Efficient In-Memory Cache Fleet Orchestration

by

Kia Shakiba

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

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

© Copyright 2026 by Kia Shakiba

Efficient In-Memory Cache Fleet Orchestration

Kia Shakiba
Doctor of Philosophy
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
2026

Abstract

In-memory caches, such as Redis and Memcached, play an important role in reducing data access latencies and the load on backend data stores by serving frequently accessed data from main memory. The effectiveness of such a cache is largely dependent on its configured parameters, such as its size and eviction policy. Unfortunately, selecting these parameters is an arduous task and therefore a typical in-memory cache often simply uses a default, non-optimal configuration which remains static for the lifetime of the cache.

The thesis of this dissertation is that significant performance and resource usage benefits can be realized through periodic cache reconfigurations and that the existing cache orchestration techniques are limited in their abilities to configure caches under modern workloads. This is primarily evident when managing large fleets of hosting servers, each running multiple caches. Previous cache orchestration techniques are limited to managing a small number of caches running on a single host.

In this dissertation, we present a comprehensive analysis of several real-world publicly-available in-memory cache workloads from which we obtain several insights. First, we outline that an effective cache orchestrator requires the accurate and efficient modeling of a cache's performance as a function of its configuration parameters. We show that a cache's eviction policy can have significant effects on its performance, though existing modeling techniques are limited in their abilities to efficiently demonstrate these effects. We propose a novel modeling technique, *Kosmo*, which accurately and efficiently models a cache's performance under various eviction policies, online. Second, we show that existing in-memory caches are unable to exploit the findings of our modeling technique as they are limited in their abilities to switch eviction policies at runtime. To address this, we propose a novel in-memory cache, *PaperCache*, which is capable of efficiently switching between any eviction policy at runtime. Finally, we demonstrate the effects of periodically modifying a cache's configuration parameters on its performance and show that significant benefits can be achieved through global optimizations across a fleet of hosting servers. We propose *Flux*, a novel online in-memory cache orchestrator which uses our proposed modeling technique, *Kosmo*, and in-memory cache, *PaperCache*, to significantly improve the resource usage and performance of the caches it manages.

Acknowledgements

Firstly, I would like to thank my supervisor, Prof. Michael Stumm, whose guidance and leadership is what led me to where I am today. The Ph.D. journey is one of many ups and downs, and having a mentor such as Michael made the journey enjoyable. Thank you for always making time when I needed it and helping me navigate the difficult times in my Ph.D. process. I am forever grateful for all the effort you put into ensuring your students feel well supported – it has not gone unnoticed. Not only did you support me in my research, your guidance also helped me become a well-rounded individual. You once gave me a piece of advice that turned out to be the most useful I have ever received: *never take advice from anyone.*

Thank you to my supervisory committee – Prof. Ashvin Goel and Prof. Ding Yuan – for all of your feedback and support during my degree. Your guidance has been invaluable throughout my research journey. I would also like to thank my fellow lab mates – Albert, David, Griffin, and Hooman – for always making the lab an inviting and enjoyable place. I am grateful for all of your helpful comments and suggestions in my work. To Prof. Willy Zwaenepoel, thank you for your help and guidance; your extensive experience and fresh perspective helped elevate the quality of my work.

To my parents, Hossein and Narges, thank you for everything you have done for me, you are truly the best parents for which anyone could ask. Having now gone through the Ph.D. journey myself, I have even greater respect for your dedication to the pursuit of higher education, all while supporting a young family in a new country. It was only through the examples you set that I was able to get to where I am today. Your encouragement for me to follow my interests led me to a field that I truly enjoy and for that, I thank you.

To Nika and Mario, thank you for always being there as the role models of my life. Nika, you always pushed me to be the best version of myself and, despite the countless years of being bullied by you, you helped make me the person I am today. From editing my undoubtedly terrible essays in grade school to helping me with my university applications, I could always rely on your support for any challenge I was facing. I am forever grateful and proud to have you as my sister. Mario, you have always been an invaluable sounding board throughout my personal and academic career. Your ability to remain calm and rational whenever I needed advice is second only to your talent in cooking – for both of which I am grateful. To Ava, the newest edition to our family, I know that with parents as great as yours you will undoubtedly grow into an amazing person, and likely one day, an engineer.

Finally, I would like to thank my partner and the love of my life, Navan. Thank you for the endless support you have given me throughout my Ph.D. journey. I cannot put into words what it means to know that you are always there to help me tackle whatever I face. The Ph.D. journey can be straining at times and it was only because of you that I was able to make it through. From reading through my papers to staying up with me all night during submissions, you were always ready to help. I can honestly say that without you, I would not have been able to achieve all that I have – they are just as much your achievements as they are mine. You have been with me from the beginning of this journey and it is only fitting that it is here that I ask you to marry me.

Contents

1 Introduction **1**

1.1 Exploiting access patterns . . . 2
1.2 Managing in-memory caches in cloud environments . . . 3
1.3 Miss Ratio Curves (MRCs) . . . 4
1.4 Eviction policies . . . 5
1.5 Thesis and contributions . . . 7
1.6 Organization . . . 8

2 Background **9**

2.1 In-memory caches . . . 9
2.1.1 Memory allocation . . . 10
2.1.2 Eviction policies . . . 11
2.1.3 Time to Live (TTL) . . . 16
2.1.4 Wire protocol . . . 17
2.2 Modeling in-memory caches . . . 17
2.2.1 Working Set Size (WSS) analysis . . . 18
2.2.2 Miss Ratio Curves (MRCs) . . . 18
2.3 Utilizing cache models for resource redistribution . . . 25
2.3.1 Cache migration . . . 26

3 Kosmo: Efficient Modeling of Non-LRU Eviction Policies **28**

3.1 Background . . . 30
3.1.1 Inclusion property . . . 30
3.1.2 Miniature Simulations . . . 33
3.2 Kosmo . . . 34
3.2.1 Kosmo data structures . . . 34
3.2.2 The Kosmo algorithm . . . 35
3.2.3 Optimizations . . . 36
3.2.4 Kosmo for LFU . . . 38
3.2.5 Other eviction policies . . . 40
3.2.6 Variable object sizes . . . 45
3.2.7 TTLs . . . 46
3.2.8 Simultaneous MRC generation . . . 47
3.3 Evaluation . . . 47

3.3.1 MiniSim implementation . . . 48
3.3.2 Environment . . . 48
3.3.3 Metrics . . . 49
3.3.4 Results . . . 49
3.3.5 Inclusion property violations . . . 52
3.4 Related work . . . 53
3.5 Conclusion . . . 53

4 PaperCache: Multi-Eviction Policy In-Memory Caching 55
4.1 Background and motivation . . . 56
4.1.1 Multi-eviction policy support in Redis . . . 56
4.1.2 SHARDS sampling . . . 57
4.1.3 Selecting the optimal eviction policy . . . 58
4.2 PaperCache . . . 58
4.3 Evaluation . . . 60
4.3.1 Policy switching accuracy . . . 61
4.3.2 Lazy eviction performance . . . 61
4.3.3 Memory overhead . . . 62
4.3.4 Latency performance . . . 62
4.3.5 CPU usage . . . 63
4.3.6 MiniStack efficacy duration . . . 64
4.3.7 Automatic policy switching behavior . . . 65
4.4 Conclusion . . . 69

5 Flux: Multi-Cache, Multi-Host Online Orchestration 70
5.1 Analysis of Caching Workloads . . . 72
5.1.1 Working set size analysis . . . 73
5.1.2 Miss ratio curve analysis . . . 73
5.1.3 Dynamic adjustment of cache sizes . . . 74
5.1.4 Periodic eviction policy switching . . . 75
5.2 Cache Orchestration . . . 77
5.2.1 Optimization problem . . . 77
5.2.2 Optimization method . . . 78
5.3 Flux . . . 79
5.3.1 Cache requirements . . . 80
5.3.2 Constraints . . . 81
5.3.3 Genetic algorithm . . . 81
5.3.4 Cache migration . . . 82
5.3.5 Flux implementation . . . 82
5.4 Case Studies . . . 82
5.4.1 Experimental Setup . . . 83
5.4.2 Choice of Objective Function . . . 84
5.4.3 Small-scale experiments . . . 85
5.4.4 Cloudphysics workloads . . . 88

5.4.5 Alibaba workloads . . . 90
5.4.6 Choice of Flux parameters . . . 92
5.5 Related work . . . 92
5.5.1 Comparison to prior work . . . 93
5.6 Concluding remarks . . . 94

6 Concluding Remarks **95**
6.1 Contributions . . . 95
6.2 Future research directions . . . 96

Bibliography **97**

List of Tables

2.1 Summary of popular eviction policies 12
2.2 Sample trace with stack distance calculations 19
2.3 Histogram of stack distances for the sample trace of Table 2.2 20

3.1 Memory usage of MiniSim for various eviction policies 28
3.2 Example of LFU violating the inclusion property 31
3.3 Example of FIFO violating the inclusion property 31
3.4 Example of 2Q violating the inclusion property 32
3.5 Example of LRFU violating the inclusion property 32
3.6 Example of MRU violating the inclusion property 32
3.7 Access trace datasets used in our simulations 47

4.1 Memory overheads of Redis and PaperCache 62
4.2 Access latency percentiles of Redis and PaperCache 63
4.3 MiniStack small efficacy duration outliers 65
4.4 % time each policy achieves the lowest and strictly lowest miss ratios 65

5.1 Access trace datasets used in our analysis. 72
5.2 Host hour and GiB hour reductions attained by Flux 83
5.3 Workloads used in our experiments 86

List of Figures

1.1 Accessed objects' frequency distributions . . . 2
1.2 Zipfian α values for real-world caching workloads . . . 2
1.3 Typical cloud-hosted in-memory cache environment . . . 3
1.4 MRCs for the same access traces as in Figure 1.1 . . . 4
1.5 MRC generated for the `src1` workload in the MSR dataset . . . 5
1.6 MRCs for the LRU, LFU, and FIFO eviction policies . . . 6
1.7 Average miss ratio savings versus LRU . . . 6
1.8 Accessed objects' reaccess duration CDFs . . . 7

2.1 RSS over time for a Redis cache configured with jemalloc and libc-malloc . . . 10
2.2 LRU and MRU eviction policies . . . 12
2.3 LFU eviction policy . . . 13
2.4 FIFO eviction policy . . . 13
2.5 CLOCK eviction policy . . . 14
2.6 2Q eviction policy . . . 15
2.7 LRFU eviction policy . . . 15
2.8 S3-FIFO eviction policy . . . 16
2.9 SIEVE eviction policy . . . 16
2.10 WSS and minimal size to achieve ideal miss ratio . . . 18
2.11 The MRC obtained by processing the sample trace of Table 2.2 . . . 20
2.12 Sample trace Olken tree . . . 21
2.13 SHARDS algorithm with fixed-size variant steps . . . 24
2.14 WSS approximation with fixed-rate SHARDS for varying sampling rates . . . 25
2.15 Stop-and-copy migration . . . 26
2.16 Source-based migration . . . 26
2.17 Destination-based migration . . . 26
2.18 Hybrid migration . . . 27

3.1 MRCs for ideal and practical LFU caches . . . 31
3.2 MAEs for different Kosmo granularity parameters . . . 37
3.3 The average number of eviction records per object in the Kosmo global table . . . 38
3.4 MRCs for the MSR web workload for fixed versus variable-sized objects . . . 46
3.5 Memory usage of Kosmo and MiniSim for all eviction policies . . . 50
3.6 Throughput of Kosmo and MiniSim for all eviction policies . . . 50

3.7 MAE of Kosmo and MiniSim for all eviction policies 51
3.8 CPU time per access for LFU, FIFO, 2Q, and LRFU for MiniSim and Kosmo 51
3.9 Ratio of accesses for which a violation of the inclusion property occurs 52

4.1 Miss ratios of LFU, LRU, and Redis switching from LFU to LRU 57
4.2 Hourly LRU, LFU, and FIFO miss ratios 58
4.3 An overview of PaperCache 58
4.4 Miss ratios of LFU, LRU, and PaperCache switching from LFU to LRU 61
4.5 PaperCache used size versus WSS 62
4.6 Total CPU usage of PaperCache when switching to 2Q, S3-FIFO, and SIEVE 63
4.7 Total CPU usage of Redis when switching to LFU 64
4.8 MiniStack efficacy durations for all considered datasets 65
4.9 Instances where each PaperCache eviction policy achieves the lowest miss ratio . . . 66
4.10 WSS over time for the Cloudphysics `w42` trace 66
4.11 Zipfian α over time for the Cloudphysics `w42` trace 67
4.12 Autocorrelation coefficient for the Cloudphysics `w42` trace 67
4.13 Cardinality over time for the Cloudphysics `w42` trace 67
4.14 Number of eviction policy switches versus cache size (% WSS) 68
4.15 % traces which observe at least 2 policy switches 69
4.16 Miss ratio savings by performing eviction policy switching 69

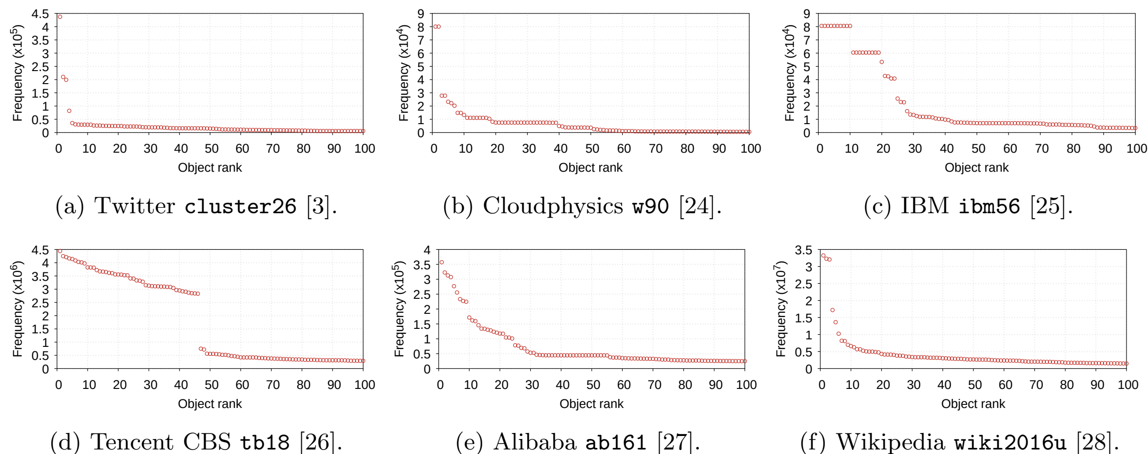
5.1 CDF of the WSS and the minimum cache sizes 73
5.2 MRCs for the LRU eviction policy for three consecutive hours 73
5.3 Cache size over time when resized to the minimum required each hour 75
5.4 Average memory savings across all workloads for different eviction policies 75
5.5 Miss ratio reduction when actively switching eviction policies 76
5.6 Three MRCs for the LFU, LRU, and FIFO eviction policies in 1-hour epochs 76
5.7 Stats from small-scale experiment with 256GiB hosts 86
5.8 Stats from small-scale experiment with 1,024GiB hosts 87
5.9 Stats from Cloudphysics experiment with 256GiB hosts 88
5.10 Stats from Cloudphysics experiment with 256GiB hosts with 5 fixed-size caches . . . 89
5.11 Stats from Cloudphysics experiment with 1,024GiB hosts 90
5.12 Stats from Alibaba experiment with 256GiB hosts 91
5.13 Stats from Alibaba experiment with 1,024GiB hosts 91
5.14 Aggregate misses of caches managed by Flux and an exhaustive search approach . . 93
5.15 Runtime of an exhaustive search orchestrator 93

Chapter 1

Introduction

In-memory caches are pervasive in large-scale distributed systems and play a critical role in reducing data access latency and the load on backend data stores by serving data directly from DRAM [1–15]. They store frequently accessed, "hot" data in the form of key-value pairs (referred to as *objects*), accessible through commands such as `GET` (to retrieve data from the cache) or `SET` (to insert data into the cache). A common use of these caches is in applications that access large storage servers, such as databases, to serve user requests. By using an in-memory cache, applications can reduce the latency of data access requests by serving data directly from main memory instead of the slower backend stores; in turn, the cache reduces the load on the backend stores, making the application more scalable. Two popular in-memory caches are Redis [16] and Memcached [17], both of which are open source. The overall goal of our research is to devise an automated cache orchestration framework which models and periodically reconfigures the configurations of managed in-memory caches on a fleet of hosting servers, online.

In-memory caches can consume a large portion of an application's operating budget, sometimes exceeding 60% of the total operating cost [18]. These caches are typically priced dependent on their allocated size (i.e., the larger the cache, the more it costs). As such, their performance is vital to ensuring their use is worth the added cost. A key performance metric for caches is the *miss ratio*. The miss ratio is defined as the number of cache misses (i.e., accesses to objects that do not exist in the cache) divided by the total number of accesses to the cache. Ideally, a cache should have as low a miss ratio as possible. Two key configuration parameters which significantly affect a cache's miss ratio are: (i) its *size* (which indicates the maximum aggregate size of its stored objects), and (ii) its *eviction policy* (which is used to select objects for removal from a full cache when new objects are to be inserted). Because a larger cache can hold more data than a smaller cache, the miss ratio is intuitively a function of the cache's allocated size, where as the cache size increases, the miss ratio generally decreases. Many eviction policies exist though selecting the optimal eviction policy (i.e., the eviction policy which will yield the lowest miss ratio) is nontrivial. Typical users of caches often select the cache's default eviction policy, or use a "rule-of-thumb" approach based on prior knowledge of their workload, which can result in suboptimal performance. Much prior work has focused on improving cache miss ratios [19–23].



(a) Twitter `cluster26` [3]. (b) Cloudphysics `w90` [24]. (c) IBM `ibm56` [25].

(d) Tencent CBS `tb18` [26]. (e) Alibaba `ab161` [27]. (f) Wikipedia `wiki2016u` [28].

Figure 1.1: Accessed objects' frequency distributions (top 100 most frequently accessed objects) for 6 traces from the Twitter [3], Cloudphysics [24], IBM [25], Tencent CBS [26], Alibaba [27], and Wikipedia [28] datasets. Note the different scales of each y-axis.

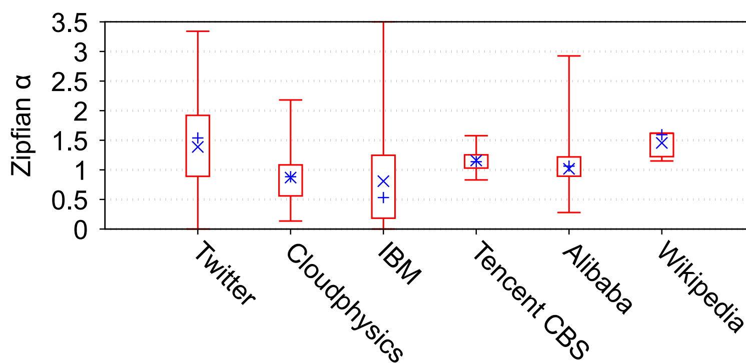


Figure 1.2: Zipfian α values for real-world caching workloads. The bottom and top lines identify the minimum and maximum results, respectively. The bottom and top of each box are the 25th and 75th results, respectively. The $\times$ and $+$ symbols indicate the mean and median results, respectively.

1.1 Exploiting access patterns

In-memory caches exploit common access patterns in workloads to significantly reduce the number of accesses to backend data stores. One such access pattern is that which exhibits a Zipfian distribution (i.e., a distribution which follows Zipf's law [29]) wherein a relatively small number of objects compose the majority of accesses [30–35]. Intuitively, these objects are good candidates to reside in an in-memory cache. Figure 1.1 shows the object access frequency distributions (up to the 100th most frequently-accessed object) for 6 randomly selected real-world cache access traces from the Twitter [3], Cloudphysics [24], IBM [25], Tencent CBS [26], Alibaba [27], and Wikipedia [28] datasets. In these traces, the 50 most popular objects get accessed 6.7 times more frequently than the next 50 objects, on average.

Zipf's law states that the ith most popular object of a workload has a relative access frequency of $1/i^{\alpha}$, where an α value ≥ 1 indicates the workload exhibits a strong Zipfian distribution [29].

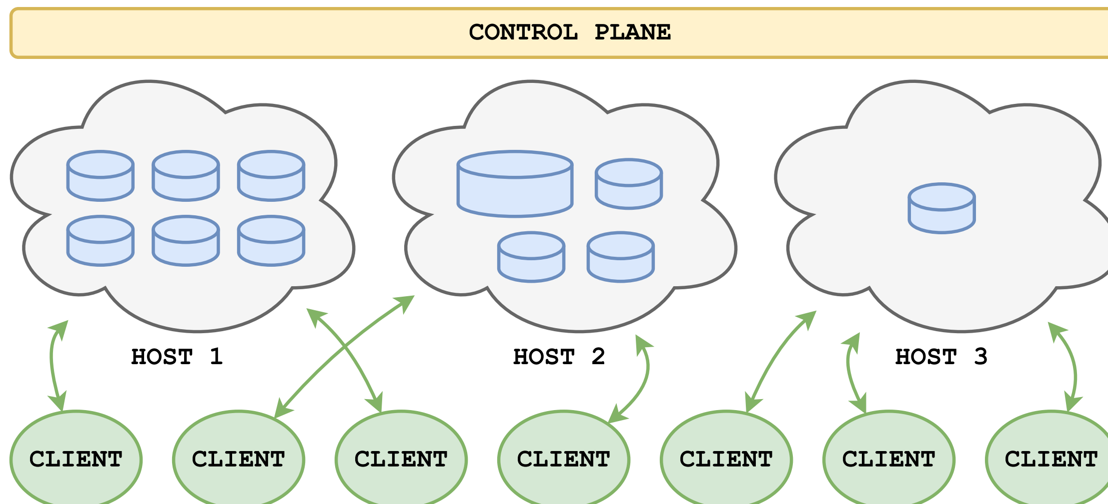


Figure 1.3: Typical cloud-hosted in-memory cache environment.

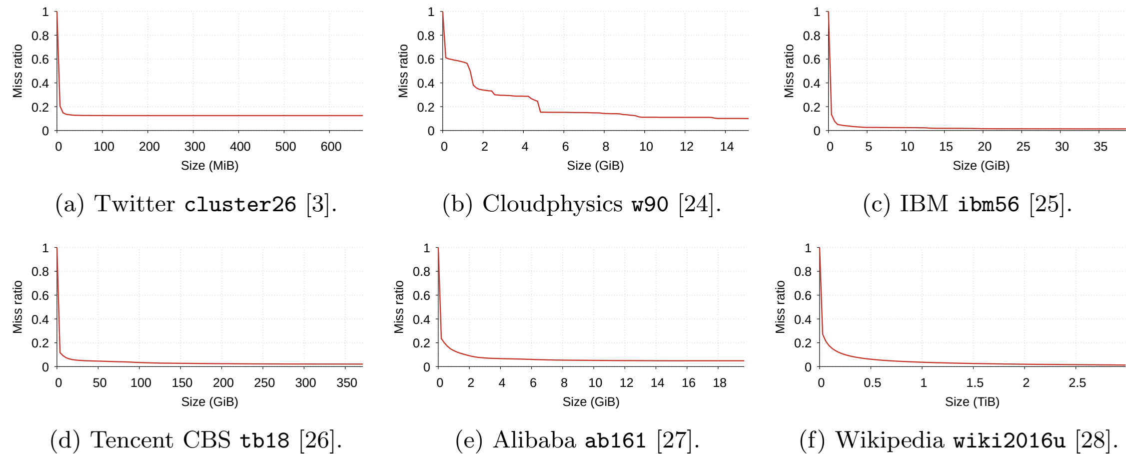
Figure 1.2 shows the α values for all access traces in the datasets considered here. The average and median α values are 1.00 and 1.04, respectively, indicating that an in-memory cache can greatly reduce the load on backend data stores for these workloads.

