4KiB	-	-
Hour	292KiB	146.4KiB	12.1KiB	4.1KiB
Day	4.8MiB	3.4MiB	290KiB	98.7KiB
Week	47.9MiB	24MiB	2MiB	691.1KiB
Month	1.4GiB	720MiB	60MiB	20.2MiB
Year	16.8GiB	8.4GiB	715MiB	243MiB
Decade	168.6GiB	84.4GiB	7GiB	2.3GiB

(7) HistoChron significantly reduces storage needs by 99% compared to access traces, efficiently offering detailed cache analysis without storing access traces. We assessed HistoChron's size reduction relative to the original size of traces. We converted access traces from CSV to a compact binary format for a fairer comparison. For example, Twitter's dataset storage space is 18TiB in CSV but only 3.7TiB in binary. We compared the aggregate size of access traces in each dataset to the aggregate size required by HistoChron (Table 5.4). HistoChron achieved an average storage reduction of 99% compared to raw access traces, eliminating the need to store access traces while still providing accurate interval-based access statistics, WSS, and MRCs.

Table 5.4: Aggregate storage requirements per dataset using 1 minute and 1 hour epochs. HistoChron variants. (Note that each dataset contains multiple access traces)

		Granularity			
		1 minute		1 hour	
Dataset	Raw Size	Uniform	NonUniform	Uniform	NonUniform
MSR (1 week)	17.0GiB	154.4MiB	139.3MiB	9.3MiB	5.6MiB
Wikipedia (20 days)	70.2GiB	784.5MiB	150.7MiB	25.6MiB	2.5MiB
IBM (1 week)	6.2GiB	582.6MiB	297.3MiB	32MiB	25.1MiB
SEC (13 years)	382.8GiB	28.1GiB	16.0GiB	1.3GiB	519.5MiB
Tencent (week)	186.8GiB	3.8GiB	2.2GiB	133.3MiB	39.4MiB
Twitter (1-4 weeks)	3.4TiB	8.8GiB	2.2GiB	332.4MiB	43.7MiB
Sum	4.1TiB	42.1GiB	21GiB	1.8GiB	635.8MiB
% reduction over raw		99%	99.5%	99.95%	99.98%

5.8 Related Work

MRC and WSS. Background information on, and prior work related to, MRCs and the WSS were presented in Chapter 2. In summary, MRC-generation advancements over five decades, transitioning from exact algorithms like those by Mattson [61] and Olken [62] to approximate ones such as CounterStacks [64], SHARDS [55], AET [113], and MiniSim [88], along with parallel WSS research [11, 14, 60, 68, 93–106], have been pivotal to optimizing cache resources [9, 56, 63–92, 106, 212].

The Sliding Window Model (SWM). The SWM, introduced in 2002, is a popular method for analyzing data streams by measuring statistics on the most recent Δ period [280]. It processes only the most recent subset of data within a fixed-size window that slides over time, enabling real-time analysis and decision-making based on current information. The key difference between HistoChron and the SWM is that HistoChron is more general than the SWM as it allows interval-based analysis between any two time epochs rather than the last Δ period.

The SWM is widely used in various domains such as resource optimization, network monitoring, security, and anomaly detection due to its relevance in modern data-intensive scenarios [217, 281–285]. As data grows exponentially, the SWM's method of summarizing data streams becomes essential, striking a balance between summary detail and query scope. A classic illustration of SWM's utility is in telecommunications, where companies prioritize recent call records for billing, subsequently archiving older data, which becomes less frequently accessed [280]. Another prominent application is in web analytics, particularly for calculating the number of unique visitors within a specific time frame (known as the *distinct-count* problem [162, 166]). Recent enhancements in this area include adapting popular distinct-count algorithms like HyperLogLog [166] to the SWM context, providing a more granular view of user engagement over time windows [217].

Incorporating the SWM into cache performance analysis is crucial due to the dynamic nature of workloads and the difficulty in storing full traces. Such integration allows for more efficient cache behavior understanding and improved anomaly detection. This necessity stems from the dynamic requirements of modern systems for resource allocation and performance optimization [64].

Table 5.5: Throughput comparison between HistoChron with FR-SHARDS and CounterStacks (CS)

	Speedup Over	
Sampling Rate R	**HiFi CS**	**LoFi CS**
0.1(10%)	23×	2.5×
0.01(1%)	91×	9.8×
0.001(0.1%)	146×	15.6×

CounterStacks. CounterStacks is the most closely related work to HistoChron [64], which is an *approximate* MRC-generation algorithm that checkpoints its internal state periodically, from which the MRC can be generated over arbitrary time intervals. The differences between CounterStacks and HistoChron are: (1) HistoChron generates exact MRCs while CounterStacks does not, (2) HistoChron provides more interval-based access statistics such as the number of accesses to newly accessed objects, the number of accesses to distinct objects, and the WSS, (3) CounterStacks only support the LRU eviction policy, while HistoChron supports all eviction policies through our extension with MiniSim, (4) CounterStacks only supports objects of uniform size while HistoChron support heterogeneous object sizes. HistoChron marks a significant step towards more dynamically adaptive and precise cache performance analysis and optimization.

We compared HistoChron with CounterStacks using the original parameters from [64, 113]: HiFi (pruning $\delta = 0.02$, 1-minute epochs) and LoFi ($\delta = 0.1$, 1-hour epochs), with a downsampling rate of 1 million for both. ***Our finding: HistoChron with SHARDS offers a significant performance improvement over CounterStacks while being more accurate.***

Regarding throughput, Table 5.5 shows a comparison between HistoChron with FR-SHARDS and the two variants of CounterStacks. FR-SHARDS is 23× to 146× faster than the HiFi variant of CounterStacks and 2.5× to 15.6× faster than the LoFi variant of CounterStacks depending on the selected sampling rate. Similarly, Table 5.6 compares the throughput between HistoChron with FS-SHARDS and the two variants of CounterStacks. FS-SHARDS is 80× to 138× faster than the HiFi variant of CounterStacks and 8× to 14× faster the LoFi variant of CounterStacks depending on the selected S_{max} value for FS-SHARDS.

Regarding accuracy, for 1 minute epochs, CounterStacks achieves an MAE of 0.48%, while HistoChron with FS-SHARDS$_{adj}$ achieves MAEs of 0.46% and 0.21% for $S_{max} = 8K$ and $32K$, respectively. This shows that having $S_{max} = 8K$ is needed to achieve accuracy similar to the HiFi variant of CounterStacks and having $S_{max} = 32K$ results in 56% higher accuracy. For 1 hour epochs, CounterStacks achieves an MAE of 2.75%, on average, while HistoChron with FS-SHARDS$_{adj}$ achieves an MAE of 1.35% for $S_{max} = 1K$, which shows 50% higher accuracy than CounterStacks LoFi even when using very small number of objects. A detailed MAE comparison between HistoChron with FS-SHARDS$_{adj}$ and CounterStacks is shown in Table 5.7.