1.2 Managing in-memory caches in cloud environments

In-memory caches are offered as a service by every major cloud provider [36–39]. In these environments, fleets of cache hosting servers, each hosting multiple containerized cache instances, are managed by a global container manager, such as Kubernetes [40]. Figure 1.3 depicts a typical cloud-hosted in-memory cache environment wherein multiple hosts each host one or more in-memory caches and each cache runs in its own container (e.g., Docker [41]). Local management of each cache (e.g., allocated size and eviction policy) is typically done by the client while global management of these containers (e.g., host placement) is performed in the control plane of a container orchestration system (e.g., Kubernetes [40]). Improving the efficiency of these caches, even slightly, can have dramatic effects on their operating cost and performance. Twitter dedicates petabytes of DRAM and hundreds of thousands of compute cores to hosting these caches [3].

As the number of hosted caches grows, their management on the hosting servers creates a complex optimization problem. Caches with significantly different workloads may not be well suited to reside on the same hosting server. For example, caches with high compute or memory resource demands may negatively affect the performance of other caches on the same hosting server. Conversely, certain caches may be well suited to be placed on the same host. For example, caches with predictable diurnal access patterns may be placed on the same server (if their access times do not overlap).

Relying on clients to select the configuration settings (e.g., allocated size and eviction policy) of their caches can lead to their inefficient operation. In part, because the selection of these configuration settings is non-trivial. Tools such as a *miss ratio curve* (MRC), which plots a cache's miss ratio as a function of its allocated size (§1.3), are vitally important to the accurate understanding of the selection of these settings; however, most clients do not have the expertise to effectively apply these tools to their caches. As a result, many caches are simply configured with the default configuration



(a) Twitter `cluster26` [3]. (b) Cloudphysics `w90` [24]. (c) IBM `ibm56` [25].

(d) Tencent CBS `tb18` [26]. (e) Alibaba `ab161` [27]. (f) Wikipedia `wiki2016u` [28].

Figure 1.4: MRCs for the same access traces as in Figure 1.1.

settings (e.g., the LRU eviction policy) and allocated significantly more memory than required to ensure performance is not degraded.

In a managed environment, cloud providers offering dynamically-scaling caches can use MRCs to better understand the current workload of their clients and make accurate scaling decisions during operation. In doing so, they can not only reduce the amount of unnecessary resources allocated to a client, but they can also reduce the number of servers required to host their clients. For example, Figure 1.4 shows the MRCs for the same 6 access traces as in Figure 1.1. Based on these MRCs, the caches can be sized significantly smaller than that required to store all objects (e.g., a cache of size 5.4GiB for the Tencent CBS `tb18` workload [26] reduces 90% of accesses to the backend data store, while 371.3GiB is required to store all objects). However, MRCs are not static – workloads often go through different phases wherein a cache's size or eviction policy required to achieve its ideal miss ratio changes. The key to being able to offer dynamically-scaling caches that are cost effective is to continuously, in real time, update the MRC of each cache as the accesses occur. For this, highly efficient MRC generation algorithms are crucial. Further, a cache orchestration infrastructure is needed that collects and processes access information from each cache and dynamically reconfigures the caches.

1.3 Miss Ratio Curves (MRCs)

One of the most effective tools used to model the performance of a cache is the *miss ratio curve* (**MRC**), which plots a cache's miss ratio as a function of its allocated size. For an in-memory cache, careful consideration must be made as to how much memory it should be using. If the cache is too small, accesses could observe a higher miss ratio; conversely, if it is too large, the cache is wasting memory resources that could be used for other cache instances (or other applications) running on the same server. The miss ratio curve allows for the accurate understanding of how increasing or decreasing a cache's size will affect the cache's miss ratio.

MRCs have many applications for both clients of caches and the providers that host them. For clients, one of the main difficulties when configuring a cache is navigating performance-cost trade-

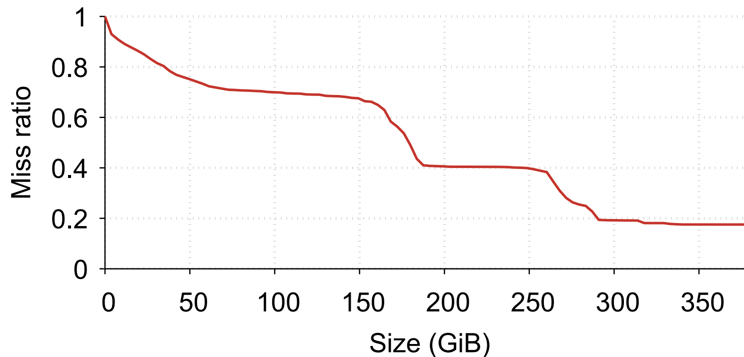


Figure 1.5: MRC generated for the `src1` workload in the MSR dataset [42].

offs when determining a suitable size that achieves an acceptable miss ratio by allowing frequently used data to be cached while not operating an unnecessarily large, and therefore costly, cache. For cache providers, determining the minimal amount of memory required to host a cache is valuable in that it can lead to significant cost savings by reducing unused server space while maintaining the same performance.

Figure 1.5 depicts an example of an MRC. The MRC shows the effect of varying the cache's size from 0GiB to 380GiB on its miss ratio. It can be seen that between roughly 160GiB and 190GiB, the cache notices a sudden drop in its miss ratio. This is referred to as a cliff. This information is especially useful for the application using the cache: if the cache was initially configured with 160GiB of memory, the MRC indicates that increasing the cache size by 30GiB would result in a significant improvement in the miss ratio. For the cloud provider, this MRC is also useful due to the plateau between 190GiB and 250GiB. In this range, the cache's size can be changed without incurring any significant change in the miss ratio. For example, if the cache is initially configured to 250GiB, it can be reduced to 190GiB without the client noticing any significant reduction in performance.

In the early days, the only way to generate an MRC was to run numerous simulations of caches with different sizes, with each simulation generating one point of the curve. A trace of cache accesses for a given workload was used as input to the simulations. In 1970, Mattson et al. were the first to develop an algorithm capable of generating an entire MRC in a single pass, given an access trace [43]. Subsequent to Mattson's seminal work, several algorithms, such as Olken [44] and PARDA [45] were introduced that were more efficient than Mattson's algorithm. While algorithms such as Mattson [43], Olken [44], and PARDA [45] produce 100% accurate MRCs, they are still too compute and memory intensive to be of use for online purposes. To combat this issue, several approximate algorithms have been proposed (such as CounterStacks [46], AET [47], SHARDS [24], Miniature Simulations [48] and Kosmo [8]) which produce approximate MRCs with reasonably low errors, but with significantly improved time and space complexities.

1.4 Eviction policies

Although in-memory caches offer faster data access and lower latencies than backend data stores, they reside on DRAM and therefore have higher operational costs. These caches typically hold a small subset of the data in the backend stores and use an *eviction policy* to select data for removal

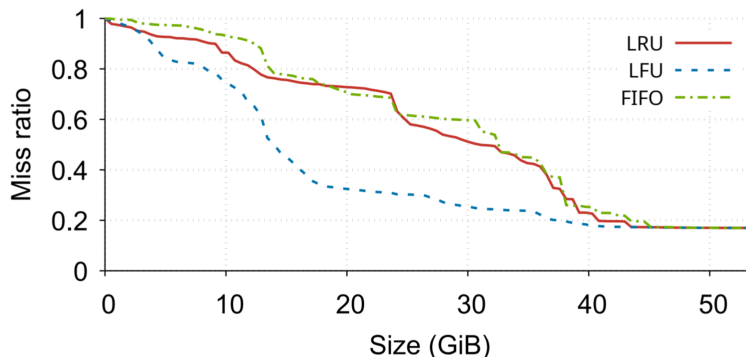


Figure 1.6: MRCs for the LRU, LFU, and FIFO eviction policies for the Cloudphysics `w15` access trace [24].

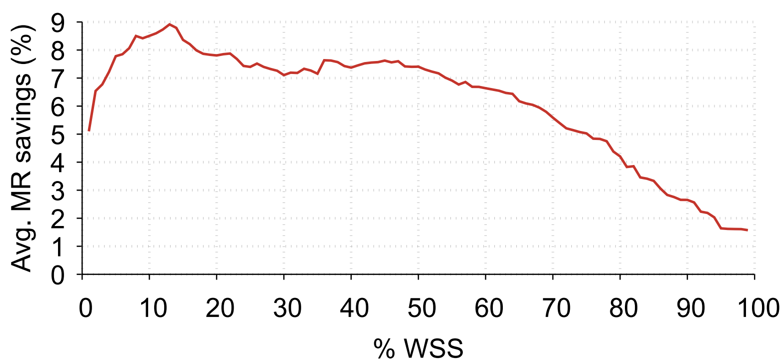


Figure 1.7: Average miss ratio savings versus LRU for all traces in the Cloudphysics dataset [24].

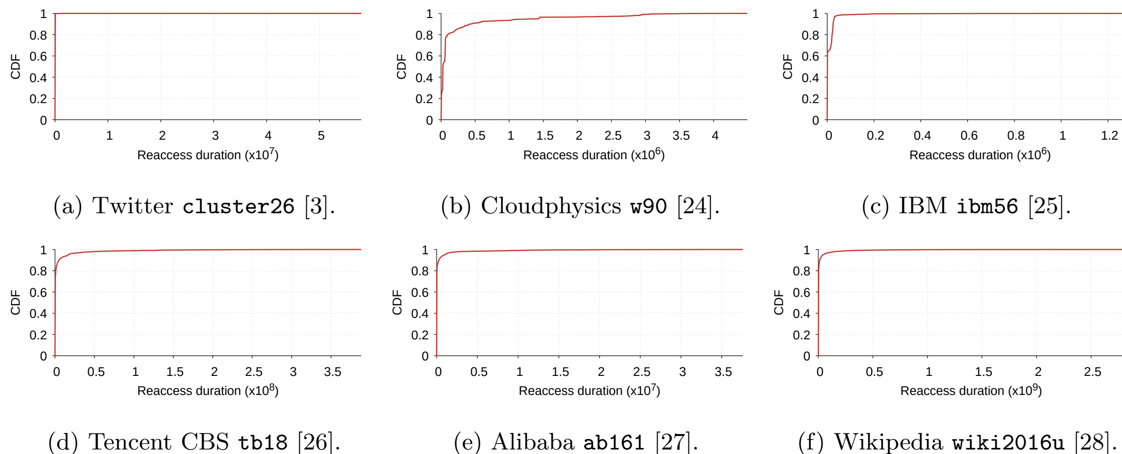
from the cache to make room for new data to be inserted.

The *least recently used* (LRU) eviction policy is the most widely deployed, and often only, eviction policy in modern in-memory caches. However, many other eviction policies have been introduced, such as *least frequently used* (**LFU**), *first-in-first-out* (**FIFO**), *most recently used* (**MRU**), CLOCK, 2Q [20], *least recently/frequently used* (**LRFU**) [49], ARC [50], S3-FIFO [10], LHD [11], or SIEVE [51], which have been shown to outperform LRU under certain workloads [10, 20, 49–51]. In §2.1.2 we describe some popular non-LRU eviction policies. Figure 1.6 shows the MRCs for the LRU, LFU, and FIFO eviction policies for the Cloudphysics `w15` access trace [24]. For cache sizes between 2GiB and 42GiB, the LFU eviction policy significantly outperforms LRU.

Figure 1.7 shows the average reduction in miss ratio a cache can achieve using a non-LRU eviction policy versus LRU for all access traces in the Cloudphysics dataset [24].[1] Here, we vary the cache's configured size from between 1% and 99% of each access trace's *working set size* (**WSS**), where the WSS is the aggregate size of all unique accesses in the trace (§2.2.1). We measured that by using a non-LRU eviction policy, the cache can achieve an average miss ratio reduction of 6.2%, up to a maximum reduction of 77.7%.

Different eviction policies excel under different access patterns and users typically follow a "rule-of-thumb" approach to select which eviction policy to apply to their cache. Consider the accessed objects' frequencies shown in Figure 1.1. In the 6 traces considered, a small number of objects get

[1] We compare LRU with the LFU, FIFO, CLOCK, SIEVE, MRU, 2Q, and S3-FIFO eviction policies.



(a) Twitter `cluster26` [3]. (b) Cloudphysics `w90` [24]. (c) IBM `ibm56` [25].

(d) Tencent CBS `tb18` [26]. (e) Alibaba `ab161` [27]. (f) Wikipedia `wiki2016u` [28].

Figure 1.8: Accessed objects' reaccess duration CDFs (i.e., CDFs of the number of accesses between subsequent accesses to the same object) for the same access traces as in Figure 1.1.

accessed frequently. Here, one would expect the LFU eviction policy to be employed as it excels for workloads that exhibit Zipfian distributions. However, consider Figure 1.8 which shows the reaccess duration CDFs for the same 6 traces, where the *reaccess duration* is the number of accesses between subsequent accesses to the same object. Here, we see these traces also exhibit high data locality, which indicates that the LRU eviction policy is also suitable. This poses an interesting question: how do we accurately select the eviction policy which will provide the best performance?

1.5 Thesis and contributions

Our thesis is that modern cache modeling techniques as well as in-memory cache implementations do not exploit the noticeable benefits of dynamic eviction policy switching and the current state-of-the-art cache orchestration schemes are insufficient for real-world environments. There are several questions regarding the modeling, implementation, and management of in-memory caches not addressed by prior work:

1. How do we accurately and efficiently model the performance of caches with non-LRU eviction policies and compare the performance of various eviction policies for the same workload in real-time? A cache's eviction policy can significantly affect its miss ratio and thus it is not always clear which eviction policy is the optimal choice for a given workload without accurate modeling techniques. Furthermore, we found that the optimal eviction policy changes over time, so modeling techniques that can be performed online are important for optimal cache performance.

2. How do we switch a cache's eviction policy at runtime without incurring downtime and how often should this switch occur? Most modern in-memory caches statically configure their eviction policies, with the exception of Redis which can dynamically switch its eviction policy without incurring downtime, though this is extremely limited.

3. How often should we resize a cache to respect the temporal and spacial localities of objects?

Increasing a cache's size is trivial; however, decreasing its size requires many interesting considerations. For example, if a cache's size is decreased too frequently (e.g., using information gathered from periodically generated MRCs), it can lead to a high miss ratio as objects which have large temporal localities will likely be evicted before their next access.

4. How do we select the optimal configuration settings of caches running on fleets of hosting servers? Considering the cache modeling technique we introduce in this dissertation (Chapter 3) which can model the eviction policy-specific performance of caches, online, and the in-memory cache we introduce (Chapter 4) which can switch its eviction policy at runtime without incurring downtime, how do we make use of these newfound tools to improve the performance of in-memory caches?

This dissertation addresses these questions and makes the following primary contributions:

1. We describe **Kosmo** [8], a novel MRC generation algorithm which generates MRCs for multiple eviction policies simultaneously, online (Chapter 3).
2. We describe **PaperCache** [52], a novel in-memory cache design which supports the instantaneous switching between any eviction policy, online (Chapter 4).
3. We describe **Flux**, a novel real-time multi-host cache orchestration system which leverages genetic algorithms to optimize the allocated size, selected eviction policy, and host assignment of client caches on a fleet of hosting servers (Chapter 5).

1.6 Organization

The remainder of this dissertation is organized as follows:

1. In Chapter 2, we describe relevant prior work, including how in-memory caches are implemented (§2.1), how cache performance is modelled (§2.2), and how the insights gained from these modeling techniques can be applied to improve the performance and aid in the management of in-memory caches (§2.3).
2. In Chapter 3, we describe how the performance of caches with non-LRU eviction policies can be modelled (§3.1), including the downsides and limitations of prior modeling techniques, and our proposed solution which advances the previous state-of-the-art approach (§3.2). The work in this chapter was previously presented in FAST'24 [8].
3. In Chapter 4, we discuss the limitations of modern in-memory caches' support of multiple eviction policies (§4.1) and introduce our solution which addresses these limitations (§4.2). The work in this chapter was previously presented in HotStorage'25 [52].
4. In Chapter 5, we describe prior techniques for managing fleets of in-memory cache hosting servers and their limitations. We illustrate that the periodic re-configuration of a cache's configured size and eviction policy is non-trivial and is not adequately addressed by prior work. We then introduce our solution to these problems (§5.3). The work in this chapter has been submitted to OSDI'26.
5. Finally, in Chapter 6, we conclude and discuss possible future research directions.

Chapter 2

Background

In this chapter, we begin by providing the necessary background to understand modern in-memory caches, including memory allocation (§2.1.1), eviction policies (§2.1.2), time-to-live (**TTL**) parameters (§2.1.3), and wire protocols (§2.1.4). Next, we describe the current state-of-the-art in modeling in-memory caches using working set size (**WSS**) analysis (§2.2.1) and miss ratio curve (**MRC**) generation (§2.2.2). Finally, we describe how modeling techniques can be used to improve the performance of in-memory caches through resource redistribution (§2.3).

2.1 In-memory caches

In-memory caches store frequently accessed, "hot," data in DRAM to reduce data access latency and the load on backend data stores, typically stored on slower, persistent storage mediums. These caches store data in varying-size chunks, referred to as *objects*, where each object consists of a key and value. The memory required to store these objects is acquired using a memory allocator, such as libc-malloc [53] or jemalloc [54]. Although DRAM offers faster data access, it has a higher operational cost and is thus often significantly smaller than the backend persistent stores. In-memory caches can therefore only hold a subset of the total data in the backend stores and employ a policy that determines which objects are removed from a full cache to make room for new objects to be inserted, referred to as the cache's "eviction policy." Caches may also employ time-based eviction wherein objects with associated time-to-live (**TTL**) parameters are evicted after their TTL has expired. As in-memory caches only hold a subset of the total data, an object may not exist in the cache when a client attempts to access it (i.e., the object has been previously selected for eviction using an eviction policy, the object expired due to a TTL, or the object was never placed in the cache). Accesses to objects which do not exist in the cache are referred to as *misses*, while accesses to objects that do exist in the cache are referred to as *hits*. A cache's *miss ratio* R refers to the ratio between the number of misses M incurred by the cache to the total number of accesses T (i.e., $R = M/T$). Ideally, a cache should have as low a miss ratio as possible.

In-memory caches typically run on separate processes than the client applications accessing them. This allows the caches to maintain application state between process restarts and allows shared access between processes (e.g., in load balancing).[1] In-memory caches such as Redis [16] or

[1] There are certain specialized applications wherein an in-memory cache runs on the same process as the client

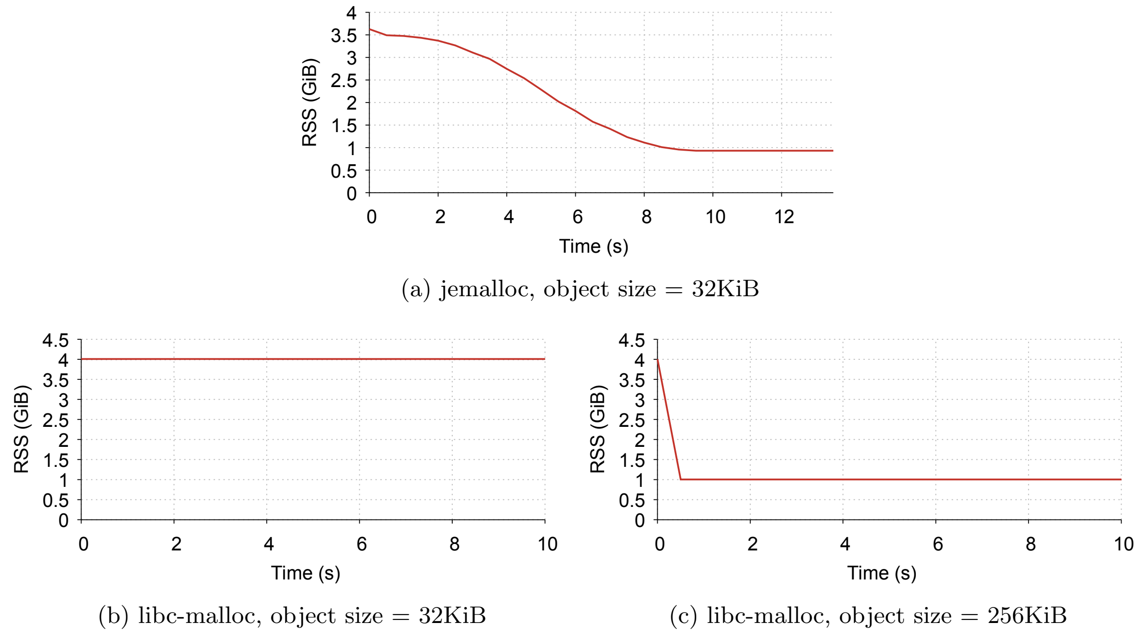


Figure 2.1: RSS over time for a Redis cache configured with libc-malloc originally of size $4GiB$ reduced to $1GiB$.

Memcached [17] use the TCP/IP protocol to facilitate inter-process communication and therefore require a data serialization/deserialization method, sometimes referred to as a *wire protocol*, to send and receive data between the client application.

2.1.1 Memory allocation

In-memory caches use memory allocators configured to acquire and release memory from the host operating system. Increasing a cache's configured size at runtime is trivial – the cache is simply allowed to request more memory from the memory allocator. If a cache's configured size is decreased during operation, as is done by cache orchestration techniques (Chapter 5), and the aggregate size of its stored objects is larger than the newly configured cache size, objects are evicted according to the eviction policy in place (§2.1.2) and its resident set size (**RSS**), which is the total amount of main memory currently held by the cache, must decrease accordingly. Doing this is more involved, and how this is done is largely dependent on the cache's memory allocator. Redis v8.0.1, for example, uses jemalloc by default when compiled on a Linux-based operating system and libc-malloc [53] when compiled on Windows.

jemalloc is configured to release freed memory to the host operating system asynchronously and returns unused pages in two stages, each following a sigmoidal decay curve. First, the unused page is labeled as "dirty" and, within a configurable `dirty_decay_ms` timer (default $= 10s$), the kernel is given the `MADV_FREE` advice, allowing it to free the page under memory pressure if it is not written to again (it is included in the RSS of the process until it is freed). Once marked with `MADV_FREE`, the

application [31], though this is not typical.

page is labeled "muzzy." Next, within a configurable `muzzy_decay_ms` (default = $10s$), the kernel is given the `MADV_DONTNEED` advice, freeing the "muzzy" page and excluding it from the RSS of the process without the need for memory pressure. To examine how this affects a cache's ability to free allocated memory to its host operating system when downsizing, we configured a Redis cache (using jemalloc v5.3) of size 4GiB and filled it to capacity with objects of size 32KiB. We then decreased the cache size to 1GiB (causing it to evict objects consuming 3GiB of memory) and examined the RSS of the process over time. Figure 2.1a shows the results of this simulation. Here, we see that the RSS of the Redis process decreases as expected in a sigmoidal curve, reaching its newly configured size at roughly 10 seconds (the default decay times of jemalloc). We repeated this experiment with objects sized between 512B and 1MiB and noticed similar results.

If running in a Windows environment (which does not support jemalloc), or if manually configured, Redis may use the default system allocator which, in the case of libc-malloc, does not release the memory of objects of size less than the `M_MMAP_THRESHOLD` configuration parameter (default of 128KiB). To examine the effects of this threshold, we repeated the previously described simulation with Redis configured with libc-malloc and objects of size 32KiB and 256KiB. Figure 2.1b and Figure 2.1c show the results of these experiments. Here, we can see that when attempting to free objects of size 32KiB, the freed memory is not reflected in the RSS of the Redis process, though it is with objects of size exceeding `M_MMAP_THRESHOLD` (e.g., 256KiB). Interestingly, we found that 98.8% of accesses in the datasets we examined in our work have a size less than the default value of `M_MMAP_THRESHOLD` which indicates that, in the vast majority of cases, evicting single objects in these caches will not result in a reduction in the cache's RSS.

2.1.2 Eviction policies

When a new object is inserted into a cache, if the cache's configured maximum size is exceeded, an existing object (or multiple existing objects) must be evicted to make room for the new object. The processes in which the cache selects an object to evict is referred to as the *eviction policy.* A cache's eviction policy may have dramatic effects on its miss ratio, and thus, its performance (Chapter 3). Note that in this section (and for the rest of our work), to remain consistent, we refer to the ordered data structure that keeps track of the order in which objects are to be evicted as a "stack", although it is also commonly referred to as a "queue." Table 2.1 shows a summary of popular eviction policies which we describe in this section.

An eviction stack supports three primary operations:

Insert Adds an entry representing an object accessed for the first time (e.g., in the case of LRU, this entry is inserted at the MRU position in the stack).

Delete Removes the entry representing the next object to be evicted from the cache (e.g., in the case of LRU, this is the LRU entry).

Reaccess Updates the entry of an existing object in the cache (e.g., in the case of LRU, this moves the entry to the MRU position in the stack).

The Optimal Eviction Policy (OPT). Belady describes an optimal eviction policy (OPT) which achieves the lowest possible miss ratio wherein the object that is to be accessed furthest in the future is selected for eviction [55]. Although OPT is not practically implementable as it requires

Table 2.1: A summary of popular cache eviction policies. We show the compute complexities of stack insertions and evictions, and "rules-of-thumb" describing when the policy is effective.

Policy	**Complexity**	**Rules-of-thumb**	**Comments**
OPT	N/A	N/A	Ideal, though impossible to implement in practice.
LRU	$O(1)$	High data locality.	Simple to implement; most widely-used policy.
MRU	$O(1)$	Cyclic or strict scanning workloads.	Rarely used in practice due to poor general-case performance.
LFU	$O(1)$	Zipf access patterns.	Higher memory overhead than LRU due to internal data structures.
FIFO	$O(1)$	Large inter-arrival gaps.	Most efficient to implement.
CLOCK	$O(1)$	High data locality.	An efficient approximation of LRU (objects are replaced in-place in the stack).
2Q	$O(1)$	Scanning patterns.	Requires prior selection of configuration parameters.
LRFU	$O(logN)$	Combined high data locality and Zipf access patterns.	Rarely used in practice due to high compute complexities.
S3-FIFO	$O(1)$	Excels under scanning patterns.	Guarantees maximum lifetime of "one-hit-wonder" objects. Requires prior selection of configuration parameters.
SIEVE	$O(1)$	High data locality.	A simple extension of the CLOCK policy to improve performance.

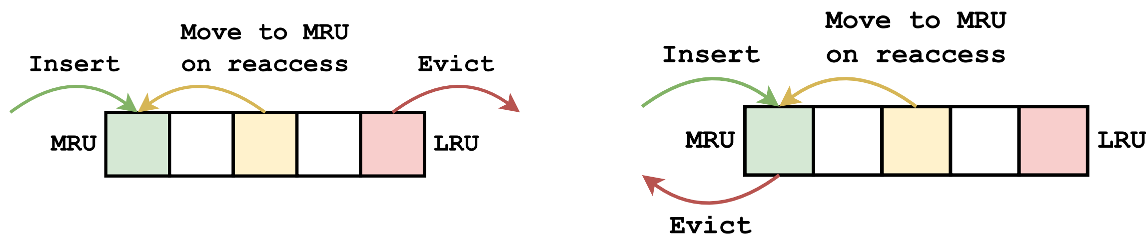


Figure 2.2: LRU (left) and MRU (right) eviction policies.

knowledge of the future, it can be simulated given a cache access trace [43, 56]. OPT is often used as a benchmark to which other eviction policies are compared [57].

Least/Most Recently Used (LRU/MRU). The *Least Recently Used* (**LRU**) and *Most Recently Used* (**MRU**) eviction policies (Figure 2.2) evict the least recently and most recently accessed object in the cache, respectively, to make room for a new object. To implement these policies, a stack of objects sorted by their last access times is used, where the object with the oldest (LRU) or youngest (MRU) time is evicted. LRU is perhaps the most widely-used eviction policy, though MRU has been found to perform better when the workload is cyclical [58].

Least Frequently Used (LFU). The *Least Frequently Used* (**LFU**) eviction policy (Figure 2.3) evicts the least frequently used object in the cache to make room for new objects. A simple method of implementing this policy is to use a stack of objects, ordered firstly by frequency count and secondly by last access time. If an object needs to be evicted, the one with the smallest frequency

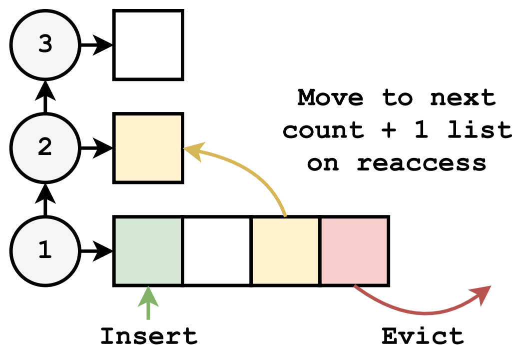


Figure 2.3: LFU eviction policy.

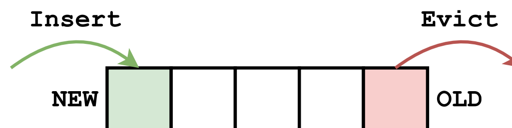


Figure 2.4: FIFO eviction policy.

count, or oldest time on a tie, is selected. LFU caches often outperform LRU caches in workloads that exhibit a Zipfian distribution [30, 59].

Objects in an LFU stack are ordered by access frequency, then by access recently if two objects have the same access frequency. A priority queue is typically used to manage the order of these objects; therefore, updating an LFU stack requires $O(\log n)$ time complexity. A well-known optimization on the LFU stack allows for constant time complexity operations [60]. Here, a linked list is maintained wherein each node corresponds to a frequency count and holds a pointer to a linked list of objects with said frequency count. To increment the frequency count of an object, it is moved from its current list to the list corresponding to the next frequency count. To evict the LFU object, the LRU object from the list corresponding to the lowest frequency count is selected.

First-In-First-Out (FIFO). The *First-In-First-Out* (**FIFO**) eviction policy (Figure 2.4) evicts objects in the same order in which they first entered the cache. It can be implemented using a stack of objects ordered by their entry times where the object with the oldest time is selected for eviction. This policy has been found to perform well on large workloads in which accesses have large inter-arrival gaps, such as those exhibited by scanning behaviors [3, 25]. Of all the eviction policies, it is the most efficient to implement [3] as accesses to existing objects in the stack do not modify the stack (and therefore the stack can be implemented without locking for concurrent accesses).

CLOCK. The *CLOCK* eviction policy (Figure 2.5) is an extension to FIFO that reduces the probability of evicting recently accessed objects [61]. In CLOCK, objects are stored in a circular buffer with a "hand" pointing to the currently selected object for eviction. Each object has an associated *reference* bit which is initialized to 0 when the object first enters the cache, and set to 1 when the object is accessed again in the future. To select an object for eviction to fit a new object in the cache, if the object currently indicated by the hand has a reference bit of 0, it is evicted, the new object is placed at its location, and the hand is moved to the next object. If the reference bit is 1, it is not selected for eviction, and instead the bit is set to 0, the hand is moved to the next object,

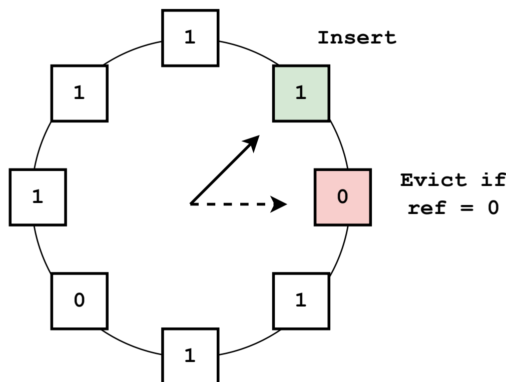


Figure 2.5: CLOCK eviction policy.

and the process repeats until an object is evicted. The CLOCK eviction policy is a low-overhead approximation of LRU as objects are not moved to the head of the stack on each access (similar to FIFO), though objects which have been accessed recently are not evicted (similar to LRU) [61, 62].

CLOCK is an optimization to the *Second Chance* eviction policy which is a different implementation of the same algorithm [62]. In Second Chance, instead of objects being stored in a circular buffer with a hand pointing to the currently selected object for eviction, a standard FIFO stack is used. When the tail object is selected for eviction, if its reference bit is 0, it is evicted. Otherwise, the reference bit is set to 0 and the object is moved to the head of the stack.

2Q. The *2Q* [20] eviction policy (Figure 2.6) 2Q's design is based on the observation that "one-hit-wonders" (i.e., accesses to objects that are only ever accessed once) may unnecessarily cause evictions to objects that should otherwise remain in the cache and should therefore be handle separately from objects which get accessed more than once.[2] 2Q maintains objects in a cache in two separate stacks: one for objects which have been accessed only once (the *A1* stack), and one for objects which have been accessed multiple times (the *Am* stack). Objects in the `A1` stack are evicted in FIFO order, and objects in the `Am` stack are evicted in LRU order. The `A1` stack is further partitioned into two stacks referred to as *A1in* and *A1out*[3] of size *Kin* and *Kout*, respectively, where `Kin` and `Kout` are ratios of the total cache size (the authors note that a `Kin` value of 25% and a `Kout` value of 50% work well in most cases). The `A1in` and `A1out` stacks differentiate themselves in the handling of objects that get accessed a second time; if the accessed object is in the `A1out` stack, it gets promoted to the `Am` stack, while if it is in the `A1in` stack, it does not. Upon the first access to an object, it is placed at the head of the `A1in` stack. If the `A1in` stack is full, the oldest object in the stack is removed and placed at the head of the `A1out` stack. If the `A1out` stack is full, the oldest object is evicted from the cache. If an object which already exists in the `A1out` stack is accessed, it is removed from the `A1out` stack and placed at the head of the `Am` stack. If the `Am` stack is full, an object is evicted using LRU.

Least Recently/Frequently Used (LRFU). The *Least Recently/Frequently Used* (**LRFU**) [49] eviction policy (Figure 2.7) combines objects' recency (i.e., the time since the object was last accessed) and frequency counts to determine which object to evict. Each object has an associated

[2] In fact, in our findings, we found that one-hit-wonders comprise roughly 2% of all accesses and 60.2% of unique accesses.

[3] The `A1out` "ghost" stack holds references to objects, not their values.

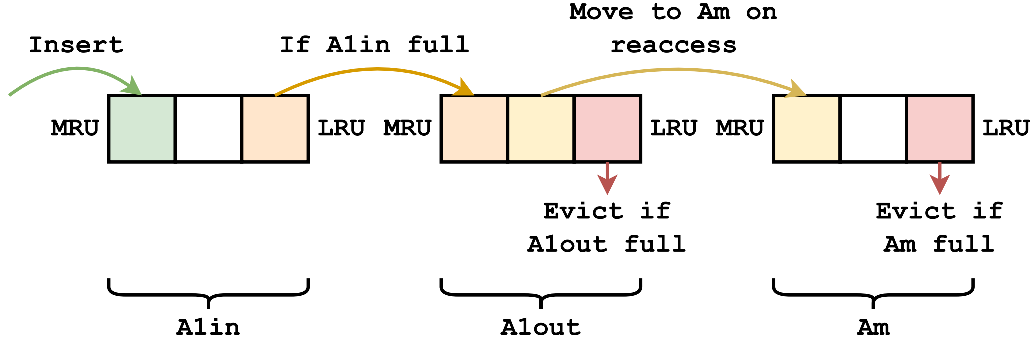


Figure 2.6: 2Q eviction policy.

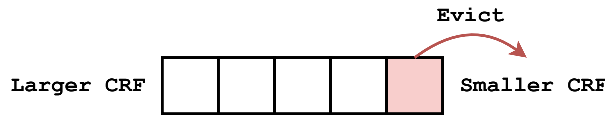


Figure 2.7: LRFU eviction policy.

Combined Recency and Frequency (CRF) value computed as $CRF = \sum_{i=1}^{k} F(t_{now} - t_{access_i})$, where k is the number of times the object has been accessed previously, t_{now} is the current time, t_{access_i} is the time at which the object was accessed the ith time, and $F(x)$ is defined as $F(x) = (\frac{1}{p})^{\lambda * x}$, where p is a value greater than or equal to two, and λ is a value between 0 and 1. Tuning the value of λ allows the cache to behave more similarly to an LFU cache (with λ closer to 0) or an LRU cache (with λ closer to 1). The object with the smallest CRF value is selected for eviction. Although the described CRF formula requires the full history of the object's access times, the authors note that given an object's last access time and last CRF value, one can calculate the updated CRF value without needing the object's access history using $CRF_{updated} = F(0) + F(t_{now} - t_{last_access}) * CRF_{last}$. The LRFU policy has been shown to outperform many other policies for a number of important workloads [49].

S3-FIFO. *S3-FIFO* [10], a recently introduced eviction policy (Figure 2.8), uses three FIFO stacks to hold objects: a small stack S, a main stack M, and a ghost stack G. Each object has an associated frequency count in the range $[0, 3]$, implemented with a 2-bit counter. Upon access to an object that is already in S or M, its frequency count is incremented by 1 (up to a maximum of 3). Upon access to an object not in S or M, the object's frequency count is initialized to 0 and it is placed at the head of M if the object exists in G or at the head of S otherwise. If S is full, the object at the tail is moved to the head of M if its frequency count is > 1 or to the head of G otherwise. If G is full, the object at the tail is evicted. If M is full, the object at the tail is evicted if its frequency count is 0. If the object at the tail of M has a frequency count ≥ 0, its frequency count is decremented by 1 and it is moved to the head of M. This process repeats for the new tail object of M until M is no longer full.

Through experimentation, the authors select 10% of the total cache size as a suitable size for S, and the remaining 90% for M. G is a metadata-only stack that stores up to the same number of entries as currently exist in M. The key benefit of the S3-FIFO eviction policy over other multi-stack policies designed to be resistant to scanning workloads (e.g., 2Q) is that objects which have been

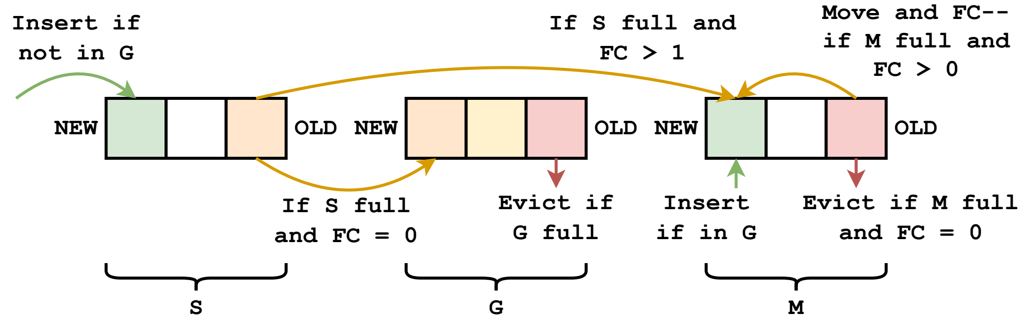


Figure 2.8: S3-FIFO eviction policy.

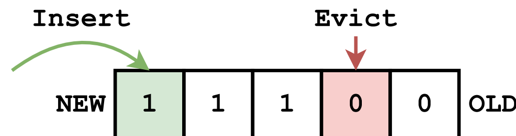


Figure 2.9: SIEVE eviction policy.

accessed more than twice are not immediately evicted from the main stack and are instead reinserted at the beginning of the main stack.

SIEVE. Another recently introduced eviction policy, *SIEVE* [51] (Figure 2.9), works similarly to the CLOCK eviction policy.[4] In SIEVE, objects are stored in a FIFO stack and each object has an associated reference bit. New objects are inserted at the head of the stack with reference bits of 0. Similar to CLOCK, an object's reference bit is set to 1 on access. A "hand" points at the currently selected object for eviction, beginning with the tail of the stack. If the selected object's reference bit is 1, the reference bit is set to 0 and the hand moves to the previous object. Once the hand reaches the head of the stack, it resumes iteration at the tail. Unlike CLOCK, in SIEVE, an object selected for eviction with a reference bit of 1 is not moved to the head of the stack. Therefore, objects are not exclusively evicted from the tail of the stack; they can be evicted from anywhere. SIEVE has been shown to have lower miss ratios than many popular eviction policies, including FIFO, LRU, and CLOCK in some cases [51].

Other eviction policies. Many other eviction policies have been proposed, such as: ARC [50], LIRS [63], LHD [11], SLRU [64], LR-LRU [65], SS-LRU [66], MQ [67], LRB [68], and GL-Cache [69]. However, the eviction policies we described above are among the most frequently cited in literature.

2.1.3 Time to Live (TTL)

When adding a new object to a cache, the client may choose to set an associated *time-to-live* (**TTL**) parameter. An object's TTL is the maximum amount of time the object may exist in the cache, before it is automatically evicted. Caches may handle TTLs using using one of three strategies: (i) proactive eviction, (ii) semi-proactive eviction, or (iii) lazy eviction. A cache employing *proactive* TTL eviction evicts objects with TTLs automatically after they have expired. Here, unlike when

[4] As CLOCK/Second Chance are different implementations of the same eviction policy, we simply refer to both as "CLOCK", as it is the more commonly used name.

evicting objects using an eviction policy, expired cached objects can be removed even while the cache is idle (i.e., no client accesses are occurring) and even if the cache has not exceeded capacity. A cache employing *semi-proactive* TTL eviction proactively evicts expired objects, though also evicts expired objects that have not yet been proactively evicted upon access. Finally, a cache employing *lazy* TTL eviction only evicts expired objects upon access.

A common implementation of semi-proactive TTL eviction in a cache is to store each object's expiry time alongside its value and using a priority queue of `<expiry, *object>` entries, sorted by the expiry times (with the soonest expiry time at the front of the queue). When a client accesses an object that has an associated TTL, the cache retrieves the object and returns it to the client if it has not expired. If it has expired, it is removed from the cache and its associated entry is purged from the TTL priority queue. To remove expired objects which may never be accessed again, the cache periodically checks the priority queue and removes any objects which have expired (this can be performed off the main thread).

Redis [16] does not store TTLs using a priority queue. Instead, it stores object expiries in a hash table of `<key, expiry>` entries. When a client accesses an object, its expire time is first checked in the hash table. If the object has expired, it is removed from the cache and the expiry hash table. As the entries in the hash table are not ordered by expiry time, Redis periodically samples random entries in the hash table to check for expiry. This method of TTL management reduces the memory overhead of storing a priority queue of object expiries at the cost of the increased computational overhead of random sampling.

Memcached [17], similar to Redis, performs semi-proactive TTL evictions wherein expired objects are evicted from the cache upon access and proactively while the cache is idle. However, unlike Redis, Memcached periodically scans all stored objects in the cache to locate and evict expired objects.

2.1.4 Wire protocol

To prevent an in-memory cache from affecting the performance of the application using it, they are typically run on separate processes. Applications therefore need an *inter-process communication* (IPC) method to access their corresponding caches. Most modern in-memory caches, such as Redis [16] and Memcached [17], rely on the TCP/IP protocol for this communication. As such, they require a data serialization and deserializtion protocol[5], typically referred to as a *wire protocol*.

Redis [16] uses the `Redis Serialization Protocol` (RESP) as its wire protocol [70]. Here, data types are identified by the first byte on the wire (e.g., a `+` byte identifies a simple string and a `-` byte identifies an error). Subsequent bytes are then parsed based on the identified data types. In the case of a simple string, the subsequent characters are read until a terminating CRLF (`\r\n`) sequence. For example, the string "OK" (a typical response by the Redis server for various commands) is serialized as `+OK\r\n`.

2.2 Modeling in-memory caches

In this section, we examine performance modeling techniques for in-memory caches. These techniques allow for the accurate understanding of the cost-performance trade-offs, such as cache size

[5]This is also referred to as data marshalling.

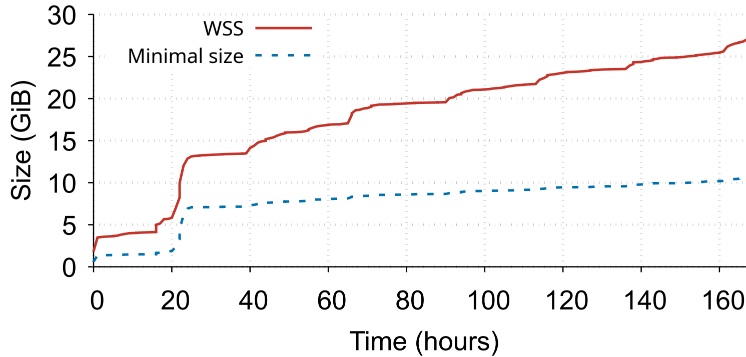


Figure 2.10: WSS and minimal size to achieve ideal miss ratio for the Cloudphysics `w38` trace [24].

versus expected miss ratio, when configuring caches. We first describe *working set size* (**WSS**) analysis (§2.2.1). Next, we discuss exact and approximate miss ratio curve generation techniques (§2.2.2).

2.2.1 Working Set Size (WSS) analysis

A cache's *working set* $W(t, \tau)$ at time t is the set of unique objects accessed during the time interval $(t-\tau, t)$ [71]. A cache's *working set size* (**WSS**) $w(t, \tau)$ is the aggregate size of objects in $W(t, \tau)$. A simple method of WSS calculation is using a hash table storing `<key, size>` entries. On each access, if the access key is not in the hash table, it is inserted and the access size is added to the WSS. If it is in the hash table, the entry's size is updated, and $size_{new} - size_{old}$ is added to the WSS. This method of WSS calculation requires $O(N)$ memory. Several WSS approximation techniques, through the use of SHARDS [24], which performs statistical sampling to reduce the number of accesses processed by the WSS algorithm, or a cardinality estimator (e.g., Cuckoo filters [72]), which computes the observed number of unique accesses without maintaining a hash table of all previously observed unique accesses, can approximate a workload's WSS with significantly less compute and memory overhead.

The WSS provides a crude metric for how to size a cache as it includes "one-hit-wonders" (i.e., objects which are only ever accessed once) in its calculation. Sizing a cache significantly smaller than its WSS will often result in equal performance. Consider Figure 2.10 which shows the WSS, $w(t, \tau)$, and the minimal cache size required to achieve the lowest possible miss ratio for the `w38` trace in the Cloudphysics dataset [24]. Reducing the cache size to the minimal required achieves a 57% memory savings when compared to sizing it to the WSS.[6]

2.2.2 Miss Ratio Curves (MRCs)

Numerous past works have examined methods to generate miss ratio curves (**MRCs**) which plot a cache's miss ratio as a function of its allocated size. In the far past, the only way to generate an MRC was to perform individual simulations of caches of different sizes and computing their resulting miss ratios. However, this is extremely compute and memory inefficient, making it infeasible to perform online. The simulations must either all reside in memory concurrently, or each simulation

[6]This was obtained by comparing the integrals of the two curves.