Table 5.6: Throughput comparison between HistoChron with FS-SHARDS and CounterStacks

	Speedup Over	
S_{max}	**HiFi CS**	**LoFi CS**
1K	138.9×	14.9×
2K	133.9×	14.3×
4K	127.2×	13.6×
8K	116.1×	12.4×
16K	100×	10.7×
32K	80.6×	8.6×

Table 5.7: MAE comparison between HistoChron with FS-SHARDS$_{adj}$ and CounterStacks

	MAE for	
	1 minute epochs (HiFi)	**1 hour epochs (LoFi)**
CounterStacks	0.48%	2.75%
FS-SHARDS$_{adj}$with S_{max} =		
1K	1.44%	1.35%
2K	0.98%	0.95%
4K	0.65%	0.67%
8K	0.46%	0.51%
16K	0.32%	0.40%
32K	0.21%	0.32%

5.9 Concluding Remarks

In this chapter, we introduced HistoChron, a novel methodology for interval-based historical analysis of cache workloads. HistoChron enables generating precise interval-based miss ratio curves, working set sizes, and cache access statistics while using just 24MiB weekly for per-minute epochs and 691KiB for per-hour epochs, showcasing its space efficiency. A system with a graphical user interface was presented, which utilizes HistoChron and is designed to assist administrators in better understanding the behavior and anomalies in their caches, in addition to providing cache sizing insights. We also presented a variant of HistoChron that is based on the SHARDS algorithm with reduced overheads, yielding approximate results with less than 1% mean error. Moreover, we adapted Miniature Simulations to work with HistoChron, thus expanding its applicability to all eviction policies.

Chapter 6

Concluding Remarks and Future Research Directions

This dissertation has explored analyzing the performance of in-memory caches, which are essential in improving the performance of modern distributed systems. We demonstrated that existing MRC-generation and WSS estimation algorithms have significant gaps in accommodating modern workload characteristics such as TTL attributes and heterogeneous object sizes. We also showed the usefulness of interval-based analysis for modern workloads given their dynamic nature. In this dissertation, we proposed MRC-generation and WSS estimation algorithms that address these gaps, thus facilitating cache performance analysis for modern workloads.

Major Contributions and Impact

1. TTL Attributes. We showed that the state-of-the-art in cache performance analysis is oblivious to TTL attributes, which are widely used in modern in-memory caches. This oversight has a significant impact on the accuracy of MRC-generation and WSS estimation. By extending MRC-generation and WSS estimation algorithms to incorporate TTL attributes, we showcased potential memory footprint reductions by an average of 69% (and up to 99%), marking a significant stride towards resource optimization and cost efficiency in cache management. Given that prominent organizations provision terabytes to petabytes of DRAM for in-memory caches and given that the costs of the caching layer can be substantial in modern applications, our contributions have a significant potential impact on most organizations that utilize in-memory caches.

2. Heterogeneous Object Sizes. We showed that heterogeneous object sizes can significantly impact the accuracy of MRCs and WSSes. We then showed how the state-of-the-art MRC-generation and WSS estimation algorithms can be extended to support heterogeneous object sizes.

3. Interval-based Historical Analysis of Caching Workloads with HistoChron. We introduced HistoChron, a novel methodology for interval-based historical analysis of cache workloads. HistoChron's ability to generate precise, interval-based MRCs, WSSes, and access statistics while using modest storage requirements empowers administrators to identify access patterns, diagnose issues, and make informed decisions regarding cache sizing and policy adjustments.

In summary, this dissertation presents tools and methodologies that facilitate the performance analysis of modern in-memory caches. Addressing the aforementioned gaps in the state-of-the-art MRC-generation and WSS estimation algorithms paves the way for a better understanding of in-memory cache resource optimization.

Future Research Directions

While this dissertation advances the field of in-memory cache workload analysis and cache configuration, it also exposes new challenges and opportunities for future research. Here, we briefly highlight some potential interesting future research directions.

HLL optimizations. We believe that substantial optimizations to our extended HLL implementations are possible. To support heterogeneous object sizes and TTLs, we needed to dramatically increase the memory footprint of the HLLs, which in turn also reduced the performance of our implementation due to poorer cache locality. We believe that a number of promising strategies can reduce the amount of memory needed. Further, we believe that using sampling (as, e.g., used by SHARDS) could further reduce processing and memory overheads significantly for HLL counters without significantly impacting accuracy. These strategies need to be investigated in more detail to better understand the overhead–accuracy trade-offs.

Furthermore, adapting our extended HLLs to the sliding window model [217] could improve the efficiency of our extended HLLs when incorporated with HistoChron.

Placement of MRC generation and WSS estimation. An interesting question is where to place the code that generates MRCs and estimates WSSes. One obvious option is to place them within the in-memory cache, by extending the cache (e.g., Memcached or Redis) implementations. There are two potential downsides to this approach:

1. Resource usage. Because MRC generation and WSS estimation consume resources, they may negatively impact the performance of the cache itself. For example, MRC generation may consume considerable physical memory, potentially taking away from physical memory that could be assigned to the cache, thus increasing the miss ratio. Or MRC generation's CPU usage may reduce the cache's response times.

2. Fault tolerance. The data generated by MRC generation and WSS estimation will reside in the cache and will be permanently lost when the cache terminates unexpectedly.

At the other extreme, the relevant information of each access (hash of the key, the accessed object's size, and TTL attribute) can be streamed to an MRC service running on a different host that generates the corresponding MRCs and estimates the WSSes. The downside of this approach is that it increases network bandwidth usage that might also increase cache response times. But the advantage of this approach is that it is easier to make fault tolerant and that it consumes less memory and CPU on the cache side.

There may be other, more suitable designs that are more efficient. For example, it may make sense to implement HLL counters in the cache and then stream those to a server that uses these counters to generate MRCs (using CounterStacks^{++}) and estimate WSSes. And it may make sense to implement SHARDS^{++} cache-side to sample the information sent to the MRC service.

We have implemented a prototype MRC service and modified a variant of Redis called Codis [286] to use this service to allow us to study the above-mentioned trade-offs in more detail.