Time	Access	LRU stack	Stack distance
1	a	a	∞
2	b	b, a	∞
3	b	b, a	1
4	c	c, b, a	∞
5	b	b, c, a	2
6	a	a, b, c	3
7	d	d, a, b, c	∞
8	c	c, d, a, b	4
9	a	a, c, d, b	3
10	a	a, c, d, b	1

Table 2.2: Sample trace with stack distance calculations [43].

must be performed individually, requiring several passes over the access trace. Beginning with Mattson's seminal work in 1970, several exact MRC generation techniques have been proposed which generate 100% accurate MRCs in a single pass. To reduce the compute and memory overheads of these techniques, many approximate techniques have been described which generate reasonably accurate MRCs with significantly less overhead. In this section, we describe several popular exact and approximate MRC generation algorithms.

Exact MRC generation

Mattson's algorithm. Mattson et al. were the first to describe a method capable of constructing a miss ratio curve from a cache's access trace in a single pass [43]. To generate the MRC, Mattson's algorithm maintains an LRU stack containing the accessed objects. Because MRC generation algorithms are modeling caches and do not store the values of the accesses, an "object" in this context simply refers to the referenced key. These objects are ordered by access recency from most recent (at the top of the stack) to least. Upon each access to an object in the access trace, if the object has been accessed before, the *stack distance* of the object is measured by iterating through and counting all objects ahead of it in the stack. The accessed object is then moved to the top of the stack. If an object has not been seen before, the stack distance is said to be infinity, and the object is inserted at the top of the stack. This stack distance identifies the smallest cache size wherein the object exists in the cache, given the access trace up until the point of access. An access to an object at any cache size equal to or larger than the object's stack distance would result in a hit, and a miss otherwise. Mattson's algorithm maintains a histogram to keep track of all previously encountered stack distances. Upon each access, once the stack distance is measured, the histogram counter corresponding to said stack distance is incremented.

Because there could be a large variety of encountered stack distances throughout an access trace, a histogram which maintains a counter for each stack distance could use a significant amount of memory. To reduce this memory usage, in practice, each counter in the histogram corresponds to a range of stack distances. Each of these counters is referred to as a *histogram bucket*.

An example of this process is shown in Table 2.2. Each timestep represents an access request for an object. The "LRU stack" row in the table shows the updated LRU stack after the current access has occurred. The "stack distance" row in the table shows the computed stack distance of the accessed object. Table 2.3 depicts the histogram of stack distances for the sample trace of Table 2.2.

Stack distance	Count
1	2
2	1
3	2
4	1
∞	4

Table 2.3: Histogram of stack distances for the sample trace of Table 2.2.

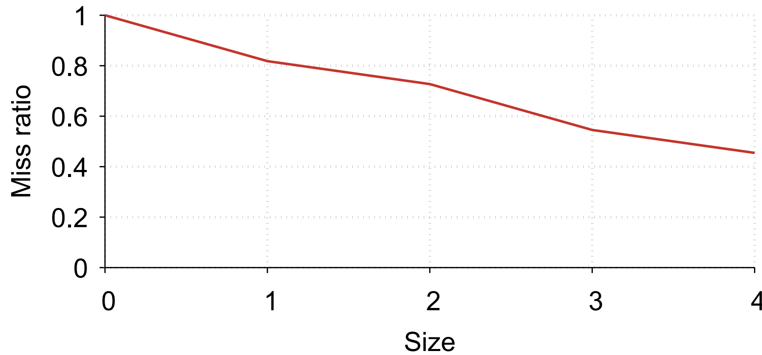


Figure 2.11: The MRC obtained by processing the sample trace of Table 2.2.

Once the entire trace has been processed, the miss ratio at any given cache size, $R(s)$, can be found using the following Equation 2.1, where s is the cache size, $d(s)$ is the stack distance histogram counter at size s, and N is the total number of accesses ($R(s)$ is effectively the inverse cumulative distribution function (**CDF**) of the stack distance histogram).

$$R(s) = 1 - \frac{\sum_{i=0}^{s} d(s)}{N} \tag{2.1}$$

The resulting MRC of the example provided in Table 2.2 is shown in Figure 2.11.

Although Mattson's algorithm is able to produce 100% accurate MRCs for LRU caches, it is compute and memory intensive. The stack takes $O(M)$ space for a trace with accesses to M unique objects. The computation of the stack distance for a given object takes $O(M)$ time, so the algorithm requires $O(MN)$ time for a trace with N accesses. To put this into perspective, the `cluster52` access trace in the Twitter dataset [3] contains 11,938,611,526 accesses to 135,845,907 unique objects (i.e., M = 135,845,907 and N = 11,938,611,526).

Olken. Olken describes a method of improving the efficiency of Mattson's stack distance calculation, required for each access in the trace, from $O(M)$ to $O(log(M))$ [44]. Whereas Mattson uses a stack to store the accessed objects (i.e., the keys in the access trace) and measures the stack distance of any given object by iterating from the top of the stack, Olken stores this data using a balanced binary tree sorted using the objects' timestamps of when they were last accessed. Each node in the tree maintains an object, a timestamp, and the size of its subtrees to allow for the calculation of stack distances without needing to traverse all nodes in the tree. With Olken, the stack distance of any given object can efficiently be measured as the total number of nodes in the tree that have a larger timestamp.

Similar to Mattson's algorithm, on each access, if the object has not been accessed before, its

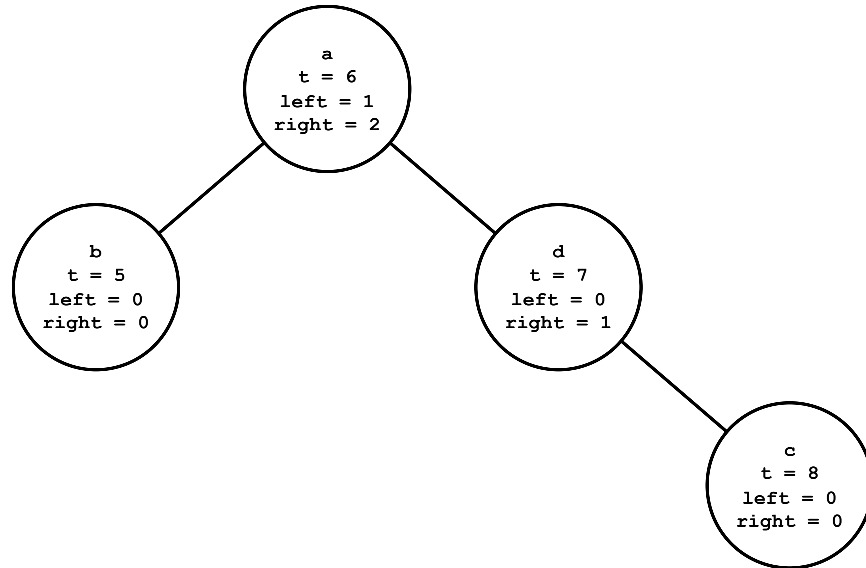


Figure 2.12: Sample trace Olken tree at $t = 8$.

stack distance is said to be infinity. The object is then inserted as a node in the tree (the object will be the right-most node in the tree as it will have the largest timestamp) and the tree is rebalanced. A reference to the created node is then inserted into a hash table (using the object as an index). If the object has been accessed before, its stack distance is measured by counting the number of nodes in the tree with a larger timestamp than that of the object's last access (i.e., the timestamp stored in the object's node, found using the object's entry in the hash table). To perform this count, a binary search traversal begins at the root of the tree. If the node being traversed has a smaller timestamp than the current object's last accessed timestamp, the traversal continues on the right subtree. If the node has a larger timestamp than that of the accessed object, the size of the node's right subtree plus the node itself is added to the stack distance and the traversal continues on the left subtree. This continues until the node containing the object being accessed is reached. At this point, the size of the node's right subtree is added to the stack distance plus one for the node itself. Once this is complete, the stack distance has been established. The timestamp of the object stored in the node is updated to the current time (i.e., the current access's timestamp), and the node is removed and re-inserted in the tree and thus moved to its correct position in the tree.

Figure 2.12 shows an example of such a tree. This figure depicts the balanced tree at time step 8 of the sample trace in Table 2.2. To calculate the stack distance of a being accessed at time step 9, the number of nodes in the tree with a larger timestamp than a is counted, plus one (for the a node itself). The traversal begins at the root of the tree (i.e., the a node). As this node contains the object being accessed, and the node is the root, the stack distance is the size of the node's right subtree plus the node itself. The right subtree has a size of 2, which results in a stack distance of 3.

Although Olken's algorithm is able to reduce the computational complexity of processing an access trace to $O(Nlog(M))$, it still must store all unique objects as nodes in its tree; therefore, its space complexity is $O(M)$.

Parda. Niu et al. describe a method of parallelising the processing of an access trace to generate an MRC [45]. In this algorithm, the input access trace with N accesses is divided into S sections, where

the first section is the first N/S accesses, the second section is the next N/S accesses, and so on. Each section is then processed in parallel using Olken, resulting in S histograms each representing the observed stack distances in each of their respective sections. These S histograms have to be combined. We describe how this is done further below. With this approach, using 64 threads, the authors were able to achieve a speedup of between 13 and 50 times relative to the traditional tree-based stack distance algorithm.

PARDA makes use of the property of stack distance calculation which states that the stack distance of any given object being accessed at time t whose last access was at time t' is independent of all accesses that occurred before t'. Because of this property, any non-infinite stack distance that is calculated using a section of the trace is a correct stack distance. However, an object being accessed for the first time in one section may have been accessed at an earlier time in a previous section. Therefore, the number of infinite stack distances observed in each thread may not be accurate. To correct for this, each thread maintains a time-ordered queue of objects which resulted in infinite stack distances called the *local infinities*. Once a thread completes the processing of its section of the trace, it sends its histogram and local infinities to the thread processing the previous section. When a thread receives the histogram and local infinities from another thread, it first adds any non-infinite stack distances to its own histogram. It then iterates over the received local infinities and appends each key it has not observed in its section of the trace to its own local infinities queue. A section can identify whether it has observed a key being accessed because the first access to that key has a stack distance of infinity and hence will be recorded in its infinities queue. For any key that it has observed in its section of the trace, it computes its stack distance by counting the number of objects in its distance tree that have a larger timestamp than the object and adding this value to the number of objects ahead of this object in the received local infinities queue. This stack distance is then added to the thread's histogram. The updated histogram and local infinities queue are then sent to the thread processing the previous section. Once this process has reached the thread processing the first section and this thread updates the histogram, the resulting histogram is complete and can be used to generate the MRC using the same method described in Mattson's algorithm.

Although PARDA provides a speedup when processing an access trace to generate an MRC, its key disadvantage is that it must have the entire access trace to be able to divide and process each section in parallel. This indicates that the algorithm is not suitable for online MRC generation.

Approximate MRC generation

Although several past works have developed methods to generate exact MRCs, they are too compute and memory intensive to use with large, modern workloads. Furthermore, to use MRCs in the dynamic adjustment of a cache's configuration settings, allowing it to adapt to changing workloads, these MRCs must be generated efficiently, online. Several approximate MRC generation algorithms have therefore been proposed which significantly reduce the resource overheads of generating these MRCs.

SHARDS. Waldspurger et al. describe an algorithm called *SHARDS* which works in conjunction with an exact MRC generation algorithm (e.g., Olken). SHARDS generates MRCs using only a sampled subset of the total trace [24]. This significantly improves the efficiency with which an MRC can be generated, and while approximate, the resulting MRC typically has reasonably low error. SHARDS can be implemented as either fixed-rate or fixed-size. The fixed-size variation provides

constant memory usage. Both variations of the algorithm depend on the same sampling method described in the following steps (outlined in Figure 2.13):

1. Compute $T_i = hash(K) \bmod P$, where K is the key of the accessed object and P is a large static number. If T_i is smaller than a selected threshold T, sample the access, otherwise discard it. The sampling rate, R, is defined as $R = T/P$.

2. Once the stack distance of the object has been computed, it must be scaled to account for the objects that were not sampled. To do this, the stack distance is divided by R. This scaled stack distance is then used to update the histogram.

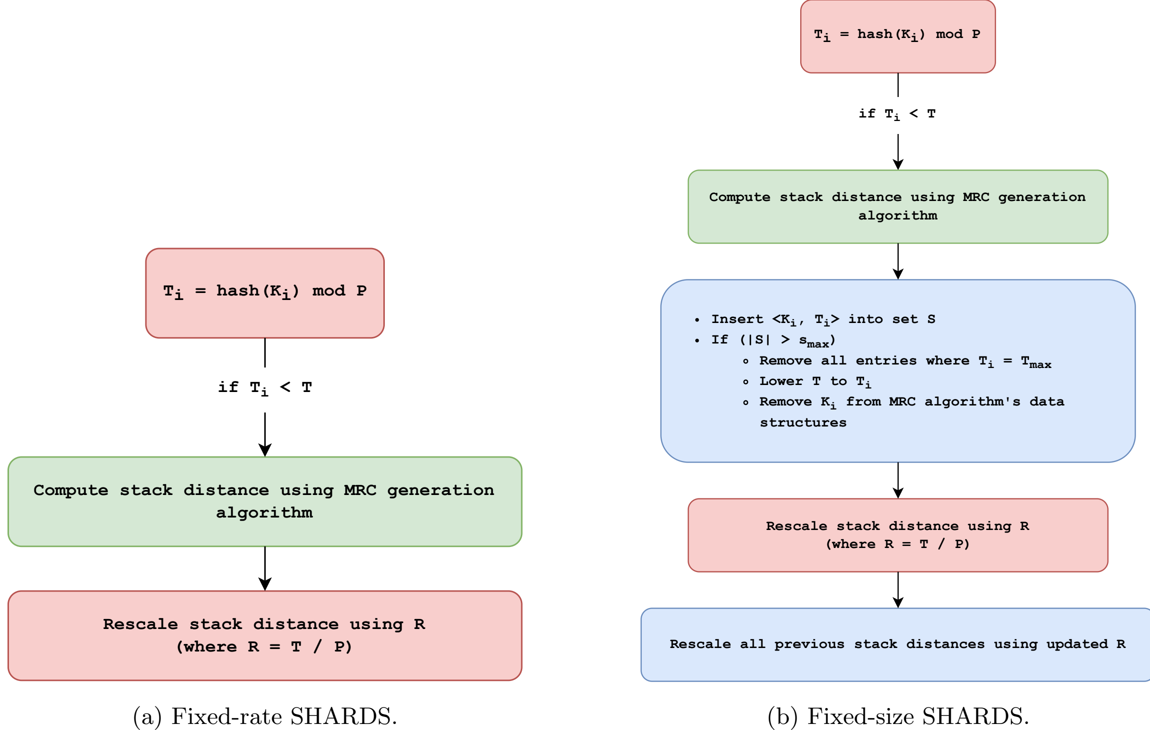
In the fixed-rate implementation, the algorithm samples the trace using a specific rate, R, choosing values for P and T accordingly. The authors of the paper found that an R value of 0.1% results in reasonably accurate MRCs [24]. One attractive feature of SHARDS is that once an object is sampled, it will continue to be sampled unless R changes.

The fixed-size implementation of the algorithm limits the size of the set of objects that exist in the MRC algorithm's internal data structures at any given time to a constant, S_{max}. To accomplish this, a step needs to be introduced between steps 1 and 2 described above. In this step, the algorithm updates a set, S, that contains all objects currently being stored in the MRC algorithm's data structures as well as their corresponding T_i values computed in step 1. When an object is accessed and is sampled, if it does not exist in the set, it is added. If the set's size then exceeds the configured threshold S_{max}, the entry with the largest T_i value (T_{max}) is removed from S and from the MRC algorithm's internal structures (e.g., Olken's distance tree). Once this is complete, the global threshold, T, is lowered to the threshold of the removed entry, T_{max}.

Because T decreases over time as the entries in the set of objects are removed in this variant of SHARDS, a step also needs to be introduced after step 2. Each time the threshold is lowered, all counter values in the histogram have to be rescaled using the new threshold. These steps are outlined by the blue stages in Figure 2.13.

The authors of SHARDS noticed that the expected number of sampled accesses, $E[N_S]$, was often not equal to the measured number, N_S. To correct for this, after the access trace has been run through the algorithm, they add the difference between $E[N_S]$ and N_S to the first histogram bucket (i.e., the histogram counter for the smallest stack distance). Because the expected number of sampled accesses in SHARDS is the total number of accesses, N, multiplied by the sampling rate, R (i.e., $E[N_S] = N * R$), the difference can be calculated as $N * R - N_S$. By applying this correction, the authors achieved significant improvements in the error created by the sampling algorithm. Interestingly, the authors proposed this adjustment in the evaluation section of the paper that introduced SHARDS and tested it on just one trace. However, our evaluation on a large set of traces (14 traces in the MSR dataset [42] and 16 traces in the Twitter dataset [3]) indicates that this technique is very effective.

Interestingly, we also found SHARDS to be particularly useful for WSS estimation. WSS calculation can incur significant memory overhead if the access trace has a large number of unique objects. We applied SHARDS sampling to the WSS calculation to significantly reduce its overhead. Figure 2.14 shows the distribution of *mean absolute errors* (**MAEs**), calculated as the mean absolute difference between each trace's actual WSS and the WSS computed using SHARDS, when using fixed-rate SHARDS sampling for WSS calculations with varying sampling rates. For sampling rates



(a) Fixed-rate SHARDS.

(b) Fixed-size SHARDS.

Figure 2.13: SHARDS algorithm (red) with fixed-size variant steps (blue).

of 10%, 1%, 0.1%, 0.01%, and 0.001%, we measured average MAEs of 0.3%, 1.1%, 3.4%, 9.2%, and 28.1%, respectively (the latter two errors would be considered unacceptable for actual use).

Miniature Simulations. Due to their inherent reliance on an LRU stack, all previously described MRC generation algorithms can only generate MRCs for caches employing the LRU eviction policy (though they can be adapted to generate MRCs for *certain* other eviction policies, which we describe in further detail in §3.1.1). Waldspurger et al. describe a method of generating MRCs called *Miniature Simulations* (which we will refer to as "MiniSim"), capable of modeling any eviction policy. MiniSim independently simulates caches at varying sizes to obtain the resulting MRC and uses SHARDS sampling to reduce compute and space overheads [48].

To generate an MRC using MiniSim, a maximum simulated cache size C_{max} is selected. A number of simulated caches (N_C) are then instantiated, each simulating a cache size between C_{max}/N_C and C_{max}. In practice, N_C is often set to 100. Upon each access in a trace, if the access is sampled by SHARDS, the access is processed by each simulated cache. When the trace is complete, the miss ratios of the simulated caches and each simulated cache's respective size are used to form an MRC.

MiniSim utilizes the sampling method proposed by SHARDS to operate on a small subset of the total trace to improve runtime performance, making it an approximate MRC generation algorithm. Before processing accesses, the size of each simulated cache is scaled by the SHARDS sampling rate R (i.e., $C_{scaled} = RC$, where R is the SHARDS sampling rate and C is the simulated cache's size).

The authors of the original paper only describe MiniSim configured with the fixed-rate SHARDS variant. Unfortunately, selecting a sampling rate without prior knowledge of the access trace is difficult, limiting MiniSim's applicability for use in online environments. For example, if a sampling

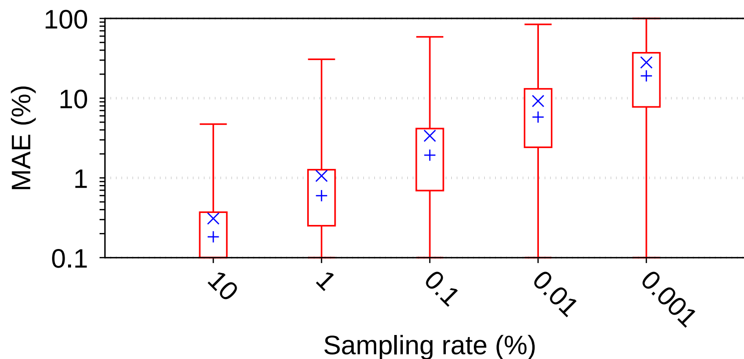


Figure 2.14: WSS approximation with fixed-rate SHARDS for varying sampling rates. Note the logarithmic scale of the y-axis.

rate of 0.1% is selected (as is recommended in the original paper [24]), and an MRC is being generated for an access trace with 10,000 accesses, only roughly 10 accesses will be sampled, resulting in an inaccurate MRC. However, in §3.3.1, we describe how we extend MiniSim for use with the fixed-size SHARDS variant, making it far more useful for MRC generation.

Other approximate MRC generation algorithms

Several other approximate MRC generation algorithms have been introduced, such as: Counter-Stacks [46], RAR-CM [73], AET [47], Statstack [74], RapidMRC [75], LAMA [76], and MIMIR [77]. However, they are not relevant to this dissertation.

2.3 Utilizing cache models for resource redistribution

Several past works have examined optimizing the allocation of resources for multi-cache hosting servers [73, 78–83]. In this section, we describe two of these works, OSCA [73] and mPart [79], as they, like Flux (Chapter 5), use MRCs to guide their optimization decisions. All past works only optimize caches running on a single host.

OSCA. OSCA constructs MRCs for the LRU eviction policy using an MRC generation algorithm called *RAR-CM*. Once every 12 hours, the OSCA algorithm redistributes the available host memory to its caches. The size of each cache is determined by minimizing the aggregate number of misses amongst all the caches on the host (i.e., minimizing the value of $\sum MRC_i(S_i) * NR_i$, where MRC_i, S_i, and NR_i is the MRC, the allocated size, and the number of accesses of cache i, respectively). This calculation is done using exhaustive search and applies dynamic programming to reduce its computational overhead.

mPart. mPart constructs an online MRC for the LRU eviction policy for each cache using the *Average Eviction Time* (AET) algorithm [47]. At configurable time intervals, referred to as *arbitrations*, the memory allocated to each cache is adjusted using the same cost function used in OSCA; however, mPart also allows for the manual configuration of a lower bound L_i and upper bound H_i which determines the minimum and maximum amount of memory a cache must have after an arbitration, respectively.

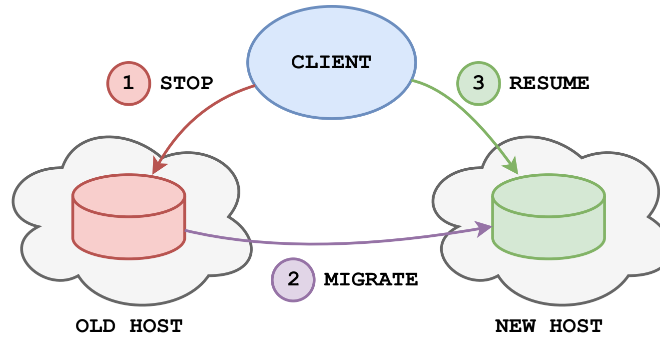


Figure 2.15: Stop-and-copy migration.

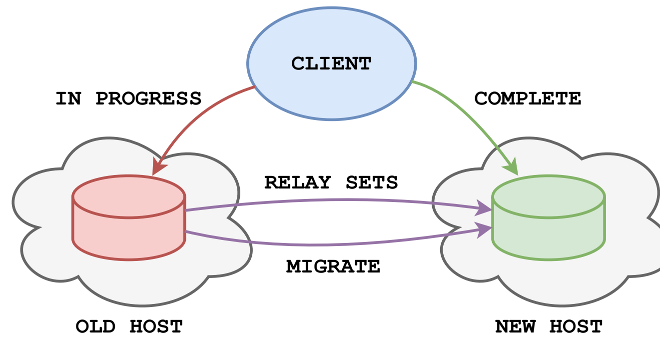


Figure 2.16: Source-based migration.

2.3.1 Cache migration

When managing multiple caches on a fleet of hosts it may be beneficial to migrate select caches to different hosts (e.g., to consolidate the caches onto fewer hosts, or to move highly accessed caches onto different hosts). Several past works have examined effective methods of migrating caches between hosting servers [16, 84–90]. These techniques are organized into four categories: (i) *stop-and-copy* [16, 84], (ii) *source-based* [85–87], (iii) *destination-based* [88], and (iv) *hybrid* [89, 90].

Stop-and-copy The *stop-and-copy* migration protocol (Figure 2.15) is the most basic form of migration. It involves stopping all accesses to the migrating cache, performing the migration, then resuming the accesses to the cache on the new host. This technique involves downtime

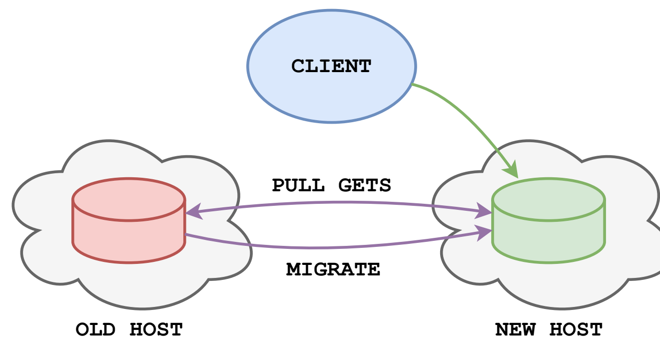


Figure 2.17: Destination-based migration.

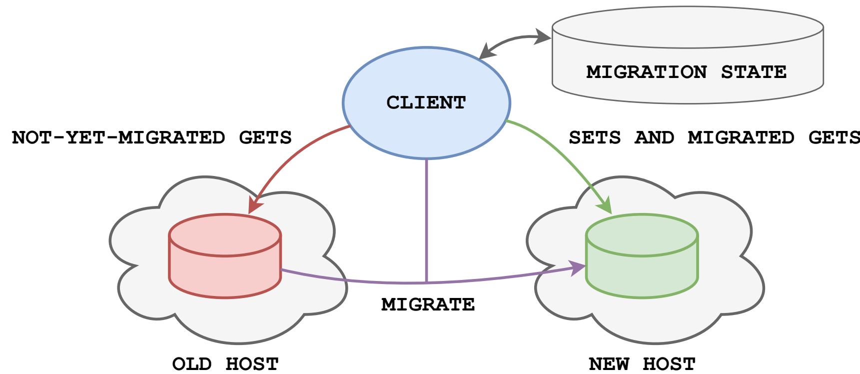


Figure 2.18: Hybrid migration.

during the migration. Redis [16] implements the stop-and-copy migration technique on a per-key basis (using the `MIGRATE` command), allowing the client to migrate individual objects from one cache to another (e.g., in the event that one object is frequently accessed, migrating just this object to a different host can significantly reduce a cache's load).

Source-based *Source-based* migration (Figure 2.16) is a method of migration in which the source (i.e., old) host begins migrating the cache to the destination (i.e., new) host asynchronously while serving accesses from the client. During migration, the client may update objects in the old host which have already been migrated to the new host, creating "dirty" data in the host. The old host is responsible for relaying any `SET`s incurred during migration to the new host to update this dirty data. Source-based migration has low access latency during migration, but a long migration time (as `SET`s occurring during the migration may accumulate). Source-based migration also results in a brief downtime as the client's accesses to the old host must be paused to transfer the final dirty data to the new host. This prevents the migration from proceeding indefinitely if the client consistently writes to the cache during migration.

Destination-based *Destination-based* migration (Figure 2.17) relies on the destination (i.e., new) host to facilitate the migration. In this protocol, the new host asynchronously migrates the cache from the source (i.e., old) host and all clients immediately route all accesses to the new host. If the client attempts a `GET` access on not-yet-migrated data, the new host pulls this data from the old host and serves it to the cache. This migration technique is typically faster than source-based migration as `SET`s during the migration do not create dirty data on the new host. However, in destination-based migration, `GET`s on not-yet-migrated data have higher access latencies.

Hybrid In *Hybrid* migration (Figure 2.18), the client sends all `GET`s and `SET`s to migrated objects to the new host. `GET`s to not-yet-migrated objects are sent to both the new and old hosts and the client is responsible for achieve data consistency. The client must therefore maintain the migration progress internally (e.g., in an intermediate key-value store [89]). Hybrid techniques have lower latencies than destination-based approaches as `GET`s to not-yet-migrated objects do not need to be retrieved from the old host, and faster migration times than source-based approaches as `SET`s during the migration do not create dirty data in the new host. However, this comes at the cost of increased complexity for the client.

Chapter 3

Kosmo: Efficient Modeling of Non-LRU Eviction Policies

The choice of eviction policy is an important factor in configuring in-memory caches. While most in-memory caches default to the *Least Recently Used* (LRU) eviction policy, it has been shown that under certain workloads, caches operate more efficiently using non-LRU eviction policies [3, 25, 30, 49, 59]. For example, the LFU eviction policy can sometimes achieve a roughly 14% reduction in miss ratio when allocated the same cache size and under the same workload [49]. Eviction policies such as *First-In-First-Out* (FIFO) also have lower computational and memory overheads than other policies, such as LRU [3].

To optimize a cache configuration in terms of both size and eviction policy, it is necessary to generate an MRC for each eviction policy under consideration. However, a key limitation of almost all existing MRC-generation algorithms is that they only model caches operating with an eviction policy that satisfies the inclusion property (e.g., LRU); as such, they do not support eviction policies such as LFU, FIFO, 2Q, *Least Recently/Frequently Used* (LRFU), or *Most Recently Used* (MRU). The only known MRC generation algorithm capable of modeling a wide array of non-LRU caches with reasonable computational efficiency is Miniature Simulations (*MiniSim*) [48]. It runs individual simulations of caches of different sizes and makes use of the SHARDS [24] sampling algorithm to improve its runtime performance. However, MiniSim has several serious drawbacks, the most notable being its high memory usage. MiniSim effectively simulates independent caches of varying sizes, often causing duplicate data to be stored in the internal structures of many of these simulated caches. We found through experimentation with numerous workloads that MiniSim configured with

Table 3.1: Average and maximum memory usage of MiniSim when generating MRCs for four eviction policies (configured with 100 simulated caches and a SHARDS sampling rate of 0.1%) when measured across 52 publicly-accessible real-world cache access traces.

Eviction policy	**Average memory usage**	**Maximum memory usage**
LFU	113MiB	3.1GiB
FIFO	57MiB	1.7GiB
2Q	40MiB	396MiB
LRFU	31MiB	597MiB

100 simulated caches can consume up to 3.1GiB of memory to generate a single MRC. Table 3.1 shows the average and maximum memory usage of MiniSim configured with 100 simulated caches and a SHARDS sampling rate of 0.1% when generating MRCs for the LFU, FIFO, 2Q, and LRFU eviction policies for 52 publicly-accessible real-world cache access traces. Further, to generate MRCs for multiple eviction policies simultaneously, these memory requirements are compounded. With these memory requirements, MiniSim will likely consume substantial memory, and hence may even interfere with the cache itself.

This chapter describes **Kosmo** [8], a new MRC generation algorithm that supports the simultaneous generation of MRCs for a variety of eviction policies that, on average, uses significantly less memory than MiniSim, making it better suited for online MRC generation. Kosmo uses a novel method of calculating reuse distances through the introduction of *eviction maps*. We show how Kosmo can be used to simultaneously generate MRCs for six eviction policies: LFU, FIFO, 2Q, LRFU, LRU, and MRU. Notably, LFU, FIFO, 2Q, LRFU, and MRU do not adhere to the inclusion property [8].

We evaluate Kosmo using a total of 52 publicly-available workloads and measure memory usage, throughput, and accuracy for LFU and FIFO, and 33 workloads for the 2Q and LRFU eviction policies. Kosmo requires an average of 3.6 times less memory, and up to 36 times less memory, than MiniSim across all eviction policies. Kosmo has an average throughput 1.3 times that of MiniSim across all eviction policies. Finally, Kosmo, which is also an approximate generation algorithm, produces MRCs with comparable accuracy to those generated by MiniSim.

Contributions

The contributions made by our work are:

- We introduce Kosmo, a novel method of simultaneously generating MRCs for a variety of eviction policies.
- We introduce a method of reconstructing the stacks of caches of varying sizes using a single copy of the cached data through our novel data structure, *eviction maps*.
- We describe how to apply eviction maps to the LFU, FIFO, 2Q, LRFU, LRU, and MRU eviction policies, allowing Kosmo to generate MRCs for these policies.
- We evaluate the performance of both Kosmo and MiniSim and show that Kosmo achieves an average memory reduction of a factor of 3.6 and up to a factor of 36.
- We examine to what degree different eviction policies violate the inclusion property.

Limitations

This work has several limitations, however. First, we only describe Kosmo for six sample eviction policies. Although we know Kosmo supports additional eviction policies beyond those described in this work, identifying which classes of eviction policies Kosmo is able to support remains an open problem. Second, the MRCs Kosmo generates are monotonically decreasing which could increase the error for eviction policies which display significant non-monotonic behavior. Finally, we recognize that the performance of MiniSim is affected by the performance of the underlying cache it

is simulating, and we make best efforts to ensure our implementations of the simulated caches are reasonably efficient.

3.1 Background

In this section, we discuss relevant prior work. Namely, we discuss the inclusion property and its importance in MRC generation. Similar to Mattson [43], Kosmo maintains a stack distance histogram with which it generates MRCs. To evaluate Kosmo, we compare it to the current state-of-the-art algorithm, MiniSim [48], which both use the SHARDS [24] sampling algorithm to reduce their compute and memory overheads.[1] These algorithms are described in §2.2.

3.1.1 Inclusion property

An important characteristic of an eviction policy is whether or not it adheres to the *inclusion property*. This property states that all objects that exist in a cache of size S at a given time, also exist in any cache of size $S' > S$, when given the same access trace [48, 77]. An extension of this property is the *strict inclusion property* which adds the further constraint that all common objects in any two caches (with the same access trace) must be in the same order in the caches' internal data structures (i.e., stacks).

An MRC generation algorithm that models an eviction policy adhering to the strict inclusion property is often referred to as a "stack algorithm," and can be implemented similarly to Mattson [43], described earlier in §2.2. If the eviction policy does not adhere to the strict inclusion property, a dedicated algorithm for the eviction policy or MiniSim must be used.

In the literature, the LFU eviction policy is often referred to as a stack algorithm, which implies it can be modelled using an algorithm similar to Mattson [24, 43, 77]. However, this is only the case for so called *ideal LFU caches*, in which the cache maintains frequency counters for all objects that were ever accessed; if an object is evicted and accessed again in the future, its counter persists and is further incremented in ideal LFU caches. In practice, LFU caches do not maintain the counters of objects that have been evicted [92], as maintaining these counters would entail significant memory overhead. These *practical LFU cache* implementations remove the counter of any object being evicted from the stack, and if a previously evicted object is accessed again, a new counter is instantiated and initialized to one.

Practical LFU caches do not adhere to the inclusion property, in contrast to ideal LFU caches. Table 3.2 shows this for a simple access trace. Here, the objects and their associated counts in two practical LFU caches of size 3 and 4 are shown. At each time step, the frequency counter (shown alongside each object in brackets) of the accessed object is incremented by one, or initialized to one if the object is being accessed for the first time. The stack of each cache is ordered from last to be evicted from the cache (i.e., the object with the largest frequency counter, or most recently accessed if two objects have the same frequency counter, on the left) to next to be evicted from the cache. At time 9, it is evident that even though the two caches were provided with the same access trace,

[1] We note that recent advancements have been made to improve the accuracy of SHARDS when sampling access traces with heterogeneous object sizes [91]. These improvements can also be applied to the sampling technique used by Kosmo and MiniSim.

Table 3.2: Sample trace for LFU caches of sizes 3 and 4 demonstrating a violation of the inclusion property. The values in the brackets next to each object in the caches are the frequency counts. At time 9, object "e" exists in the cache of size 3, but not in the cache of size 4.

Time	Access	LFU cache size 3	LFU cache size 4
1	a	a(1)	a(1)
2	b	b(1), a(1)	b(1), a(1)
3	c	c(1), b(1), a(1)	c(1), b(1), a(1)
4	d	d(1), c(1), b(1)	d(1), c(1), b(1), a(1)
5	a	a(1), d(1), c(1)	a(2), d(1), c(1), b(1)
6	d	d(2), a(1), c(1)	d(2), a(2), c(1), b(1)
7	b	d(2), b(1), a(1)	b(2), d(2), a(2), c(1)
8	e	d(2), e(1), b(1)	b(2), d(2), a(2), e(1)
9	f	d(2), e(1), f(1)	b(2), d(2), a(2), f(1)

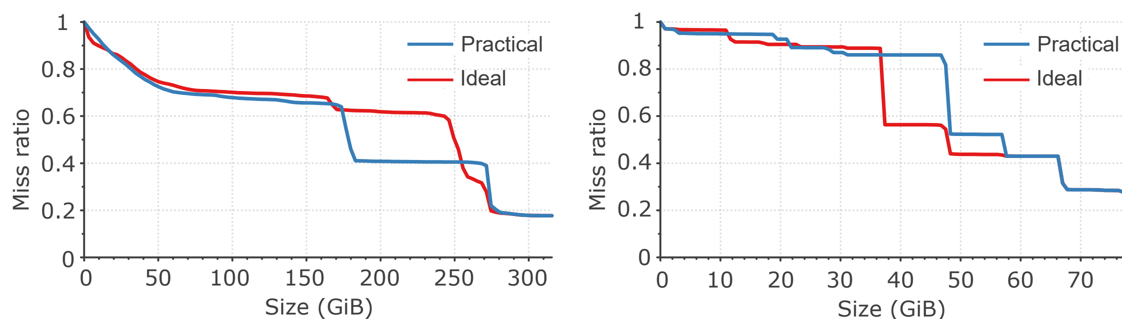


Figure 3.1: MRCs for ideal and practical LFU caches for the MSR `src1` (left) and `web` (right) workloads [42].

Table 3.3: Sample trace for FIFO caches of sizes 3 and 4 demonstrating a violation of the inclusion property. At time 6, object "a" exists in the cache of size 3, but not in the cache of size 4.

Time	Access	FIFO cache size 3	FIFO cache size 4
1	a	a	a
2	b	b, a	b, a
3	c	c, b, a	c, b, a
4	d	d, c, b	d, c, b, a
5	a	a, d, c	d, c, b, a
6	e	e, a, d	e, d, c, b

object "e" exists in the cache of size 3, but not in the cache of size 4. This is a violation of the inclusion property.

An interesting question is whether it is feasible to use MRCs generated under the assumption of ideal LFU caches (which adhere to the strict inclusion property) to model the miss ratios of practical LFU caches. Figure 3.1 demonstrates that this is not the case. The figure depicts the MRCs for ideal and practical LFU caches for two workloads. For the MSR `src1` workload [42], the miss ratios deviate substantially for cache sizes between 190GiB and 240GiB. Similarly, for the MSR `web` workload [42], the miss ratios deviate significantly for cache sizes between 38GiB and 46GiB.

Violations of the inclusion property can be shown for many other eviction policies, including:

FIFO Table 3.3 shows a simple access trace for two caches of size 3 and 4 under FIFO. At time 6,

Table 3.4: Sample trace for 2Q caches of sizes 4 and 8 demonstrating a violation of the inclusion property. Here, we use the author-recommended $K_{in} = 0.25$ and $K_{out} = 0.5$ values, and cache sizes 4 and 8 to ensure the A_{in}, A_{out}, and A_m stacks are all of size ≥ 1. At time 8, object "a" exists in the cache of size 4, but not in the cache of size 8.

		2Q cache size 4			**2Q cache size 8**		
Time	**Access**	$\mathbf{A_{in}\|1\|}$	$\mathbf{A_{out}\|2\|}$	$\mathbf{A_m\|1\|}$	$\mathbf{A_{in}\|2\|}$	$\mathbf{A_{out}\|4\|}$	$\mathbf{A_m\|2\|}$
1	a	a	–	–	a	–	–
2	b	b	a	–	b, a	–	–
3	a	b	–	a	b, a	–	–
4	c	c	b	a	c, b	a	–
5	d	d	c, b	a	d, c	b, a	–
6	e	e	d, c	a	e, d	c, b, a	–
7	f	f	e, d	a	f, e	d, c, b, a	–
8	g	g	f, e	a	g, f	e, d, c, b	–

Table 3.5: Sample trace for LRFU caches of sizes 3 and 4 demonstrating a violation of the inclusion property. Here, we use arbitrarily selected values of $p = 2$ and $\lambda = 0.5$. The values in the brackets next to each object in the caches are the CRF values. At time 9, object "e" exists in the cache of size 3, but not in the cache of size 4.

Time	**Access**	**LRFU cache size 3**	**LRFU cache size 4**
1	a	a(1)	a(1)
2	b	b(1), a(1)	b(1), a(1)
3	c	c(1), b(1), a(1)	c(1), b(1), a(1)
4	d	d(1), c(1), b(1)	d(1), c(1), b(1), a(1)
5	a	a(1), d(1), c(1)	a(1.25), d(1), c(1), b(1)
6	b	b(1), a(1), d(1)	b(1.25), a(1.25), d(1), c(1)
7	e	e(1), b(1), a(1)	b(1.25), a(1.25), e(1), d(1)
8	f	f(1), e(1), b(1)	b(1.25), a(1.25), f(1), e(1)
9	g	g(1), f(1), e(1)	b(1.25), a(1.25), g(1), f(1)

Table 3.6: Sample trace for MRU caches of sizes 3 and 4 demonstrating a violation of the inclusion property. The size column indicates the value size of each access. At time 4, object "a" exists in the cache of size 3, but not in the cache of size 4.

Time	**Access**	**Size**	**MRU cache size 3**	**MRU cache size 4**
1	a	2	a	a
2	b	2	b	b, a
3	a	2	a	a, b
4	c	1	c, a	c, b

object "a" exists in the cache of size 3 but not in the cache of size 4.

2Q Table 3.4 shows a simple access trace for two caches of size 4 and 8, respectively, under 2Q with $K_{in} = 0.25$ and $K_{out} = 0.5$. At time 8, object "a" exists in the cache of size 4 but not in the cache of size 8. This is for the following reason. At time 2, object "a" resides in the $A1_{out}$ stack, so at time 3, when "a" is next accessed, it is promoted to the A_m stack. However, in the larger cache, the object is in the $A1_{in}$ stack and is not promoted. This object is therefore evicted sooner in the larger cache and persists in the smaller cache, later causing a violation of the inclusion property.

LRFU Table 3.5 shows a simple access trace for two caches of size 3 and 4 under LRFU. Here, objects "a" and "b" are accessed for the second time at times 5 and 6, respectively, where they exist in the larger cache but not in the smaller. Their CRF values therefore grow in the larger cache and they are moved to the top of the stack, while they are assigned the initial CRF value of $F(0)$ in the smaller cache. These objects become stale in the larger cache, remaining at the top of the stack even though they are never accessed again the future. Other, more recently accessed objects are evicted instead which causes an inclusion property violation at time 9 where object "e" exists in the cache of size 3, but not in the cache of size 4.

MRU Table 3.6 shows a simple access trace for two caches of size 3 and 4 under MRU. Interestingly, while MRU was previously thought to not violate the inclusion property [48, 57, 93, 94], we found that caches under MRU violate the inclusion property when the objects are variable sized. We therefore assign each access a size. At time 4, object "a" exists in the smaller cache but not in the larger cache, because "a", with a size of 2, is evicted from the larger cache to make space for "c" with a size of 1.

In §3.3.5, we evaluate the frequency of inclusion property violations for various eviction policies using real-world cache access traces.

3.1.2 Miniature Simulations

For ease of reading, we repeat key aspects of the MiniSim [48] MRC generation algorithm described in §2.2.2 and introduce additional aspects that are relevant to this chapter.

Recall from §2.2 that Mattson and Olken are only able to generate MRCs for caches employing eviction policies that adhere to the strict inclusion property. For all other eviction policies, MiniSim is the only known reasonably efficient method of generating MRCs. Although MiniSim [48] can generate MRCs for any eviction policy, it has two key shortcomings. First, it has high memory usage as each data point on the curve simulates an instance of a cache, and the different simulations of the caches do not share any of their internal data structures. These caches, especially those with similar sizes, often contain many of the same objects, yet each cache allocates memory for these objects independently. Experimentally, we found that MiniSim used an average of 113MiB to generate a single MRC for the LFU eviction policy, with up to a maximum of 3.1GiB. To reduce this memory usage significantly, one would have to reduce the sampling rate of SHARDS or reduce the number of simulated caches, which in turn would reduce the accuracy of the resulting MRC.

A second key shortcoming is that the range of cache sizes to be simulated must be defined before the input trace is first processed and cannot be modified while the simulation is ongoing. This is limiting when generating MRCs online for live workloads, where the workload's working set size is unknown ahead of time. Because the maximum simulated cache size C_{max} cannot be modified after the simulation begins, a large, worst case, value is typically selected to ensure the access trace's working set size (i.e., the cache size required to store all unique objects) is likely to be captured. This prevents MiniSim from being able to focus the sizes of the simulated caches to regions of the MRC which lie within the workload's actual working set size.[2] For example, although Twitter tends to

[2] The sizes of MiniSim's simulated caches can be configured non-uniformly [48], though this would require knowledge of either the shape of the MRC or a specific point of interest around which to cluster the sizes of the simulated caches (e.g., the current size of the production cache) before processing the access trace. Further, the shapes of some workloads' MRCs can change dramatically over time [46, 80, 81].

overprovision its caches [3], of the publicly available access traces in the Twitter dataset, 25% have a working set size of less than 2GiB. If a large C_{max} value, such as 200GiB, is selected to model one of these access traces, the simulated caches are sized in increments of 200GiB/100 = 2GiB, preventing points of interest on the MRC from being observable. This requires MiniSim to be configured with a large number of simulated caches as reducing this to a smaller value will further reduce the resulting MRC's granularity.

3.2 Kosmo

We now present Kosmo, a new MRC generation algorithm capable of generating approximate MRCs simultaneously for a variety of eviction policies, including those that do not adhere to the inclusion property. We begin by presenting the algorithm in general terms (§3.2.2) and then describe several optimizations that significantly improve its efficiency (§3.2.3). We then describe the Kosmo algorithm for the LFU eviction policy (§3.2.4) specifically, followed by the required extensions to support other evictions policies (§3.2.5). We then show how Kosmo can be extended to support variable object sizes (§3.2.6) and TTLs (§3.2.7). Finally, we describe how Kosmo can generate MRCs for multiple eviction policies simultaneously (§3.2.8).