Cloud cache management. Our evaluation of numerous publicly available workloads showed that many of these workloads can be serviced by in-memory caches with a relatively small memory footprint, especially if the workload's data is TTL-endowed. This implies that to maximize resource efficiency, multiple cache instances should be co-located on available cache hosts. This raises interesting questions on how a fleet of cache hosts should be managed and what policies should be used for this management. We envision that the MRC service described above be extended to manage in-memory caches in cloud environments. Requests to instantiate an in-memory cache are directed to the MRC service, which then instantiates the cache (placement) on a target host depending on the MRC and WSS information it has collected previously for the target workload. Periodically, the MRC service will initiate a resizing of a cache based on the recent behavior of the cache's workload. To optimize hardware resource utilization without impacting performance, the MRC service may decide to migrate a cache instance from one host to another (e.g., when the sizes of several co-located cache instances are increased). How best to do this all in an automated fashion is an interesting open question.

Minimal traces. Producing the results of the evaluations presented in this dissertation was challenging. The combined traces we used consume approximately 15TiB of storage space for the metadata only. As more companies make their cache traces available, and the fact that in-memory datasets are doubling in size every few years, we expect storage space requirements to increase significantly [4, 25–27]. (For example, Twitter collected 300 one-week-long access traces (most of which were not made public) and these traces consumed 80TiB of storage [125].) Moreover, processing the traces used in our evaluation to obtain MRCs and WSS estimates consumed several months of compute time. This raises the interesting question of whether, given an access trace, it is possible to produce an equivalent access trace that is significantly smaller than the original target trace but exhibits the same caching behavior.

To begin exploring this question, we endeavored to, given an MRC, establish a trace of minimal length that would produce the same MRC. We were able to do this successfully for LRU-specific MRCs. This would allow us, for example, to generate MRCs for each one hour interval and generate a minimal trace for each. The traces for each successive interval can then potentially be concatenated to produce a smaller trace that exhibits the same caching behavior as the original trace for the LRU eviction policy. An interesting question is whether this can be generalized: given MRCs for several different eviction policies (e.g., LRU, LFU, FIFO), is it possible, given an original trace, to generate a significantly shorter trace that exhibits the same caching behavior as the original trace for all target eviction policies. If so, it would allow companies to distribute a set of MRCs instead of large traces and thus not have to worry about leaking privileged information.

Bibliography

[1] Arthur W Burks, Herman H Goldstine, and John Von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. In *The origins of digital computers: Selected papers*, pages 399–413. Springer, 1946.

[2] Herman H Goldstine and Adele Goldstine. The electronic numerical integrator and computer (ENIAC). *IEEE Annals of the History of Computing*, 18(1):10–16, 1996.

[3] Christianto C Liu, Ilya Ganusov, Martin Burtscher, and Sandip Tiwari. Bridging the processor-memory performance gap with 3D IC technology. *IEEE Design & Test of Computers*, 22(6):556–564, 2005.

[4] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. A modern primer on processing in memory. In *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann*, pages 171–243. Springer, 2022.

[5] Bryan Harris and Nihat Altiparmak. Ultra-Low latency SSDs impact on overall energy efficiency. In *Proc. 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'20)*, 2020.

[6] Ali Munir, Ihsan Ayyub Qazi, and Saad Bin Qaisar. On achieving low latency in data centers. In *Proc. Intl. Conf. on Communications (ICC'13)*, pages 3721–3725. IEEE, 2013.

[7] Todd Hoff. Latency is everywhere and it costs you sales - how to crush it. `https://highscalability.com/latency-is-everywhere-and-it-costs-you-sales-how-to-crush-it/`, 2009.

[8] Steve Souders. Blog post on Velocity. `http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html?ref=highscalability.com`, 2009.

[9] 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.

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

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

[12] Peter J Denning. Virtual memory. *ACM Computing Surveys (CSUR)*, 2(3):153–189, 1970.

[13] Peter J Denning. The locality principle. *Communications of the ACM*, 48(7):19–24, 2005.

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

[15] James Z. Teng and Robert A. Gumaer. Managing IBM database 2 buffers to maximize performance. *IBM Systems Journal*, 23(2):211–218, 1984.

[16] Louis Degenaro, Arun Iyengar, Ilya Lipkind, and Isabelle Rouvellou. A middleware system which intelligently caches query results. In *Proc. IFIP/ACM Intl. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware'00)*, pages 24–44, 2000.

[17] Michael N Nelson, Brent B Welch, and John K Ousterhout. Caching in the Sprite network file system. *ACM Trans. on Computer Systems (TOCS)*, 6(1):134–154, 1988.

[18] Alan J Smith. Disk cache—miss ratio analysis and design considerations. *ACM Trans. on Computer Systems (TOCS)*, 3(3):161–203, 1985.

[19] Pei Cao and Sandy Irani. Cost-Aware WWW proxy caching algorithms. In *Proc. Symp. on Internet Technologies and Systems (USITS'97)*, 1997.

[20] Maurice J. Bach. *The design of the UNIX Operating System.* Prentice-Hall, Inc., 1986.

[21] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, and David A Patterson. Raid: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26 (2):145–185, 1994.

[22] 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.

[23] Amazon. Caching challenges and strategies. https://aws.amazon.com/builders-library/caching-challenges-and-strategies/, Feb, 2024.

[24] Daniel S Berger. *Design and Analysis of Adaptive Caching Techniques for Internet Content Delivery.* PhD thesis, Technische Universität Kaiserslautern, 2018.

[25] Kimberly Keeton. Memory-driven computing (presentation). In *Proc. 15th Conf. on File and Storage Technologies (FAST'17)*, 2017.

[26] Dmitrii Ustiugov, Alexandros Daglis, Javier Picorel, Mark Sutherland, Edouard Bugnion, Babak Falsafi, and Dionisios Pnevmatikatos. Design guidelines for high-performance SCM hierarchies. In *Proc. the Intl. Symp. on Memory Systems*, pages 3–16, 2018.

[27] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. TMO: transparent memory offloading in datacenters. In *Proc. the 27th ACM Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*, pages 609–621, 2022.

[28] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. Understanding ephemeral storage for serverless analytics. In *Proc. USENIX Annual Technical Conf. (ATC'18)*, pages 789–794, 2018.

[29] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *Proc. 13th USENIX Symp. on Operating Systems Design and Implementation (OSDI'18)*, pages 427–444, 2018.

[30] Jiyi Wu, Lingdi Ping, Xiaoping Ge, Ya Wang, and Jianqing Fu. Cloud storage as the infrastructure of cloud computing. In *Proc. Intl. Conf. on Intelligent Computing and Cognitive Informatics*, pages 380–383, 2010.

[31] Memcached.org. Memcached. https://memcached.org. [Online; retrieved April 2023].

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

[33] Redis Labs. Redis. https://redis.io. [Online; retrieved April 2023].

[34] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *Proc. 9th Conf. on Operating Systems Design and Implementation (OSDI'10)*, page 47–60, 2010.