Both Kosmo and MiniSim simulate caches of different sizes to generate an MRC. The key difference is that MiniSim maintains a stack for each cache throughout the duration of the simulation, while Kosmo reconstructs the stacks dynamically and only as needed. MiniSim keeps track of the miss ratios of the different caches and constructs the MRC using these miss ratios once it has processed the entire access trace, while Kosmo uses an approach similar to Mattson: it records stack distances encountered in a histogram and, in the end, constructs the MRC from the histogram.

Further, MiniSim always simulates the same pre-configured cache sizes, regardless of the working set size of the access trace, while Kosmo simulates a different set of cache sizes on each access, the largest simulated cache size being the reuse distance of the currently accessed object minus one. This allows Kosmo to generate MRCs with similar error rates to that of MiniSim while simulating far fewer caches, which leads to lower memory and compute overheads for Kosmo.

The simulated caches in an instance of MiniSim do not share any internal data structures; therefore, an object may exist simultaneously in the stacks of multiple caches, causing MiniSim to consume large amounts of memory. In contrast, Kosmo maintains the data representing an object only once in a global data structure. Each object in this data structure contains the minimal amount of data required to allow the stack of a cache of any size to be reconstructed dynamically.

3.2.1 Kosmo data structures

Kosmo maintains all objects ever accessed in a data structure called the *global table*, implemented as a dynamic hash table. Each object in the global table has an associated *eviction map* which, in turn, consists of a set of *eviction records*. Whenever an object is evicted from any of the caches (of different sizes) being simulated, an eviction record is added to the eviction map of the object.[3] This eviction record includes the size of the cache from which the object was evicted, as well as other

[3] In our initial description of Kosmo in §3.2.2, we present a highly inefficient algorithm that, on each access, simulates a series of caches at byte-level granularity. However, in §3.2.3, we present an optimization that reduces the number of simulated caches and simulates caches of varying sizes on each access.

policy-specific information described further below. Using an object's eviction map, Kosmo is able to determine at any time whether the object exists in a cache of a specified size. If it exists in the cache, Kosmo can determine its position within the cache's internal data structure, referred to as the cache's *stack*, using the policy-specific information in the eviction records. An eviction map also includes a reference to the associated object, allowing it to access the object's properties, such as the last access time or ideal frequency count in the case of LFU caches.

Eviction maps are eviction policy-specific and must be implemented on a per-policy basis. However, all eviction maps support three functions:

1. `FindSmallestExisting()`: For a given object, return the size of the smallest cache that contains the object.[4]

2. `GetSortingKey()`: For the eviction map's associated object, return the object's sorting key, given a cache size, S. The sorting key is calculated using policy-specific information in the eviction records, allowing Kosmo to properly order objects in the stack of the cache of size S it is reconstructing.

3. `Insert()`: Insert a new eviction record.

The specific implementation of eviction maps for the LFU eviction policy is described in §3.2.4. The implementations for the FIFO, 2Q, LRFU, LRU, and MRU policies are described in §3.2.5. The implementations for the LRU and MRU policies are provided to demonstrate Kosmo's generality and are not included in the experimental analysis.

3.2.2 The Kosmo algorithm

We first describe a variant of the Kosmo algorithm that is highly inefficient. Here, for ease of understanding, we assume Kosmo is running on a system with infinite CPU and memory resources. In §3.2.3 we describe a series of optimizations that allow Kosmo to practically generate MRCs for real-world workloads. For now, we assume a single eviction policy. Upon each access, the Kosmo algorithm performs the following sequence of steps:

1. Calculate the reuse distance D of the accessed object and update the histogram counter associated with D.

2. Reconstruct the stacks of the caches of sizes $S < D$, for every possible cache size at byte-level granularity.[5] These are the caches that do not contain the object being accessed.

3. Select an object from each reconstructed stack for eviction (to make space for the accessed object) and place a new eviction record in the eviction map of the evicted object.

Using the accessed object's key, its associated eviction map is found using the global table. The object's reuse distance D can be determined using the object's eviction map by finding the smallest

[4] One could also structure the eviction map to determine the largest cache size which does *not* contain the object and restructure the Kosmo algorithm to accommodate this, though this will not have a noticeable effect on the accuracy of the resulting MRC.

[5] We note that simulating every possible cache size at byte-level granularity is highly impractical for real-world workloads. We present this description of Kosmo for ease of understand and, in §3.2.3, we present a series of optimizations that significantly improve Kosmo's efficiency.

sized cache in which, according to the eviction records, the object exists. The histogram counter associated with D is then incremented. If the object is not found in the global table, its reuse distance is set to infinity, as it is being accessed for the first time.

For eviction policies that do not adhere to the inclusion property, an interesting question is: what is the reuse distance of an object? For eviction policies that *do* adhere to the inclusion property, the reuse distance is clear. It is simply the minimal cache size that contains the accessed object; any larger cache will contain the object, while any smaller cache will not. For eviction policies that do *not* adhere to the inclusion property, an object may exist in a cache of size S, but not exist in some caches of size $S' > S$. Nevertheless, we argue that the size of the smallest cache containing the accessed object (even if there are larger caches that do not contain it) should be the reuse distance. There are several motivations for this choice. First, this choice allows for an important optimization to ensure eviction maps do not contain a large number of eviction records, which we describe in §3.2.3. Second, the choice simplifies the calculation of an object's reuse distance. Third, in our experiments, we found it is rare for an access to cause a violation of the inclusion property and maintain this violation for large ranges of cache sizes, which we show in §3.3.5.

Immediately after an object O is accessed, it must exist at all cache sizes. Hence, for each cache of size S in which the object does not exist when it is accessed, some object needs to be evicted to make space for O. To select an object for eviction, Kosmo reconstructs the stack of the cache by iterating through all objects in the global table. Using each object's eviction map, Kosmo determines if the object exists in the cache (of size S) and, if so, where in the cache's stack the object resides relative to other objects using the objects' sorting keys. Through this process, Kosmo reconstructs the cache's full stack. The object at the top of the reconstructed stack is selected as the object to be evicted and a new eviction record is accordingly inserted in the object's eviction map.

It is clear that Kosmo, as described, is highly inefficient. First, because the global table contains an entry for every object ever accessed, it can grow quite large. Second, Kosmo must reconstruct the stack for every cache of size less than the accessed object's reuse distance to determine which objects need to be evicted from which stacks. Finally, as an eviction record is inserted into an object's eviction map each time it is selected for eviction, the eviction maps may contain a large number of eviction records.

3.2.3 Optimizations

We describe four optimizations to the algorithm described above: cache size granularity, eviction record pruning, the use of SHARDS, and parallel stack reconstruction.

Granularity

To reduce the number of cache stacks that need to be reconstructed on each access, a granularity parameter G is introduced. This parameter limits the number of caches that are reconstructed on each access to a fixed number, namely G. For example, when an object O is accessed and cache sizes less than or equal to 3GiB are found to not contain O, their stacks must therefore be reconstructed to perform the necessary evictions; with a granularity parameter of $G = 100$, the stacks of only 100 caches are reconstructed in size increments of $3\text{GiB}/100 = 30\text{MiB}$.

When simulating caches at a reduced granularity, we introduce potential errors when determining whether an object exists at a cache of a certain size. For example, if we determine the smallest

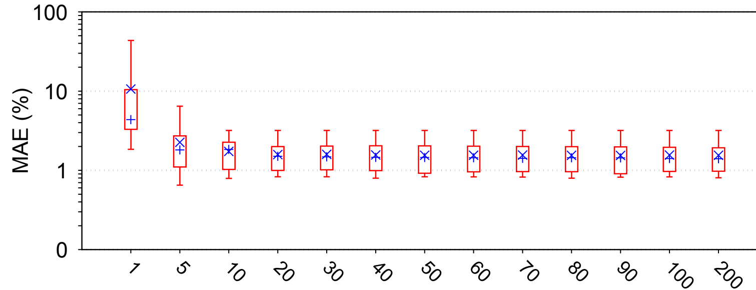


Figure 3.2: MAEs for different granularity parameters for all workloads in the MSR dataset [42]. A fixed-sized implementation of SHARDS was used with S_{max} = 2,048 for granularities between 1 and 200. For each parameter value, the top whisker identifies the maximum result while the bottom whisker identifies the minimum. The top of the box identifies the 75th percentile result, and the bottom of the box identifies the 25th percentile result. The $\times$ and $+$ symbols indicate the mean and median MAEs, respectively.

cache of size S which contains an object O while using a granularity parameter G, the object may actually exist at some cache of size between $(S - M/G, S]$, where M is the maximum cache size being considered. However, while this makes Kosmo an approximate MRC generation algorithm, we found that introducing this granularity parameter does not significantly impact Kosmo's accuracy while providing noticeable improvements in Kosmo's performance.

We experimentally evaluated appropriate values of G. A higher value of G typically means a lower MAE (mean absolute error); however, this also leads to increased computational overhead. Figure 3.2 shows the experimental results for varying values of G and the corresponding MAEs for all workloads in the MSR dataset [42]. An interesting observation is that Kosmo can achieve a low mean MAE with even a small value of G. As evident in this figure, selecting a G value greater than 10 does not significantly reduce the mean MAE. We therefore conservatively select 10 as our granularity parameter, G, and use it throughout all our simulations.

Upon accessing an object with reuse distance D, Kosmo only simulates G caches of sizes $S < D$. As a result, it is able to achieve a comparable MAE while simulating significantly fewer caches than MiniSim. The accessed object is assumed to already exist in caches of sizes $S \geq D$, so they do not need to be simulated. In contrast, MiniSim simulates all (typically 100) considered cache sizes on each access, regardless of the accessed object's reuse distance.

Eviction record pruning

To reduce the number of eviction records in an object's eviction map, each time a new eviction record is added, indicating the object is being evicted from the cache of size S, all eviction records with cache size $S' < S$ are removed. In doing so, Kosmo is implicitly assuming the inclusion property holds, where an object being evicted from a cache of size S will thereafter also not exist at any cache of size $S' < S$. This may introduce inaccuracies for eviction policies which do not adhere to the inclusion property, however, we have found these inaccuracies to be negligible (§3.3.4).

Pruning drastically reduces the size of each object's eviction map. Figure 3.3 shows the effect of pruning on the average number of eviction records in objects' eviction maps throughout a typical workload access trace. On average, pruning reduces the average size of the eviction maps (i.e., the numbers of records it contains) by a factor of roughly 387. This drastically reduces the memory

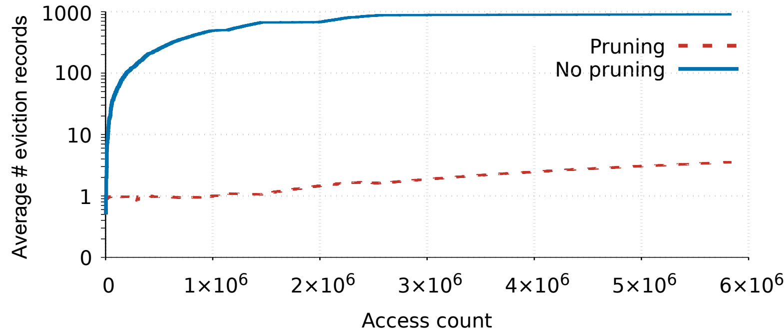


Figure 3.3: The average number of eviction records per object in the global table throughout the MSR *web* workload [42] using fixed-rate SHARDS ($R = 0.001$) shown with pruning (dashed red line) and without pruning (solid blue line). The access counts on the x-axis refer to the unsampled access counts and not the number of sampled accesses. Note the logarithmic scale of the y-axis.

required to store the eviction records as well as the computational overhead of searching through the eviction records when determining if the object exists in a cache of size S.

SHARDS

On each access, Kosmo must iterate through all objects in its global table to reconstruct the stacks of various cache sizes. The global table could include billions of objects. For this reason, we use SHARDS to spatially sample the accesses. This lessens the number of objects in the global table and reduces the stack reconstruction time. The fixed-size variant of SHARDS is particularly useful for Kosmo, as it limits the size of the global table to a known constant: the S_{max} value of SHARDS.

Parallel stack reconstruction

As the reconstruction of a cache's stack does not modify the global table, the reconstruction of the stacks of multiple caches can all be done in parallel. This improves the response time of Kosmo, but increases the memory overhead because the stacks of multiple caches exist in memory simultaneously. This creates a trade-off between throughput and memory usage. However, as we show in §3.3.4, Kosmo uses significantly less memory than MiniSim, even when reconstructing stacks in parallel.

3.2.4 Kosmo for LFU

We now describe how Kosmo generates MRCs for caches employing the LFU eviction policy. We assume that LFU adheres to the non-strict inclusion property, even though this is not the case in practice. We found that while a practical LFU cache regularly violates the strict inclusion property, it does not violate the (non-strict) inclusion property often. Although objects in the stacks of caches of different sizes may be in different orders, the objects in each cache's stack are typically a subset of the objects in the stack of a larger cache. Experimentally, we noticed that, on average, only 1.5% of accesses to an LFU cache cause the (non-strict) inclusion property to be violated. Our experimentation results will show that assuming the (non-strict) inclusion property holds for practical LFU caches produces negligible errors in the resulting MRCs (Figure 3.7).

To support the LFU eviction policy specifically, Kosmo maintains the following information in its data structures. First, each object in the global table maintains: (i) a timestamp of when the

Algorithm 1: Eviction map for LFU cache object.

```
    Ref    : object
    Record: map → OrderedMap < cache_size, count >
 1  Eviction Map LFU:
 2      Function Insert(cache_size):
 3          map.insert(cache_size, object.global_count)
 4      Function FindSmallestExisting():
 5          for record in map do
 6              if record.count == object.global_count then
 7                  return record.size + 1
 8          return object.size
 9      Function GetSortingKey(cache_size):
10          rec ← map.find_smallest(≥ cache_size)
11          if !rec then
12              return object.global_count
13          if object.global_count == rec.count then
14              return object.last_access_time
15          return object.global_count − rec.count
```

object was last accessed and (ii) a *global counter*. The global counter is incremented by one each time the object is accessed and is therefore the same as the object's frequency count in an ideal LFU cache. Second, each eviction record in the object's associated eviction map contains: (i) the size of the cache from which the object was evicted and (ii) the object's global counter value when the eviction occurred. The latter makes it possible to infer the object's frequency count for a specific cache size, referred to as the *local count*, as it would be in a practical implementation of a cache of that size.

Algorithm 1 contains the pseudocode for the eviction map's three functions. The `Insert()` function creates a new eviction record for the supplied cache size and adds it to the object's eviction map. The `FindSmallestExisting()` function returns the smallest cache size which contains the object (this is analogous to the object's reuse distance). How this is done is described further below. The `GetSortingKey()` function returns a value with which the object's position in a stack of the supplied size can be determined relative to other objects.[6]

Calculating reuse distance. The reuse distance of an object identifies the smallest cache size which contains the object being accessed. As we assume the inclusion property holds, any larger cache will also contain the object. To obtain the reuse distance of an object O when it is accessed, we search O's eviction records to identify the record with the largest cache size S wherein the record's count value is equal to the object's current global counter. The record's count value being equal to the global counter indicates that O has not been accessed in a cache of size S since O was last evicted (at which time this record was inserted). We can therefore conclude that the object does not exist in any cache of size $S' \leq S$. We can also conclude that O exists in all caches of size $S'' > S$. As a result, the object's reuse distance is $S + 1$. If no eviction record is found with a counter value

[6] Our descriptions here assume fixed-sized objects, though the algorithms shown in the listings accommodate variable-sized objects as described in §3.2.6 (i.e., `object.size` [variable-sized] in the listings corresponds to `1` [fixed-sized] in the text).

equal to the global counter, we can conclude the object exists at all cache sizes, so the reuse distance is 1.

Reconstructing cache stacks. To reconstruct the stack of a cache of size S, we use the eviction map for each object in the global table to: (i) determine if the object exists in a cache of size S, and (ii) if it does, calculate its *sorting key*. To determine if the object exists in a cache of size S, we simply check if S is greater than or equal to the object's reuse distance. If the object is found to exist in a cache of size S, its sorting key is the object's frequency count in a cache of size S, which we refer to as the object's *local count*, paired with its last access time. Once the sorting key is calculated, the object can then be placed in the correct position in the stack being reconstructed.

To calculate the *local count* of an object O in a cache of size S, we search O's eviction map for an eviction record with the smallest cache size S'' that satisfies $S'' \geq S$. If no such record exists, then O has never been evicted from any cache of size $S'' \geq S$ since O was first accessed, and therefore its local count is equal to its global count. Otherwise, if an eviction record with size $S'' \geq S$ is found, the local count is equal to O's global count minus the count value in the eviction record (i.e., O's global count when it was evicted from the cache of size S''), given that the local count should equal the number of accesses to O since its last eviction.

Updating eviction maps. After reconstructing the LFU stack of a cache, Kosmo selects the object at the top of the stack (i.e., the object with the smallest sorting key) for eviction. A new eviction record is inserted with the cache size and the object's global count into said object's eviction map.

3.2.5 Other eviction policies

The Kosmo algorithm, as described in §3.2.2, is not designed for any specific eviction policy. Kosmo can generate MRCs for other eviction policies by using the same process, but using policy-specific implementations for the eviction maps. Here, we briefly describe eviction maps for five other policies, namely FIFO, 2Q, LRFU, LRU, and MRU as examples.

FIFO. Recall from §2.1.2 that the FIFO eviction policy evicts objects in the same order in which they first entered the cache and accesses to an object which exists in a FIFO cache does not change its position within the cache's stack.

The design of the FIFO eviction map is fundamentally different than that of the LFU eviction map. A FIFO eviction record of object O identifies the cache sizes for which object O *does* exist in the cache, in contrast to LFU eviction records which indicated the cache sizes for which it *does not.* An object O's FIFO eviction map is constructed (as described further below) such that each of its eviction records stores a cache size S and the *entry time* of the object at size S: `<cache_size, entry_time>`. Each record, `<S, t>`, indicates object O's entry time for caches of sizes S', where $S \leq S' < S_{next}$ and S_{next} is the cache size stored in O's eviction record with the next largest cache size.

Algorithm 2 contains the pseudocode for the FIFO eviction map's three functions. On each access to an object O, the reuse distance is calculated as the smallest cache size S contained in O's the eviction records. This is because, the eviction record with the smallest cache size indicates the smallest cache in which the objects exists (i.e., the object's reuse distance).

Eviction records are maintained as follows. An eviction record gets added for an object O when it is accessed (similar to updating an object's global counter in the case of the LFU eviction map).

Algorithm 2: Eviction map for FIFO cache object.

```
    Ref    : object
    Record: map → OrderedMap < cache_size, timestamp >
 1  Eviction Map FIFO:
 2      Function Insert(cache_size):
 3          rec ← map.find_largest(≤ cache_size)
 4          rec.cache_size = cache_size + 1
 5          map.remove(≤ cache_size)
 6      Function FindSmallestExisting():
 7          rec ← map.find_smallest()
 8          return rec.cache_size
 9      Function GetSortingKey(cache_size):
10          rec ← map.find_largest(≤ cache_size)
11          if !rec then
12              return 0
13          return rec.timestamp
```

On every access to an object, as the object must exist at all cache sizes immediately after it has been accessed, a new eviction record with a cache of size 1 is inserted into the object's associated eviction map, if such a record does not already exist. This eviction record stores the entry time (i.e., the current time) of the object for any cache size smaller than the object's reuse distance (i.e., any cache size at which the object does not currently exist). If a record with a cache of size 1 was already present in the eviction map, the object already existed at all cache sizes and its entry time for these cache sizes should remain unchanged (the eviction map is therefore not modified).

A new eviction record for object O is also added to O's eviction map whenever object O is evicted from a cache. If it is evicted from a cache of size S, record `<S+1, entry_time>` is added to O's eviction map. The record's `cache_size` is $S+1$ because O is being evicted from the cache of size S and hence will be present in the cache of size $S+1$. To obtain O's entry time for the caches of size S and larger, we locate the record with the largest cache size S' such that $S' \leq S$. This record holds O's entry time into all caches of size S' and larger. Further, we perform pruning by removing all records with cache sizes $S' \leq S$.

When reconstructing the stack of a cache of size S, the objects' sorting keys are their entry times into the cache. To determine the entry time of an object O in a cache of size S, we find the eviction record with the largest cache size S' in O's eviction map such that $S' \leq S$. The entry time contained in this record is the object's entry time for a cache of size S. After reconstructing the stack of a cache, the object with the smallest sorting key (i.e., the object with the earliest entry time into the cache in this case) is selected for eviction and a new eviction record is inserted into its associated eviction map.

2Q. Recall from §2.1.2 that the 2Q eviction policy stores objects in two separate stacks: one for objects which have been accessed only once (the A1 stack), and one for objects which have been accessed multiple times (the Am stack). The A1 stack is further partitioned into two stacks (A1in and A1out) of size Kin and Kout (where Kin and Kout are ratios of the total cache size). Objects in the A1 stack are evicted in FIFO order, and objects in the Am stack are evicted in LRU order.

The 2Q eviction map maintains two sets of eviction records: one corresponding to the A1 stack,

Algorithm 3: Eviction map for 2Q cache object.

```
   Ref     : object
   Record: a1_map → OrderedMap < cache_size, timestamp >
   Record: am_map → OrderedMap < cache_size, count >
   Record: Kin, Kout
1  Eviction Map 2Q:
2      Function Insert(cache_size):
3          insert_a1(cache_size * (Kin + Kout))
4          insert_am(cache_size)
5      Function FindSmallestExisting():
6          a1_rec ← a1_map.find_smallest()/(Kin + Kout)
7          am_rec ← am_map.find_smallest(count ≥ 2)
8          if !am_rec then
9              return a1_rec.cache_size
10         return am_rec.cache_size
11     Function GetSortingKey(cache_size):
12         a1in_size ← cache_size * Kin
13         a1out_size ← cache_size * (Kin * Kout)
14         a1in_rec ← a1_map.find_largest(≤ a1in_size)
15         a1out_rec ← a1_map.find_largest(≤ a1out_size)
16         if !a1out_rec then
17             return A1(a1in_rec.timestamp)
18         am_exists ← map.find_any(> a1in_size and ≤ a1out_size and
             object.global_count − count ≥ 2)
19         if !am_exists then
20             return A1(a1out_rec.timestamp)
21         return Am(object.last_access_time)
```

and the other corresponding to the Am stack. While 2Q further partitions the A1 stack into two stacks, A1in and A1out, we model both as a single combined FIFO stack with a single set of eviction records since objects evicted from A1in are placed at the head of A1out. The position in the combined FIFO stack determines whether an object being accessed a second time should be promoted to Am. Each record in the A1 set of eviction records holds the same information as a FIFO eviction record and can be used to determine the entry time of an object in the A1 stack. For an object to exist in the Am stack, it must have been accessed at least twice. We can track the number of accesses of each object at varying cache sizes using the same method we used for LFU eviction records. The handling of an access to the eviction map's associated object and the insertion of a new eviction record are done using the same methods previously described for the LFU and FIFO eviction maps.

Algorithm 3 contains the pseudocode for the 2Q eviction map's three functions. On each access to an object, its reuse distance is dependent on whether the target object is in the A1 or the Am stack. We therefore calculate two different reuse distances assuming the object is in each stack and select the smaller value (as the cache associated with the larger reuse distance will inherently also contain the object according to the inclusion property) as the reuse distance.

To calculate the reuse distance of the object assuming it exists in the A1 stack, similar to the method used with FIFO, we find the smallest A1 stack of size S_{A1} which contains the object by finding the eviction record in the A1 eviction record set with the smallest cache size. The

corresponding cache size which contains the object is then $S_{A1}/(Kin + Kout)$. To calculate the reuse distance of the object assuming it exists in the Am stack, we search the Am eviction record set for the eviction record with the smallest cache size which has a local count ≥ 2. This eviction record corresponds to the smallest cache of size S_{Am} which contains the object in the Am stack. If no such record exists, the object must exist in the A1 stack and the previously calculated cache size corresponding to the A1 stack is selected as the reuse distance. Otherwise, the reuse distance is $min(S_{A1}/(Kin + Kout), S_{Am})$.

When reconstructing a 2Q stack, we reconstruct the A1 and Am stacks separately. Unlike the previously described policies, an object in a 2Q cache can exist in one of three different stacks: A1in, A1out, or Am. To determine in which stack an object exists for a cache of size S, we use the object's 2Q eviction map to determine its FIFO entry time if it were to exist in the A1in or A1out stack and its local count if it were to exist in the Am stack. As the size of the A1in and A1out stacks are only a ratio of the total cache size, we find the object's entry time in the A1in and A1out stacks for caches of size $S_{A1in} = S * Kin$ and $S_{A1out} = S * (Kin + Kout)$, respectively.[7] If no entry time is found for the object in the A1out stack, it must exist in the A1 stack with the associated A1in entry time. If an entry time is found for the object in the A1out stack, we search the Am eviction records to determine if the object has a local count ≥ 2 for a cache of size $S_{A1in} < S' \leq S_{A1out}$. If such a record exists, it indicates the object has been accessed at least twice while existing in the A1out stack and is therefore in the Am stack. Otherwise, it is in the A1 stack with the associated A1out entry time.

S3-FIFO. Recall from §2.1.2 that the S3-FIFO eviction policy stores objects in three separate FIFO stacks: one for objects which have been accessed only once (the S stack), one for objects which have been accessed multiple times (the M stack), and one which stores the metadata of evicted objects and allows for their immediate placement in the M stack if they are accessed again in the future (the G stack).

The implementation of an eviction map for the S3-FIFO eviction policy [10] is a simple adaptation of the eviction map for the 2Q eviction policy. It uses three sets of eviction records: one for the S stack, one for the M stack, and one to track the number of accesses to each object (as with LFU and 2Q eviction records). When reconstructing the S3-FIFO stack, Kosmo reconstructs a separate stack for the S and M stacks. The G stack is inherently maintained by the object's existence in Kosmo's global table. When updating the eviction maps of objects, if an object in the M stack is selected for eviction and has a local count ≥ 0, it's local count is reduced by 1 and its eviction record is not further updated. This mimics the behavior of objects selected for eviction from the M in the S3-FIFO eviction policy (i.e., an object is not immediately evicted from the M stack if its frequency count is ≥ 1; its frequency count is decremented and it is placed at the head of the M stack). We believe a similar technique would work for other multi-stack eviction policies (e.g., ARC [50]) but leave this for future work.

LRFU. Recall from §2.1.2 that the LRFU eviction policy orders objects by their associated *Combined Recency and Frequency* (CRF) value which has an initial value of $F(0)$, where $F(x) = (\frac{1}{p})^{\lambda * x}$, p is a value greater than or equal to two, and λ is a value between 0 and 1. Upon access to an object, its CRF value is updated as $CRF_{updated} = F(0) + F(t_{now} - t_{last_access}) * CRF_{last}$.

[7] Here, we use the combined size of the A1in and A1out stacks when searching for the object's entry time in the A1out stack as the stacks behave as one coherent FIFO stack.

Algorithm 4: Eviction map for LRFU cache object.

```
   Ref    : object
   Record: map → OrderedMap < cache_size, crf >
1  Eviction Map LRFU:
2    Function Insert(cache_size):
3      rec ← map.find_largest(≤ cache_size)
4      rec.cache_size = cache_size + 1
5      map.remove(≤ cache_size)
6    Function FindSmallestExisting():
7      rec ← map.find_smallest()
8      return rec.cache_size
9    Function GetSortingKey(cache_size):
10     rec ← map.find_largest(≤ cache_size)
11     if !rec then
12       return 0
13     return rec.crf
14   Function Update(access):
15     for record in map do
16       record.crf ← f(0) + f(access.timestamp − object.last_access_time) ∗ record.crf
```

The LRFU eviction map is implemented similarly to that of FIFO. Each object in a cache has an associated CRF value (see Figure 2.1.2). Each eviction record in an LRFU eviction map contains the cache size S and the CRF value of the object at S. It identifies the CRF value of the associated object for caches of sizes S', where $S \leq S' < S_{next}$ and S_{next} is the cache size stored in the eviction record with the next largest cache size. On each access to object O, a new eviction record is added to O's eviction map with S set to 1 and CRF set to $F(0)$. Similar to FIFO, if an eviction record with a cache size of 1 already exists, it is not modified. Moreover, the CRF value of each eviction record is updated using the current CRF value and the object's last access time.

Algorithm 4 contains the pseudocode for the LRFU eviction map's three main operations. Similar to when using a FIFO eviction map, the reuse distance of an accessed object is determined using the object's eviction map as the cache size contained in the eviction record with the smallest cache size.

When reconstructing the stack of a cache of size S, the objects in the stack are ordered by their CRF values from smallest to largest. To determine the CRF value of an object O in a cache of size S, we find the eviction record with the largest cache size S' in O's eviction map such that $S' \leq S$. The CRF value contained in this record is the object's CRF value at a cache of size S.

After stack reconstruction, the object with the largest CRF value is selected for eviction. A new eviction record is then inserted into the object's eviction map. This process is identical to that of a FIFO eviction map.

LRU. Recall from §2.1.2 that the LRU eviction policy evicts the least recently used object from a cache. In practice, one would always generate an MRC for the LRU eviction policy using SHARDS and Olken as it is far more efficient than any other known methods. We present a method of generating this MRC using Kosmo simply to demonstrate Kosmo's generality.

Algorithm 5 contains the pseudocode for the LRU eviction map's three main operations. Because

Algorithm 5: Eviction map for LRU cache object.

```
   Ref    : object
   Record: cache_size ← 0
1  Eviction Map LRU:
2      Function Insert(size):
3          cache_size ← size
4      Function FindSmallestExisting():
5          return cache_size + 1
6      Function GetSortingKey(size):
7          return object.last_access_time
8      Function Update():
9          cache_size ← object.size
```

the LRU eviction policy adheres to the strict inclusion property, the order of the objects in the simulated caches (of different sizes) will always be the same and can be determined using the objects' last accessed times. By simply storing the reuse distance D of each object as the sole eviction record in its eviction map, Kosmo can reconstruct the stack of a cache of size S by first determining which objects exist in the cache (any object with a reuse distance $D' \leq S$), then ordering the objects that exist by their last access times. The last access times of objects are stored alongside each object in the global table.

When reconstructing the stack of a cache of size S, the objects' sorting keys are their last access times. To determine if an object should be included in the reconstructing stack of size S, we select objects which contain an eviction record that have a cache size $S'' > S$.

After reconstructing the stack of a cache, the object with the smallest last access time is selected for eviction. The same object may be selected for eviction from multiple reconstructed cache stacks. As the object's eviction map has only one eviction record, the object is evicted from the cache with the largest size. In doing so, as LRU adheres to the strict inclusion property, Kosmo is effectively evicting the object from all caches of smaller sizes as well.

As an object is moved to the front of the LRU stack each time the object is accessed, immediately after, it will exist in the stacks of all caches, regardless of size, until it is again evicted from a cache. Therefore, each time an object is accessed, its associated eviction map updates the object's stored reuse distance to 1 to indicate it now exists at all cache sizes.

MRU. Recall from §2.1.2 that the MRU eviction policy evicts the most recently used object from a cache.

Algorithm 6 contains the pseudocode for the MRU eviction map's three main operations. The implementation of an eviction map for the MRU eviction policy is virtually identical to that of the LRU eviction policy except that the sorting key in the MRU eviction map is the negative value of the object's last access time.

3.2.6 Variable object sizes

The Kosmo algorithm described thus far generates MRCs assuming the cache is being used for fixed-size objects. However, modern applications use caches to store objects of varying size (e.g., key-value

Algorithm 6: Eviction map for MRU cache object.

```
   Ref    : object
   Record: cache_size ← 0
1  Eviction Map MRU:
2      Function Insert(size):
3          cache_size ← size
4      Function FindSmallestExisting():
5          return cache_size + 1
6      Function GetSortingKey(size):
7          return −object.last_access_time
8      Function Update(access):
9          cache_size ← object.size
```

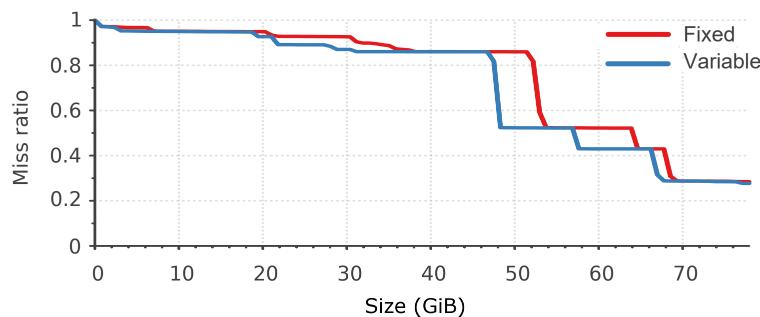


Figure 3.4: MRCs for the MSR web workload [42] for fixed versus variable-sized objects using the LFU eviction policy.

caches). As such, the MRCs generated by these algorithms may not adequately represent the miss ratios experienced by the caches under these workloads. Figure 3.4 demonstrates the difference in MRCs for the same workload when taking variable-sized objects into account versus not taking them into account. It is evident that these two MRCs differ significantly and variable-sized objects should be accounted for accordingly in MRC generation algorithms.

With fixed-sized objects, Kosmo inserts an eviction record into the eviction map of only one object when a new object is accessed. However, with variable-sized objects, more than one object may need to be evicted. A simple modification allows the algorithm to handle variable-sized objects. While a cache's stack is being reconstructed, the cache's used size is calculated by summing the size of all objects which exist in the cache. Objects are then evicted from the top of the stack until the total used size of the cache is less than or equal to the cache's size. (For each evicted object, an eviction record may have to be added to the object's associated eviction map, depending on the eviction policy.)

3.2.7 TTLs

Taking *time-to-live* (TTL) parameters into account can also significantly affect the resulting MRC [95]. Minor modifications to the Kosmo algorithm can allow for the support of TTL parameters for objects. When an object is first accessed, a corresponding *expiry time* is calculated based on its TTL.

Table 3.7: Access trace datasets used in our simulations.

Dataset	Access traces	Total accesses
MSR [42]	13	434,212,008
Twitter [3]	24	99,200,180,813
SEC [96, 97]	15	26,482,889,754

(If the TTL is 0, no expiry time is specified.) Expiry time describes the time at which the object should be evicted from all caches, regardless of size. To handle this in Kosmo, when iterating through the global table upon each access to reconstruct a cache's stack, each object's expiry time is compared against the current time (i.e., the time of the current access). If the object has not expired and exists in the cache, it is added to the stack; otherwise, it is excluded.

3.2.8 Simultaneous MRC generation

One key advantage of Kosmo is its ability to generate MRCs for multiple eviction policies simultaneously in a single pass. In the previous descriptions of eviction maps, each object in the global table has one associated eviction map. The type of eviction map used is based on the eviction policy for which an MRC is being generated. However, we can easily extend the global table to have multiple eviction maps associated with each object – one for each eviction policy being supported. This allows Kosmo to reconstruct the internal stack of any cache size for any eviction policy being supported.

Minor modifications to the previously described Kosmo algorithm (§3.2.2) must be made to support multi-policy MRC generation. As an MRC must be generated for each eviction policy, the algorithm must maintain a separate histogram per policy. Upon each access to an object, the reuse distance is calculated for each eviction policy and its corresponding histogram is updated. Each eviction policy-specific reuse distance is used to determine which cache stack sizes must be reconstructed for each eviction policy. As each object's eviction policy-specific eviction maps are independent of one another, the eviction maps can be updated in parallel once the eviction policy-specific cache stack reconstructions are complete and the objects to be evicted have been identified.

By performing the reuse distance calculation, cache stack reconstruction, and eviction map updating for each configured eviction policy simultaneously upon each access, Kosmo can, at any point, generate an MRC for any configured eviction policy using the eviction policy's histogram.

3.3 Evaluation

We evaluated Kosmo using 52 publicly-accessible cache access traces from MSR [42], Twitter [3], and SEC [96, 97]. Table 3.7 shows a summary of the datasets we used in our evaluation. For the Twitter dataset, we used the recommended traces as specified by Twitter [98] as well as 7 other randomly selected traces in the dataset.[8] Similar to prior studies, we only considered the GET/READ accesses in each trace [46, 99].

For LFU and FIFO, we evaluated Kosmo's performance using all 52 access traces. For 2Q and LRFU, we used all access traces in the MSR and SEC datasets, and 5 randomly selected access

[8] The randomly selected traces are: `cluster1`, `cluster3`, `cluster8`, `cluster10`, `cluster26`, `cluster50`, and `cluster53`.

traces from the Twitter dataset.[9] We use a randomly-selected subset of our traces for these eviction policies due to time limitations (as the generation of the accurate MRCs for the LFU eviction policy required roughly 4 months of continuous compute time).

We ran both Kosmo and MiniSim with three configurations of SHARDS: one fixed-rate and two fixed-size. The authors of SHARDS noted that for fixed-rate SHARDS, an R value of 0.001, and for fixed-size SHARDS, an S_{max} value of 2,048 produce reasonably accurate MRCs [24]. We selected these same values for our simulations, but also used fixed-size SHARDS with an S_{max} value of 1,024 to examine the memory and throughput benefits as well as the reduction in accuracy. For all fixed-size configurations of SHARDS, we used an initial sampling rate of $R = 0.1$ as we found this produces accurate results.

3.3.1 MiniSim implementation

We implemented the MiniSim algorithm as described by the authors of the original paper, with the same configuration parameters [48]. The authors only describe MiniSim configured using the fixed-rate SHARDS variant; therefore, we extended MiniSim to also support the fixed-sized SHARDS variant.

Unlike fixed-rate SHARDS, which keeps the sampling rate R constant throughout the access trace, fixed-size SHARDS gradually decreases R to ensure that at any given time there are at most S_{max} distinct objects in the MRC generation algorithm's internal data structures. SHARDS tracks these unique objects in a set S. To extend MiniSim to support fixed-size SHARDS, we initially scale each simulated cache size by the initial sampling rate R. For each access, if the sampling rate R is decreased to R_{new}, we remove all objects no longer in S from all simulated caches. We then rescale the size of each simulated cache by R_{new} using the eviction policy of the cache. A key insight is that when the size of a simulated cache is rescaled, we also rescale the cache's access counter (i.e., the number of accesses the cache has observed) and hit counter by the factor R_{new}/R_{old}.

The implementation of MiniSim described in the original paper statically allocates the required memory for each simulated cache before processing an access trace. This is possible as the sampling rate R, and thus the size of each simulated cache, is fixed. In our extension to support a varying sampling rate, we allocate memory dynamically so as to be able to release memory when R decreases.

Our LFU implementation follows a well-known algorithm, which is optimized for throughput, to allow for constant time complexity for each access [60]. Our implementation of 2Q follows that described by Johnson and Shasha in the original paper. We used Kin and Kout values of 25% and 50%, respectively, for both our Kosmo and MiniSim simulations. These were the same values suggested in the original paper. Our implementation of the LRFU eviction policy follows the description in the original paper [49]. We arbitrarily selected a λ value of 0.5 for our experiments; we also experimented with other values of λ, such as 0.001, and found the results to be similar.

3.3.2 Environment

All experiments were done on Ubuntu 22.04.2 with an AMD Ryzen Threadripper 3990x (64 cores) with 256GB of $DDR4-3200MHZ$ DRAM. The access traces were stored in binary format on a Sabrent Rocket Q 8TB. Both Kosmo and MiniSim use a thread pool with separate threads for

[9] The randomly selected traces are: `cluster7`, `cluster22`, `cluster31`, `cluster45`, and `cluster50`.

reconstructing Kosmo stacks and separate threads for MiniSim's simulated caches. We tested various thread pool sizes and observed the best performance for MiniSim when the thread pool's size was equal to the number of cores. Kosmo's performance remained the same after the thread pool's size exceeded the configured granularity (i.e., the maximum number of concurrently reconstructed cache stacks).

3.3.3 Metrics

We used three metrics in our evaluation: memory usage, throughput, and MRC accuracy.

Memory usage. To measure the memory usage of each algorithm, for each access trace, we ran the algorithm in an isolated process and measured the high water mark of the RSS [100] after it had processed the entire access trace. This metric has been used in prior work to evaluate the memory usage of MRC generation algorithms [24].

Throughput. To measure the throughput of each algorithm, for each access trace, we divided the total runtime by the number of accesses in the trace. IO time is excluded from the measurement of the total runtime.

Accuracy. To measure the error of both Kosmo and MiniSim, we calculated the *mean absolute error* (MAE) of each of the generated MRCs using the corresponding exact MRC. As no algorithm exists capable of generating exact MRCs for the LFU, FIFO, 2Q, and LRFU eviction policies, we performed 100 full simulations of caches of varying size (evenly distributed over the access trace's working set size) for each policy and for each access trace. These 100 points are the same points selected when running MiniSim. To measure the error, we obtained the MAE by calculating the difference between the exact MRC and the approximate MRCs generated by Kosmo and MiniSim at these points and then taking the mean.

3.3.4 Results

Figure 3.5 to Figure 3.7 show the performance results of Kosmo and MiniSim. For each algorithm, the range of results for the various traces in the datasets is shown. For each result, the bottom and top whiskers identify the minimum and maximum values, respectively. The bottom and top of each box are the 25th and 75th percentile values, respectively. The $\times$ and $+$ symbols indicate the mean and median values, respectively.

Figure 3.5 shows the memory usage of Kosmo and MiniSim for the LFU, FIFO, 2Q, and LRFU eviction policies. We found that Kosmo uses an average of 3.6 times less memory than MiniSim, and up to 36 times less in the extreme case.

Figure 3.6 shows the throughput of Kosmo and MiniSim for the LFU, FIFO, 2Q, and LRFU eviction policies. We found that Kosmo has an average throughput 1.3 times higher than that of MiniSim. Notably, Kosmo has a lower average throughput than MiniSim (0.54 times that of MiniSim) for the 2Q eviction policy. This is attributed to Kosmo reconstructing two stacks (A1 and Am) on each access.

Figure 3.7 shows the MAE of Kosmo and MiniSim for the LFU, FIFO, 2Q, and LRFU eviction policies. We found Kosmo and MiniSim to typically generate MRCs with similar accuracy. Across all simulations, Kosmo and MiniSim had an average MAE within 0.25% of one another. Although

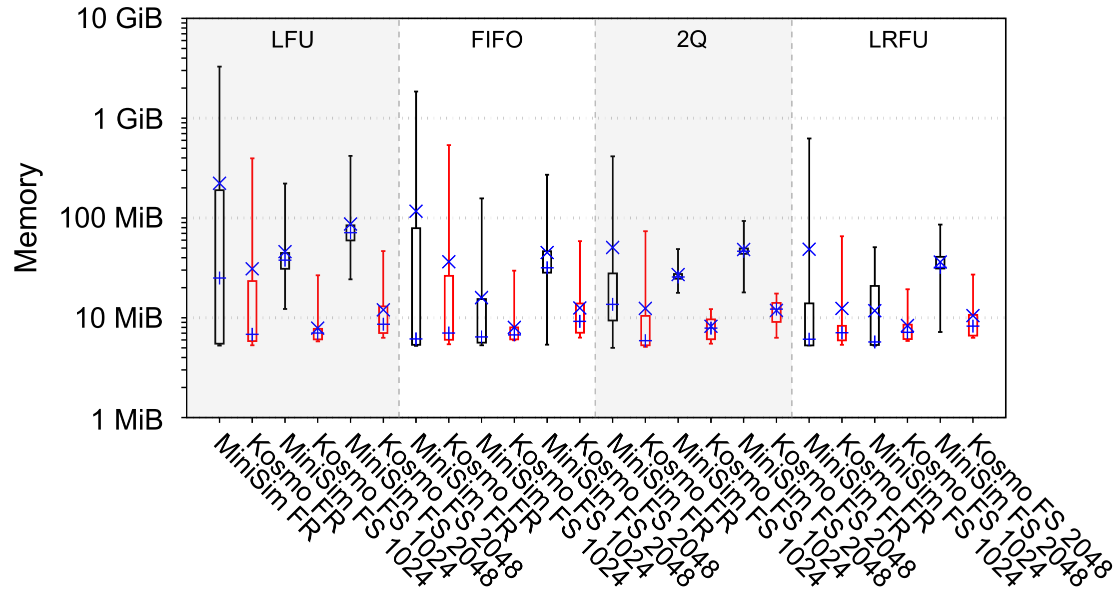


Figure 3.5: Memory usage of Kosmo and MiniSim for all eviction policies. Note the logarithmic scale of the y-axis.

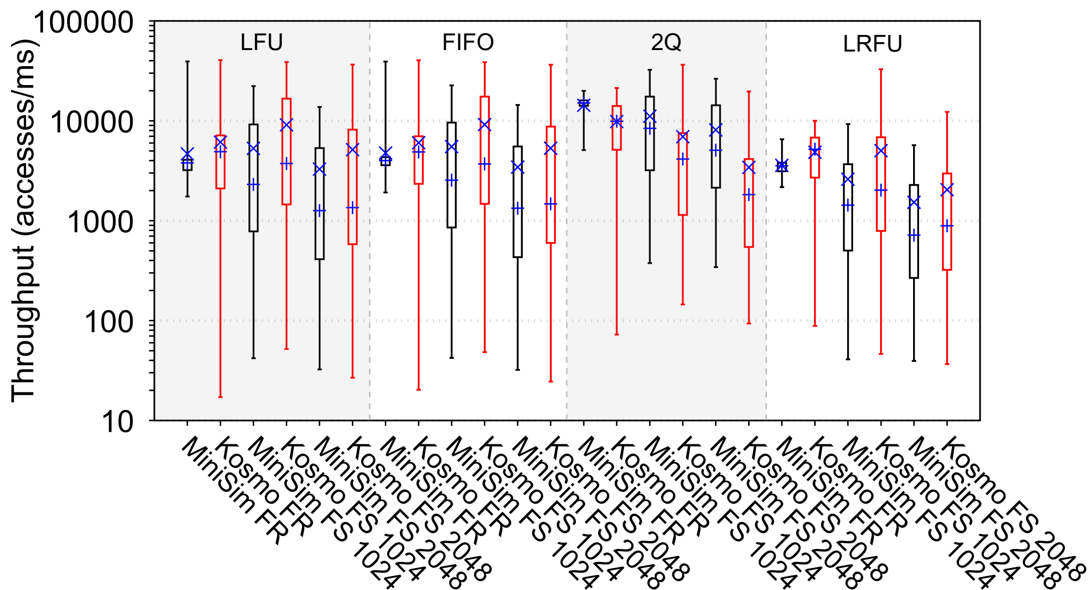


Figure 3.6: Throughput of Kosmo and MiniSim for all eviction policies. Note the logarithmic scale of the y-axis.