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

[36] Alex Hall, Alexandru Tudorica, Filip Buruiana, Reimar Hofmann, Silviu-Ionut Ganceanu, and Thomas Hofmann. Trading off accuracy for speed in PowerDrill. In *Proc. Intl. Conf. on Data Engineering, Chicago (ICDE'16)*, pages 2121–2132, 2016.

[37] 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.

[38] 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.

[39] 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.

[40] Amazon. Amazon Elasticache. https://aws.amazon.com/elasticache/. [Online; retrieved Apr-2023].

[41] Google. Google Memorystore. https://cloud.google.com/memorystore. [Online; retrieved April-2023].

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

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

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

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

[46] DigitalOcean. Managed Redis. https://www.digitalocean.com/products/managed-databases-redis/. [online; retrieved April-2023].

[47] Alibaba. ApsaraDB for Memcache. https://www.alibabacloud.com/product/apsaradb-for-memcache. [online; retrieved April-2023].

[48] ObjectRocket. ObjectRocket for Redis. https://www.objectrocket.com/managed-redis/. [online; retrieved April-2023].

[49] Tencent. TencentDB for Redis. https://intl.cloud.tencent.com/product/crs. [online; retrieved April-2023].

[50] 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.

[51] Sadhana Rai and Basavaraj Talawar. Nonvolatile memory technologies: Characteristics, deployment, and research challenges. *Frontiers of Quality Electronic Design (QED) AI, IoT and Hardware Security*, pages 137–173, 2023.

[52] Aniruddha N Udipi, Naveen Muralimanohar, Niladrish Chatterjee, Rajeev Balasubramonian, Al Davis, and Norman P Jouppi. Rethinking DRAM design and organization for energy-constrained multi-cores. In *Proc. 37th Intl. Symp. on Computer Architecture (ISCA'10)*, pages 175–186, 2010.

[53] Roy Bryant, Alexey Tumanov, Olga Irzak, Adin Scannell, Kaustubh Joshi, Matti Hiltunen, Andres Lagar-Cavilla, and Eyal De Lara. Kaleidoscope: Cloud micro-elasticity via VM state coloring. In *Proc. 6th Conf. on Computer Systems*, pages 273–286, 2011.

[54] Bogdan Nicolae, Pierre Riteau, and Kate Keahey. Transparent throughput elasticity for IAAS cloud storage using guest-side block-level caching. In *Proc. 7th Intl. Conf. on Utility and Cloud Computing*, pages 186–195, 2014.

[55] 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.

[56] 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.

[57] Private communication with the authors of [39], Sep 14, 2022.

[58] Memcached Maintainers. Over-provisioning in practice. `https://github.com/memcached/memcached/issues/543`. [Online; retrieved Jan-2024].

[59] Amazon. Scale your Amazon ElastiCache for Redis clusters with Auto Scaling. `https://aws.amazon.com/blogs/database/scale-your-amazon-elasticache-for-redis-clusters-with-auto-scaling/`, Jan, 2022.

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

[61] 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.

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

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

[64] 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.

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

[66] 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)*, page 177–188, 2004.

[67] 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.

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

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] 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. Design, Automation and Test in Europe Conf. and Exhibition (DATE'22)*, page 1197 – 1202, 2022.

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

[78] 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.

[79] Xiaojun Guo, Hua Wang, Ke Zhou, Hong Jiang, Yaodong Han, and Guangjie Xing. FLOWS: Balanced MRC profiling for heterogeneous object-size cache. In *Proc. European Conference on Computer Systems (EuroSys'24)*, pages 421–440, 2024.

[80] Peng Wang, Yu Liu, Ziqi Liu, Zhelong Zhao, Ke Liu, Ke Zhou, and Zhihai Huang. ε-LAP: A lightweight and adaptive cache partitioning scheme with prudent resizing decisions for content delivery networks. *IEEE Transactions on Cloud Computing*, 2024.

[81] Tom Walsh. Generating miss rate curves with low overhead using existing hardware. Master's thesis, University of Toronto, 2009.

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

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

[84] 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.

[85] 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.

[86] 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. ISBN 9780769546759. doi: 10.1109/IPDPS.2012.117.

[87] 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. ISBN 9781479989300. doi: 10.1109/HPCA.2015.7056022.

[88] 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.

[89] 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.

[90] 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.

[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. 18th ACM Intl. Conf. on Computing Frontiers (CF'21)*, pages 63–72, 2021.

[92] 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 (SC'23)*, pages 25–36, 2023.

[93] 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.

[94] M.C. Easton and B.T. Bennett. Transient-free working-set statistics. *Communications of the ACM*, 20(2):93 – 99, 1977. doi: 10.1145/359423.359431.

[95] 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.

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

[97] 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.

[98] 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.

[99] 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.

[100] 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.

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

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

[103] 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.

[104] 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. Measurement and Analysis of Computing Systems (POMACS'18*, pages 1–22, 2018.

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

[106] 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 Conf. (ATC'23)*, pages 467–484, 2023.

[107] Memcached. Github issues: LFU eviction policy #543. `https://github.com/memcached/memcached/issues/543`, . [Online; retrieved Jan-2024].

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

[109] 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.

[110] Twitter. Anonymized cache request traces from Twitter production. https://github.com/twitter/cache-trace, . [Online; retrieved April-2023].

[111] Tobias Ziegler, Carsten Binnig, and Viktor Leis. ScaleStore: A fast and cost-efficient storage engine using DRAM, NVMe, and RDMA. In *Proc. Intl. Conf. on Management of Data (SIGMOD'22)*, pages 685–699, 2022.

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

[113] 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.

[114] Damiano Carra, Giovanni Neglia, and Pietro Michiardi. Elastic provisioning of cloud caches: A cost-aware TTL approach. *Trans. on Networking*, 28(3):1283–1296, 2020.

[115] 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. doi: 10.1145/1416944.1416949.

[116] James Ryans. Using the EDGAR log file data set. Available at SSRN 2913612, 2017.

[117] 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].

[118] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, 2009.

[119] Zachary Drudi, Nicholas J. A. Harvey, Stephen Ingram, Andrew Warfield, and Jake Wires. Approximating hit rate curves using streaming algorithms. In *Proc. Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM'15)*, pages 225–241, 2015.

[120] Zachary Drudi. A streaming algorithms approach to approximating hit rate curves. Master's thesis, University of British Columbia, 2014.

[121] Sari Sultan, Kia Shakiba, Albert Lee, Michael Stumm, Ming Chen, and Chung-Man Abelard Chow. Systems and methods to generate a cache miss ratio curve where cache data has a time-to-live, May 9 2024. US Patent App. 17/982,136.

[122] Sari Sultan, Kia Shakiba, Albert Lee, Michael Stumm, Ming Chen, and Chung-Man Abelard Chow. Systems and methods to generate a miss ratio curve for a cache with variable-sized data blocks, May 16 2024. US Patent App. 17/982,070.

[123] Sari Sultan, Kia Shakiba, Albert Lee, Paul Chen, and Michael Stumm. TTLs Matter: Efficient cache sizing with TTL-aware miss ratio curves and working set sizes. In *Proc. European Conference on Computer Systems (EuroSys'24)*, pages 387–404, 2024.

[124] Kia Shakiba, Sari Sultan, and Michael Stumm. Kosmo: Efficient online miss ratio curve generation for eviction policy evaluation. In *Proc. 22nd USENIX Conf. on File and Storage Technologies (FAST'24)*, pages 89–105, 2024.

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

[126] Internet Engineering Task Force (IETF). Domain name system (DNS) IANA considerations. https://aws.amazon.com/elasticache/. [Online; retrieved Aug-2023].

[127] Jaeyeon Jung, Arthur W Berger, and Hari Balakrishnan. Modeling TTL-based internet caches. In *Proc. 22nd Conf. Computer and Communications Societies*, volume 1, pages 417–426, 2003.

[128] Nicaise Choungmo Fofack, Philippe Nain, Giovanni Neglia, and Don Towsley. Performance evaluation of hierarchical TTL-based cache networks. *Computer Networks*, 65:212–231, 2014.

[129] Daniel S Berger, Philipp Gland, Sahil Singla, and Florin Ciucu. Exact analysis of TTL cache networks. *Performance Evaluation*, 79:2–23, 2014.

[130] 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.

[131] Soumya Basu, Aditya Sundarrajan, Javad Ghaderi, Sanjay Shakkottai, and Ramesh Sitaraman. Adaptive TTL-based caching for content delivery. *Trans. on Networking*, 26(3): 1063–1077, 2018.

[132] Damiano Carra, Giovanni Neglia, and Pietro Michiardi. TTL-based cloud caches. In *Proc. Conf. on Computer Communications*, pages 685–693, 2019.

[133] 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.

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

[135] 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.

[136] 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.

[137] Memcached. Memcached protocol. https://github.com/memcached/memcached/blob/master/doc/protocol.txt, . [Online; retrieved Jan-2024].

[138] European Parliament and Council of the European Union. Regulation (EU) 2016/679 of the European Parliament and of the Council. https://data.europa.eu/eli/reg/2016/679/oj, 2016. Online 2024.

[139] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, and Mor Harchol-Balter. 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.

[140] Redis. Diving into Redis 6.0. https://redis.com/blog/diving-into-redis-6/, . [Online; retrieved Jan-2024].

[141] Redis. Discussion by the maintainers of Redis. https://news.ycombinator.com/item?id=19662501, . [Online; retrieved Jan-2024].

[142] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. Faster slab reassignment in Memcached. In *Proc. of the Intl. Symp. on Memory Systems (MEMSYS'19)*, pages 353–362, 2019.

[143] 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.

[144] 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 (APSsys'18)*, pages 1–8, 2018.

[145] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. Cache replacement based on reuse-distance prediction. In *Proc. 25th Intl. Conf. on Computer Design (ICCD'07)*, pages 245–250, 2007.

[146] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V Veidenbaum. Improving cache management policies using dynamic reuse distances. In *Proc. 45th Intl. Symp. on Microarchitecture (MICRO'12)*, pages 389–400, 2012.

[147] Nathan Beckmann and Daniel Sanchez. Cache calculus: Modeling caches through differential equations. *IEEE Computer Architecture Letters*, 16(1):1–5, 2015.

[148] RL Mattson. Role of optical memories in computer storage. *Applied Optics*, 13(4):755–760, 1974.

[149] Pierre Michaud. Some mathematical facts about optimal cache replacement. *ACM Trans. on Architecture and Code Optimization (TACO'16)*, 13(4):1–19, 2016.

[150] Nimrod Megiddo and Dharmendra S Modha. ARC: A self-tuning, low overhead replacement cache. In *Proc. 19th Conf. on File and Storage Technologies (FAST'03)*, pages 115–130, 2003.

[151] Akanksha Jain and Calvin Lin. Back to the future: Leveraging Belady's algorithm for improved cache replacement. In *Proc. 43rd Intl. Symp. on Computer Architecture (ISCA'16)*, pages 78–89, 2016.

[152] Terence Kelly and Daniel Reeves. Optimal web cache sizing: Scalable methods for exact solutions. *Computer Communications*, 24(2):163–173, 2001.

[153] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proc. IEEE Computer Communications (INFOCOM'99)*, volume 1, pages 126–134. IEEE, 1999.

[154] Laszlo A Belady, Robert A Nelson, and Gerald S Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of the ACM*, 12(6):349–353, 1969.

[155] Xiaoming Gu and Chen Ding. On the theory and potential of LRU-MRU collaborative cache management. In *Proc. Intl. Symp. on Memory Management (ISMM'11)*, page 43–54, 2011.

[156] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM'85)*, 32(3):652–686, 1985.

[157] Qingpeng Niu. PARDA git repository. https://bitbucket.org/niuqingpeng/file_parda/.

[158] Michael A Bender, Daniel DeLayo, Bradley C Kuszmaul, William Kuszmaul, and Evan West. Increment-and-freeze: Every cache, everywhere, all of the time. In *Proc. 35th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA'23)*, pages 129–139, 2023.

[159] Jacob Taylor Wires. *Storage system design for fast nonvolatile memories.* PhD thesis, University of British Columbia, 2017.

[160] Jacob Taylor Wires, Andrew Warfield, Stephen Frowe Ingram, Nicholas James Alexander Harvey, and Zachary Drudi. Systems, devices and methods for generating locality-indicative data representations of data streams, and compressions thereof, 2017. US Patent App. 15/308,162. 2017.

[161] Nick Harvery. Counter stacks: Storage workload analysis via streaming algorithms. https://web.archive.org/web/20171124130718/http://www.cs.ubc.ca/~nickhar/Talks/2014/HRC/HRC.pptx. [Online; retrieved 12-June-2020. No longer available as of 31-Jul-2020, archive.org is used to retrieve it.].

[162] Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[163] 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.

[164] 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. doi: 10.1007/3-540-45726-7_1.

[165] Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities. In *Proc. European Symp. on Algorithms*, pages 605–617, 2003.

[166] 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.

[167] 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.

[168] Frédéric Giroire. Order statistics and estimating cardinalities of massive data sets. *Discrete Applied Mathematics*, 157(2):406–427, 2009.

[169] 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.

[170] 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.

[171] 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.

[172] Hazar Harmouch and Felix Naumann. Cardinality estimation: An experimental survey. *Proc. of the VLDB Endowment*, 11(4):499–512, 2017.

[173] Carl A Waldspurger, Irfan Ahmad, Alexander Garthwaite, and Nohhyun Park. System and method for efficient cache utility curve construction and cache allocation, 2016. US Patent 9,418,020. 2016.

[174] Akanksha Jain and Calvin Lin. Cache replacement policies. *Synthesis Lectures on Computer Architecture*, 14(1):1–87, 2019.

[175] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. EELRU: simple and effective adaptive page replacement. *Performance Evaluation Review*, 27(1):122–133, 1999.

[176] R. Karedla, J.S. Love, and B.G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994. doi: 10.1109/2.268884.

[177] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *Proc. 34th Intl. Symp. on Computer Architecture (ISCA'07)*, page 381–391, 2007.

[178] Kaushik Rajan and Govindarajan Ramaswamy. Emulating optimal replacement with a shepherd cache. In *Proc. 40th Intl. Symp. on Microarchitecture (MICRO'07)*, pages 445–454, 2007.

[179] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proc. 37th Intl. Symp. on Computer Architecture (ISCA'10)*, pages 60–71, 2010.

[180] Daniel A Jiménez. Insertion and promotion for tree-based pseudoLRU last-level caches. In *Proc. 46th Intl. Symp. on Microarchitecture (MICRO'13)*, pages 284–296, 2013.

[181] Theodore Johnson and Dennis Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proc. 20th Intl. Conf. on Very Large Data Bases (VLDB'94)*, pages 439–450, 1994.

[182] John T Robinson and Murthy V Devarakonda. Data cache management using frequency-based replacement. In *Proc. Intl. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'90)*, pages 134–142, 1990.

[183] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. on Computers*, 50(12):1352–1361, 2001.

[184] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. FIFO queues are all you need for cache eviction. In *Proc. 29th Symp. on Operating Systems Principles (SOSP'23)*, pages 130–149, 2023.

[185] Nathan Beckmann and Daniel Sanchez. Maximizing cache performance under uncertainty. In *Proc. Intl. Symp. on High Performance Computer Architecture (HPCA'17)*, pages 109–120, 2017.

[186] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *Proc. 29th Intl. Symp. on Computer Architecture (ISCA'02)*, pages 209–220, 2002.

[187] Mazen Kharbutli and Yan Solihin. Counter-based cache replacement algorithms. In *Proc. Intl. Conf. on Computer Design (ICCD'05)*, pages 61–68, 2005.

[188] Priyank Faldu and Boris Grot. Leeway: Addressing variability in dead-block prediction for last-level caches. In *Proc. Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'17)*, pages 180–193, 2017.

[189] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proc. 28th Intl. Symp. on Computer Architecture (ISCA'01)*, pages 144–154, 2001.

[190] Vivek Seshadri, Onur Mutlu, Michael A Kozuch, and Todd C Mowry. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *Proc. 21st Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'12)*, pages 355–366, 2012.

[191] Samira Manabi Khan, Yingying Tian, and Daniel A Jimenez. Sampling dead block prediction for last-level caches. In *Proc. 43rd Intl. Symp. on Microarchitecture (MICRO'10)*, pages 175–186, 2010.

[192] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. SHiP: Signature-based hit predictor for high performance caching. In *Proc. 44th Intl. Symp. on Microarchitecture (MICRO'11)*, pages 430–441, 2011.

[193] Daniel A Jiménez and Elvira Teran. Multiperspective reuse prediction. In *Proc. 50th Intl. Symp. on Microarchitecture (MICRO'17)*, pages 436–448, 2017.

[194] Song Jiang and Xiaodong Zhang. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *Performance Evaluation Review*, 30(1):31–42, 2002.

[195] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *Proc. USENIX Annual Technical Conf. (ATC'01)*, pages 91–104, 2001.

[196] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *Proc. 15th USENIX Symp. on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, 2018.

[197] Gil Einziger, Roy Friedman, and Ben Manes. TinyLFU: A highly efficient cache admission policy. *ACM Trans. on Storage (ToS)*, 13(4):1–31, 2017.

[198] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The RocksDB experience. In *Proc. 19th Conf. on File and Storage Technologies (FAST'21)*, pages 33–49, 2021.

[199] Morteza Hoseinzadeh. A survey on tiering and caching in high-performance storage systems. *arXiv preprint arXiv:1904.11560*, 2019.

[200] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. CFLRU: A replacement algorithm for flash memory. In *Proc. Intl. Conf. on Compilers, Architecture and Synthesis for Embedded Systems (CASES'06)*, pages 234–241, 2006.

[201] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, and Cheng Chen. Improving flash-based disk cache with lazy adaptive replacement. *ACM Trans. on Storage (TOS'16)*, 12(2):1–24, 2016.

[202] Adam Leventhal. Flash storage memory. *Communications of the ACM*, 51(7):47–51, 2008.

[203] Chunling Wang, Dandan Wang, Yupeng Chai, Chuanwen Wang, and Diansen Sun. Larger cheaper but faster: SSD-SMR hybrid storage boosted by a new SMR-oriented cache framework. In *Proc. Symp. Mass Storage Syst. Technol.(MSST'17)*, 2017.

[204] JM Thorington and J David Irwin. An adaptive replacement algorithm for paged-memory computer systems. *IEEE Trans. on Computers*, 100(10):1053–1061, 1972.

[205] Hong-Tai Chou and David J DeWitt. An evaluation of buffer management strategies for relational database systems. *Algorithmica*, 1(1-4):311–336, 1986.

[206] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB'96)*, volume 96, pages 330–341, 1996.

[207] Gerhard Hasslinger, Juho Heikkinen, Konstantinos Ntougias, Frank Hasslinger, and Oliver Hohlfeld. Optimum caching versus LRU and LFU: Comparison and combined limited look-ahead strategies. In *Proc. Intl. Symp. on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt'18)*, pages 1–6, 2018.

[208] Redis. Redis eviction policies. https://redis.io/docs/manual/eviction/, . [Online].

[209] Jui-Hao Chiang, Tzi-Cker Chiueh, and Han-Lin Li. Memory pressure balancing on virtualized servers. In *Proc. 21st Intl. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA'15)*, page 70 – 79, 2015. doi: 10.1109/RTCSA.2015.29.

[210] Deepavali Bhagwat, Mahesh Patil, Michal Ostrowski, Murali Vilayannur, Woon Jung, and Chethan Kumar. A practical implementation of clustered fault tolerant write acceleration in a virtualized environment. In *13th USENIX Conf. on File and Storage Technologies (FAST 15)*, pages 287–300, 2015.

[211] Zhigang Wang, Xiaolin Wang, Fang Hou, Yingwei Luo, and Zhenlin Wang. Dynamic memory balancing for virtualization. *ACM Trans. on Architecture and Code Optimization*, 13(1), 2016.

[212] 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)*, page 17, 2011.

[213] 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.

[214] Edsger W. Dijkstra. Sets are unibags. https://www.cs.utexas.edu/users/EWD/transcriptions/EWD07xx/EWD786a.html, 1989. [Online; retrieved 2022].

[215] 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.

[216] Saguiitay. Cardinality estimation. https://github.com/microsoft/CardinalityEstimation/. [Online; retrieved April 2023].

[217] 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.

[218] G Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proc. 8th Intl. Symp. on High Performance Computer Architecture (HPCA'02)*, pages 117–128, 2002.

[219] Jan Gecsei. Determining hit ratios for multilevel hierarchies. *IBM Journal of Research and Development*, 18(4):316–327, 1974.

[220] Gabriel M Silberman. Stack processing techniques in delayed-staging storage hierarchies. *Communications of the ACM*, 26(11):999–1007, 1983.

[221] Brian Walter O'Krafka. *Design and evaluation of directory-based cache coherence systems.* PhD thesis, University of California, Berkeley, 1991.

[222] Rabin Andrew Sugumar. *Multi-configuration simulation algorithms for the evaluation of computer architecture designs.* PhD thesis, University of Michigan, 1993.

[223] GS Shedler and DR Slutz. Derivation of miss ratios for merged access streams. *IBM Journal of Research and Development*, 20(5):505–517, 1976.

[224] Yutao Zhong and Wentao Chang. Sampling-based program locality approximation. In *Proc. 7th Intl. Symp. on Memory Management (ISMM'08)*, pages 91–100, 2008.

[225] Cheng Pan, Xiaolin Wang, Yingwei Luo, and Zhenlin Wang. Penalty-and locality-aware memory allocation in Redis using enhanced AET. *ACM Trans. on Storage (TOS'21)*, 17 (2):1–45, 2021.

[226] Pan Cheng. AETLib. https://github.com/PanCheng111/AETLib. [Online; retrieved 2022].

[227] Peter J Denning and Stuart C Schwartz. Properties of the working-set model. *Communications of the ACM*, 15(3):191–198, 1972.

[228] Erik Berg and Erik Hagersten. Statcache: A probabilistic approach to efficient and accurate data locality analysis. In *Proc. Intl. Symp. on Performance Analysis of Systems and Software (ISPASS'04)*, pages 20–27. IEEE, 2004.

[229] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *Proc. Conf. on Programming Language Design and Implementation (PLDI'03)*, pages 245–257, 2003.

[230] Yutao Zhong, Xipeng Shen, and Chen Ding. Program locality analysis using reuse distance. *ACM Trans. on Programming Languages and Systems (TOPLAS'09)*, 31(6):1–39, 2009.

[231] Richard E. Kessler, Mark D Hill, and David A Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Trans. on Computers*, 43(6):664–675, 1994.

[232] Calin Cascaval, Evelyn Duesterwald, Peter F Sweeney, and Robert W Wisniewski. Multiple page size modeling and optimization. In *Proc. 14th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 339–349, 2005.

[233] Derek L Schuff, Milind Kulkarni, and Vijay S Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proc. 19th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'10)*, pages 53–64, 2010.

[234] Kristof Beyls and Erik H D'Hollander. Discovery of locality-improving refactorings by reuse path analysis. In *Proc. 2nd Intl. Conf. on High Performance Computing and Communications (HPCC'06)*, pages 220–229, 2006.

[235] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Trans. on Mathematical Software (TOMS'85)*, 11(1):37–57, 1985.

[236] Thomas Martin Conte. *Systematic computer architecture prototyping.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[237] Thomas M. Conte, Mary Ann Hirsch, and W-MW Hwu. Combining trace sampling with single pass methods for efficient cache simulation. *IEEE Trans. on Computers*, 47(6):714–720, 1998.

[238] Albert Lee. PR-MRC: MRC construction using non-statistical sampling. Master's thesis, University of Toronto (Canada), 2022.

[239] Yul H Kim, Mark D Hill, and David A Wood. Implementing stack simulation for highly-associative memories. *ACM SIGMETRICS Performance Evaluation Review*, 19(1): 212–213, 1991.

[240] Kexiang Wang. A4S: Co-designing scaling and placement in stream processing systems with memory-parallelism curves. Master's thesis, University of Toronto (Canada), 2023.

[241] Changpeng Fang, S Can, Soner Onder, and Zhenlin Wang. Instruction based memory distance analysis and its application to optimization. In *14th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 27–37. IEEE, 2005.

[242] Thomas M Conte and Wen-Mei W Hwu. Single-pass memory system evaluation for multiprogramming workloads. Technical Report UILU-ENG-90-2214, CSG-122, University of Illinois at Urbana-Champaign Coordinated Science Laboratory, 1990.

[243] Xudong Shi, Feiqi Su, Jih-Kwon Peir, Ye Xia, and Zhen Yang. Modeling and stack simulation of CMP cache capacity and accessibility. *IEEE Trans. on Parallel and Distributed Systems*, 20(12):1752–1763, 2009.

[244] C Eric Wu, Yarsun Hsu, and Y-H Liu. Stack simulation for set-associative V/R-type caches. In *Proc. 16th Intl. Computer Software and Applications Conf.*, pages 332–333, 1992.

[245] W-MW Hwu and Thomas M Conte. The susceptibility of programs to context switching. *IEEE Trans. on Computers*, 43(9):994–1003, 1994.

[246] C Eric Wu, Yarsun Hsu, and Yew-Huey Liu. Efficient stack simulation for shared memory set-associative multiprocessor caches. In *Proc. Intl. Conf. on Parallel Processing (ICPP'93)*, volume 1, pages 163–170, 1993.

[247] Lulu He, Zhibin Yu, and Hai Jin. FractalMRC: online cache miss rate curve prediction on commodity systems. In *Proc. 26th Intl. Parallel and Distributed Processing Symp. (IPDPS'12)*, pages 1341–1351, 2012.

[248] Daniel S Berger, Sebastian Henningsen, Florin Ciucu, and Jens B Schmitt. Maximizing cache hit ratios by variance reduction. *ACM SIGMETRICS Performance Evaluation Review*, 43(2): 57–59, 2015.

[249] Nicolas Gast and Benny Van Houdt. Transient and steady-state regime of a family of list-based cache replacement algorithms. In *Proc. Intl. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'15*, pages 123–136, 2015.

[250] Erol Gelenbe. A unified approach to the evaluation of a class of replacement algorithms. *IEEE Trans. on Computers*, 100(6):611–618, 1973.

[251] Edward Grady Coffman and Peter J Denning. *Operating Systems Theory.* Prentice-Hall Englewood Cliffs, NJ, 1973.

[252] John McCabe. On serial files with relocatable records. *Operations Research*, 13(4):609–618, 1965.

[253] PJ Burville and JFC Kingman. On a model for storage and search. *Journal of Applied Probability*, 10(3):697–701, 1973.

[254] Asit Dan and Don Towsley. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *Proc. Intl. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'90)*, pages 143–152, 1990.

[255] Predrag R Jelenković. Asymptotic approximation of the move-to-front search cost distribution and least-recently used caching fault probabilities. *The Annals of Applied Probability*, 9(2): 430–464, 1999.

[256] Predrag R Jelenković and Ana Radovanović. Least-recently-used caching with dependent requests. *Theoretical Computer Science*, 326(1-3):293–327, 2004.

[257] Antonis Panagakis, Athanasios Vaios, and Ioannis Stavrakakis. Approximate analysis of LRU in the case of short term correlations. *Computer Networks*, 52(6):1142–1152, 2008.

[258] Konstantinos Psounis, An Zhu, Balaji Prabhakar, and Rajeev Motwani. Modeling correlations in web traces and implications for designing replacement policies. *Computer Networks*, 45(4): 379–398, 2004.

[259] Massimo Gallo, Bruno Kauffmann, Luca Muscariello, Alain Simonian, and Christian Tanguy. Performance evaluation of the random replacement policy for networks of caches. *ACM SIGMETRICS Performance Evaluation Review*, 40(1):395–396, 2012.

[260] Elizabeth J O'neil, Patrick E O'Neil, and Gerhard Weikum. An optimality proof of the LRU-K page replacement algorithm. *Journal of the ACM (JACM'99)*, 46(1):92–112, 1999.

[261] Christine Fricker, Philippe Robert, and James Roberts. A versatile and accurate approximation for LRU cache performance. In *Proc. 24th Intl. Teletraffic Congress (ITC'12)*, pages 1–8, 2012.

[262] Marek Chrobak, Gerhard J Woeginger, Kazuhisa Makino, and Haifeng Xu. Caching is hard—even in the fault model. *Algorithmica*, 63(4):781–794, 2012.

[263] I. L. Traiger, , and D. R. Slutz. One-pass techniques for the evaluation of memory hierarchies. Technical report, Thomas J. Watson IBM Research Center., 1971.

[264] Parijat Dube, Li Zhang, David Daly, and Alan Bivens. Performance of large low-associativity caches. *ACM SIGMETRICS Performance Evaluation Review*, 37(4):11–18, 2010.

[265] David Daly, Parijat Dube, Kaoutar El Maghraoui, Dan Poff, and Li Zhang. A hybrid approach for large cache performance studies. In *Proc. 8th Intl. Conf. on Quantitative Evaluation of Systems*, pages 47–56, 2011.

[266] IL Traiger and RL Mattson. The evaluation and selection of technologies for computer storage systems. In *Proc. AIP Conf.*, pages 1–12, 1972.

[267] James G Thompson and Alan Jay Smith. Efficient (stack) algorithms for analysis of write-back and sector memories. *ACM Trans. on Computer Systems (TOCS'89)*, 7(1):78–117, 1989.

[268] Susan J Eggers, Edward D Lazowska, and Yi-Bing Lin. Techniques for the trace-driven simulation of cache performance. In *Proc. 21st Conf. on Winter Simulation (WSC'89)*, pages 1042–1046, 1989.

[269] Rabin A Sugumar and Santosh G Abraham. Set-associative cache simulation using generalized binomial trees. *ACM Trans. on Computer Systems (TOCS'95)*, 13(1):32–56, 1995.

[270] Jari Ahola. RECET – a real-time cache evaluation tool. In *Proc. 3rd Intl. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'95.)*, pages 376–381, 1995.

[271] CK Chow. On optimization of storage hierarchies. *IBM Journal of Research and Development*, 18(3):194–203, 1974.

[272] Dominique Thiebaut. On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio. *IEEE Trans. on Computers*, 38(7):1012–1026, 1989.

[273] Wen-Hann Wang and Jean-Loup Baer. Efficient trace-driven simulation method for cache performance analysis. *ACM SIGMETRICS Performance Evaluation Review*, 18(1):27–36, 1990.

[274] Wen-Hann Wang and Jean-Loup Baer. Efficient trace-driven simulation methods for cache performance analysis. *ACM Trans. on Computer Systems (TOCS'91)*, 9(3):222–241, 1991.

[275] Mark Holliday. Techniques for cache and memory simulation using address reference traces. *Intl. Journal in Computer Simulation*, 1(1):129–151, 1991.

[276] Darren Erik Vengroff and Guang R Gao. Partial sampling with reverse state reconstruction: A new technique for branch predictor performance estimation. In *Proc. 4th Intl. Symp. on High-Performance Computer Architecture (HPCA'98)*, pages 342–351, 1998.

[277] Parijat Dube, Michael Tsao, Li Zhang, and Alan Bivens. Performance modeling and characterization of large last level caches. In *Proc. 20th Intl. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 379–388, 2012.

[278] Patrick M Kelly and T Michael Cannon. Candid: Comparison algorithm for navigating digital image databases. In *Proc. 7th Intl Working Conf. on Scientific and Statistical Database Management*, pages 252–258, 1994.

[279] BP Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.

[280] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, 31(6):1794–1813, 2002.

[281] Jing Wei and Xin-fa Zeng. Optimal computing resource allocation algorithm in cloud computing based on hybrid differential parallel scheduling. *Cluster Computing*, 22:7577–7583, 2019.

[282] Junshe Wang, Jinliang Liu, and Hongbin Zhang. Access control based resource allocation in cloud computing environment. *Intl. Journal of Network Security*, 19(2):236–243, 2017.

[283] Shuja ur Rehman Baig, Waheed Iqbal, Josep Lluis Berral, and David Carrera. Adaptive sliding windows for improved estimation of data center resource utilization. *Future Generation Computer Systems*, 104:212–224, 2020.

[284] Éric Fusy and Frécéric Giroire. Estimating the number of active flows in a data stream over a sliding window. In *Proc. 4th Workshop on Analytic Algorithmics and Combinatorics (ANALCO'07)*, pages 223–231, 2007.

[285] Zijie Zeng, Lin Cui, Mimi Qian, Zhen Zhang, and Kaimin Wei. A survey on sliding window sketch for network measurement. *Computer Networks*, 226:109696, 2023.

[286] Codis. Codis - a Redis cluster solution. https://github.com/CodisLabs/codis. [Online].