Kosmo generates MRCs with lower MAEs, on average, for LFU and LRFU (0.16% and 0.86% lower for LFU and LRFU, respectively), it generates MRCs with higher MAEs, on average, for FIFO and 2Q (0.44% and 1.6% higher for FIFO and 2Q, respectively). This is attributed to the higher rates of violations of the inclusion property for FIFO and 2Q, which we show in §3.3.5. Further, although

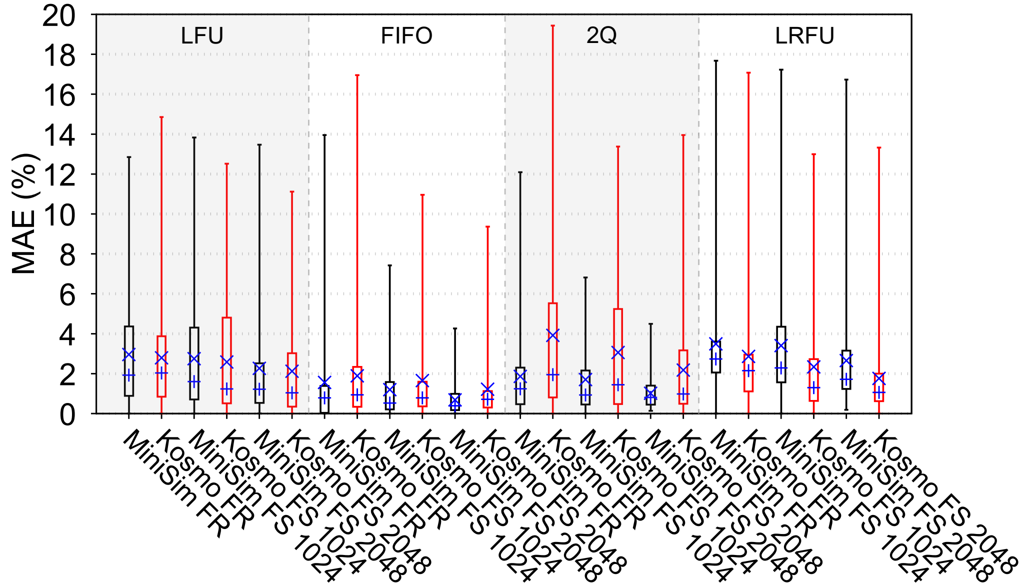


Figure 3.7: MAE of Kosmo and MiniSim for all eviction policies.

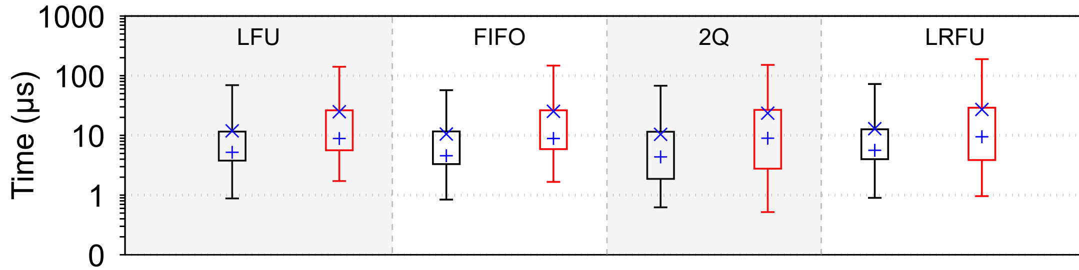


Figure 3.8: CPU time per access for the LFU, FIFO, 2Q, and LRFU eviction policies for MiniSim (left, black) and Kosmo (right, red) using fixed-sized SHARDS (S_{max} = 2,048).

the average MAE for the MRCs generated by Kosmo for 2Q is 1.6% higher than those generated by MiniSim, the median is only 0.35% higher. This is attributed to the high MAE of one access trace, `src1` in the MSR dataset [42], which has an unusually high MAE. This access trace violates the inclusion property at a significantly higher rate than other access traces.

To evaluate the CPU usage of Kosmo and MiniSim, we measured the CPU time per access for each access trace. Figure 3.8 shows that Kosmo's CPU time per access is roughly 1.85 times higher than that of MiniSim for LFU and 2 times higher for FIFO, 2Q, and LRFU. The inconsistency between the lower average CPU time per access of MiniSim than that of Kosmo, and the higher average throughput of Kosmo than that of MiniSim can be attributed to MiniSim's threads idling more frequently than Kosmo's threads. As operations on MiniSim's simulated caches occur in parallel, on separate threads, some stack operations (e.g., those on smaller simulated caches which notice more frequent evictions) may take longer to complete than others, causing some threads to idle.

To evaluate the effects of varying the number of MiniSim's simulated caches on its performance, we also tested MiniSim with 20 and 50 simulated caches for the LFU eviction policy (instead of 100).

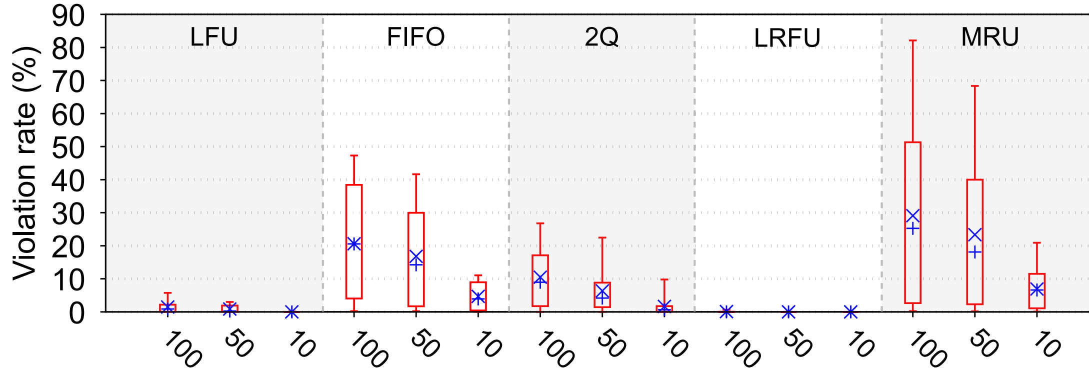


Figure 3.9: Ratio of accesses for which a violation of the inclusion property occurs for the LFU, FIFO, 2Q, LRFU, and MRU eviction policies across all workloads in the MSR dataset [42] when simulated with 100, 50, and 10 points.

With 20 simulated caches, MiniSim consumes roughly 1.2 times the memory of Kosmo on average and exhibits roughly similar throughput, however it has roughly 2 times the MAE of Kosmo. With 50 simulated caches, MiniSim consumes roughly 2.3 times the memory of Kosmo and has 10% lower throughput with roughly identical MAE. Notably, as discussed in §3.1.2, the C_{max} value of MiniSim must be selected before knowledge of the access trace; therefore, using a low number of simulated caches such as these may result in unobservable points of interest on the MRC.

3.3.5 Inclusion property violations

An interesting question is that while Kosmo assumes the inclusion property holds for its configured eviction policies, how often do violations of the inclusion property occur in practice? Figure 3.9 shows the percentage of accesses for which a violation of the inclusion property occurs (i.e., accesses which reference an object that does not exist in a cache of size S but exists in a cache of size $S' < S$) for the LFU, FIFO, 2Q, LRFU, and MRU eviction policies across all access traces in the MSR dataset [42]. We found these violations by simulating 100 caches of varying sizes evenly distributed over the access trace's working set size and, for each access, finding the smallest simulated cache in which the object exists, then searching for a larger cache in which it does not.

To examine the severity of these violations, indicated by the difference between the size of the smaller cache wherein an object exists and the larger cache wherein it does not, we repeated these simulations with 50 and 10 simulated caches. This increases the size intervals between the simulated caches and thus, if the number of violations remains high, we can infer the violations occur in large ranges of cache sizes. This metric of severity is useful in understanding the accuracy of Kosmo's generated MRCs, as a large difference between the cache sizes for which a violation of the inclusion property occurs indicates that Kosmo may reconstruct the stacks of and update the eviction maps of objects from significantly incorrect ranges of cache sizes, leading to high errors when computing objects' reuse distances.

For the LFU eviction policy, we found that 1.5% of accesses violated the inclusion property when measured with 100 simulated caches. This reduces by 40.9% and 99.3% to 0.88% and 0.01% when measured with 50 and 10 points, respectively. This indicates that inclusion property violations do

not occur frequently for caches under LFU, and, when they do occur, they are not severe and thus Kosmo's reuse distance calculations (and its generated MRCs) will not notice high errors. For the FIFO eviction policy, we found that 20.6% of accesses violated the inclusion property with 100 points, reducing by 18.5% and 77.2% to 16.8% and 4.7% for 50 and 10 points, respectively. For the 2Q eviction policy, we found that 10.5% of accesses violated the inclusion property with 100 points. This reduces by 39.4% and 84.3% to 6.4% and 1.7% for 50 and 10 points, respectively. We found that violations of the inclusion property are rare for the LRFU eviction policy (0.08% of accesses violated the inclusion property with 100 points). The higher rates of violations under FIFO and 2Q explain the higher MAEs of MRCs generated by Kosmo for these policies; however these MAEs are typically still within acceptable margins. Interestingly, we found that MRU violates the inclusion property at a higher rate than the other evaluated eviction policies with an average of 29.1% when simulated with 100 points, while MRU is often considered to not violate the inclusion property [48, 57, 93, 94]. (We note that for the MRU eviction policy, violations of the inclusion property only occur when considering variable-sized objects.)

3.4 Related work

There have been many proposed MRC generation algorithms [24, 43–48, 74, 77, 93, 99, 101–103]; however, these are largely focused on the LRU eviction policy. Many studies have suggested new eviction policies to improve on observed limitations of policies such as LRU [11, 20, 49, 50, 104–109]. However, to the best of our knowledge, prior to Kosmo, MiniSim was the only algorithm capable of generating MRCs for policies which do not adhere to the strict inclusion property.

Beckmann and Sanchez describe a probabilistic method of generating MRCs for other age-based eviction policies [101], such as *protecting distance based policy* (PDP) [104] or *inter-reference gap distribution replacement* (IGDR) [108].

Yu et al. propose an extension to MiniSim, called *DFShards*, to modify the number of simulated caches during runtime [93]. However, we found that the cost of instantiating a new simulated cache at runtime significantly reduces the throughput of DFShards making it unsuitable for online MRC generation.

Since the publication of our work on Kosmo [8], Zhao et al. proposed a new extension to MiniSim called *LAShards* [110]. Similar to DFShards [93], LAShards modifies the number of simulated caches during runtime. LAShards reduces the overhead of periodically generating MRCs for a workload by monitoring properties of the workload's access trace, such as the working set size and minimum miss ratio, and only generating an MRC when these change more than a configurable threshold.

3.5 Conclusion

In this chapter, we proposed Kosmo, a novel method for the simultaneous generation of miss ratio curves (MRCs) for multiple eviction policies. We showed that the current method of generating MRCs for eviction policies that do not adhere to the strict inclusion property have significant memory overhead and are therefore not suitable for online MRC generation. Our experimental results show that Kosmo uses significantly less memory than MiniSim configured with 100 simulated caches while maintaining similar accuracy. Kosmo uses 3.6 times less memory than MiniSim on average, and up

to 36 times less in the most extreme case. Kosmo has an average throughput 1.3 times that of MiniSim.

Kosmo enables the generation of multi-eviction policy MRCs online with low resource usage. This allows for the real-time evaluation of a cache's miss ratio and quantifies the effects of modifying its configuration parameters on its performance. We use Kosmo to inform the decisions made by Flux (Chapter 5), an online cache orchestration algorithm which manages a fleet of cache hosting servers, allowing it to periodically reconfigure its managed caches and reduce resource usage through cache resizing and server consolidation, or improve cache performance by switching eviction policies.

As future work, we plan to expand Kosmo's supported eviction policies to more complex policies, such as LHD [11], LIRS [63], or ARC [50]. We also plan to improve on Kosmo's throughput by reducing its computational overhead through the use of more specialized data structures.

Chapter 4

PaperCache: Multi-Eviction Policy In-Memory Caching

A cache's eviction policy can significantly affect its performance, typically measured as the cache's *miss ratio* (i.e., the ratio of the number of accesses to data not found in the cache to the total number of accesses) [8]. The *least recently used* (**LRU**) eviction policy is the most widely deployed policy, and is used as the default (and in most cases only) policy in popular in-memory caches, such as Redis [16] or Memcached [17]. However, many alternative eviction policies exist, such as *least frequently used* (**LFU**), *first-in-first-out* (**FIFO**), *most recently used* (**MRU**), 2Q [20], *least recently/frequently used* (**LRFU**) [49], ARC [50], S3-FIFO [10], LHD [11], or SIEVE [51], which have been shown to outperform LRU for specific workloads.

Recently proposed cache performance modeling techniques, such as Kosmo [8] (described in the previous chapter) and MiniSim [48], are able to accurately determine a cache's optimal eviction policy based on its workload and allocated size. These algorithms generate *miss ratio curves* (**MRCs**) which plot a cache's miss ratio as a function of its allocated size. Unfortunately, most modern caches do not support multiple eviction policies, and those that do, such as Redis [16], make compromises on policy accuracy to reduce management overhead (§4.1.1).[1]

This chapter introduces **PaperCache** [52], a novel cache design that supports dynamic switching between any eviction policy during runtime. PaperCache uses a unique eviction policy approximation technique with a *MiniStack* to temporarily approximate the behavior of an eviction policy while switching between policies and servicing new accesses. We evaluate the performance of Paper-Cache and the accuracy of MiniStacks using publicly-available real-world cache access traces from Twitter [3], Cloudphysics [24], IBM [25], Tencent CBS [26], Alibaba [27], and Wikipedia [28]. In §4.3, we show that PaperCache can switch between eviction policies instantaneously while continuing to service new accesses and achieves a miss ratio within 1% of exact implementations of its eviction policies. We demonstrate that PaperCache's ability to periodically switch to its most performant eviction policy at runtime can reduce the miss ratio by between 8.2% and 48.5% when compared to statically configured eviction policies.

[1] CacheLib [31] also supports multiple eviction policies, including arbitrary user-definable policies. However, it does not support the dynamic switching between them at runtime which we show is important in §4.1.3.

Contributions

In this chapter, we make the following contributions:

- We demonstrate that Redis' eviction policy approximation technique leads to inaccurate policy behavior after switching policies during runtime (§4.1.1).
- We demonstrate that the optimal eviction policy for a cache can change over time (§4.1.3).
- We introduce the design and implementation of PaperCache, an in-memory cache that supports the efficient dynamic switching between any eviction policy in real-time (§4.2), and we evaluate PaperCache on real-world workloads.

Limitations

Our work on PaperCache has the following limitations:

- Our analysis is based on publicly-available workloads and our findings may not apply to all caching workloads.
- PaperCache has higher metadata memory overhead than, say, Redis (10.8% and 70% higher when storing between 100,000 and 1,000,000 objects, respectively, when configured with 8 eviction policies).

4.1 Background and motivation

In this section, we describe relevant prior work and the motivation for PaperCache. We first describe how Redis implements its LRU and LFU eviction policies, and how it switches between them (§4.1.1). Next, we briefly re-introduce SHARDS, a sampling technique introduced in §2.2.2, because PaperCache uses SHARDS to reduce the overhead of storing eviction policy metadata (§4.1.2). Finally, we show that the optimal eviction policy for a cache may change over time as motivation for PaperCache's dynamic eviction policy switching capabilities (§4.1.3).

4.1.1 Multi-eviction policy support in Redis

Popular open-source in-memory caches are rather limited in the number of eviction policies they support. Memcached [17] only supports a variant of LRU, while other caches, such as CacheLib [31], support multiple eviction policies but cannot switch between them at runtime. Redis supports LRU and LFU and can switch between them at runtime [16]. In Redis, switching the eviction policy is a *passive* configuration change (i.e., it does not trigger any action by an idle cache).

Redis employs several performance optimizations, which makes its eviction policies only approximations of LRU and LFU, but which reduces its metadata memory usage. Two key optimizations are:

- An object is allocated a 24-bit metadata field to store any required data pertaining to the eviction policy currently in force. For LRU, this is the last access time. For LFU, this is the logarithmic frequency count (8 bits) per object and the last access time with reduced precision (16 bits). There is only one such field, so its content can only pertain to one eviction policy

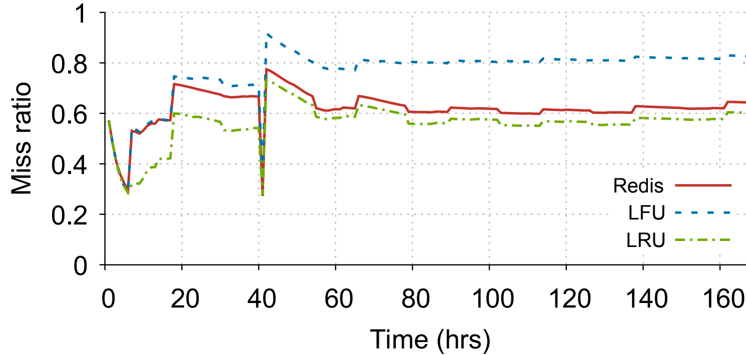


Figure 4.1: Miss ratios of LFU, LRU, and Redis switching from LFU to LRU at 40 hrs for the Cloudphysics `w96` trace [24].

at a time. On a policy switch, the content gets updated to that of the new policy upon access to the object. As a result, the behavior of the cache may deviate from the expected behavior for a period of time.

- Redis does not maintain an LRU or LFU stack. To select an object for eviction, a number of objects (5 by default) are randomly selected and the oldest (in the case of LRU) or least frequently accessed (in the case of LFU) is selected for eviction.[2] This method of eviction policy approximation causes the observed miss ratio to deviate from that of exact implementations of the LRU or LFU eviction policies; typically for the worse.

To observe the effectiveness of Redis' ability to switch its eviction policy between LRU and LFU, we instantiated a Redis v8.0.1 cache of size 500MiB, initially configured with the LFU eviction policy,[3] and measured its miss ratio compared to exactly implemented LFU and LRU caches of the same size. Figure 4.1 shows the results as obtained over the duration of the `w96` trace in the Cloudphysics dataset [24]. At time 40 hours, the Redis eviction policy is switched from LFU to LRU. The miss ratio of the Redis cache resembles that of LFU before 40 hours with a period of high error between hours 18 and 40, due to its approximate implementation of LFU. After 40 hours, the miss ratio of the Redis cache follows that of LRU with an error of roughly 5%.[4]

4.1.2 SHARDS sampling

Waldspurger et al. describe a sampling technique called *SHARDS*, originally designed for use in efficient MRC generation [24] as described in §2.2.2. SHARDS samples a stream of incoming cache accesses using a configurable sampling rate R. On each access, SHARDS computes $T_i = hash(K) \bmod P$, where K is the access key and P is a large static number (the authors use $P = 2^{24}$). If $T_i < RP$, the access is sampled. An important characteristic of SHARDS is that once a key is sampled, it will always be sampled. (This corresponds to fixed-rate SHARDS. We do not describe fixed-sized SHARDS here because PaperCache does not use that.)

[2]This method would allow Redis to support some other eviction policies with minimal changes (e.g., in the case of FIFO, maintaining the object's entry time into the cache). However, adapting this eviction strategy to more complex policies (e.g., ARC [50], LHD [11], LIRS [63], 2Q [20], S3-FIFO [10], etc.) would likely be challenging.

[3]We use `allkeys-lfu` for LFU and `allkeys-lru` for LRU.

[4]Interestingly, repeating this experiment, we noticed different results each time, which can be attributed to Redis' eviction policy approximation (i.e., randomly selecting N objects as candidates for eviction).

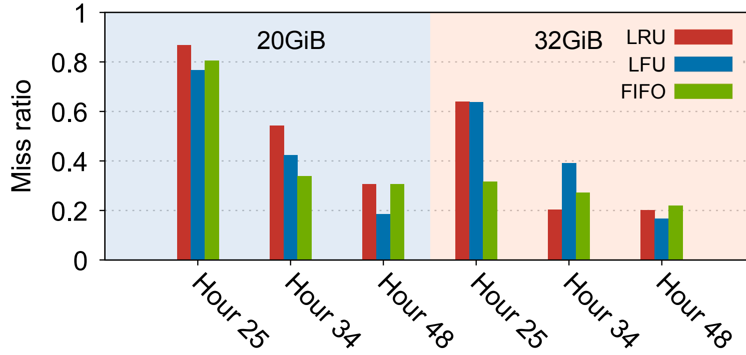


Figure 4.2: Hourly LRU, LFU, and FIFO miss ratios for the Cloudphysics `w24` trace [24] for caches of size 20GiB and 32GiB.

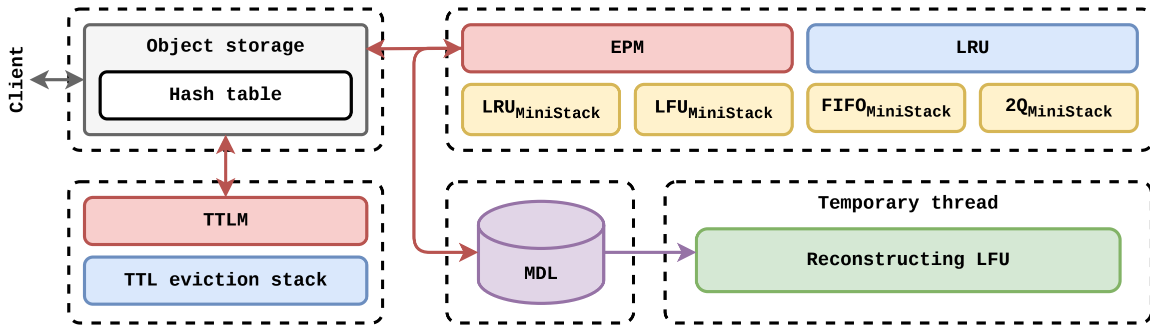


Figure 4.3: An overview of PaperCache (LRU). Sections in dashed rectangles run on separate threads.

4.1.3 Selecting the optimal eviction policy

The optimal eviction policy for a workload – i.e., the eviction policy with the lowest miss ratio – can change over time. Figure 4.2 shows the miss ratios for the LRU, LFU, and FIFO eviction policies for the Cloudphysics `w24` trace [24] for caches of size 20GiB and 32GiB. For a cache of size 20GiB, at 25 hours, the optimal eviction policy is LFU. At 34 hours, this changes to FIFO. Finally, at 48 hours, this changes back to LFU. Selecting an eviction policy is further complicated by the fact that the optimal eviction policy depends on the cache's allocated size. For the same Cloudphysics `w24` trace, using a cache size this time of 32GiB, at 25 hours, the optimal eviction policy is FIFO, at 34 hours, this changes to LRU, and finally at 48 hours, it changes to LFU. In §4.3.7, we show that this behavior is present in most of the workloads we evaluated – we found that in 86.8% of the workloads, a cache allocated 10% of the workload's WSS observed at least 2 switches of the optimal eviction policy over the duration of the access trace.

4.2 PaperCache

In §4.1.3, we showed that the selection of the appropriate eviction policy on a per-workload basis can have significant benefits on performance. In this section, we present PaperCache, a novel in-memory cache that can dynamically switch between any eviction policy at runtime. Figure 4.3 depicts PaperCache's architecture. It has three notable design elements. First, PaperCache maintains a full stack

of object metadata for the currently active eviction policy (e.g., LRU in the figure). This stack is used to perform evictions (exactly) according to the active policy, and hence the stack would be ordered by access recency for LRU, access frequency for LFU, and so on. The stack is updated and cache evictions are handled asynchronously by an *eviction policy manager* (**EPM**) running on a separate worker thread.

Second, for each configured eviction policy, PaperCache maintains an approximate stack of metadata, referred to as a *MiniStack*.[5] The SHARDS fixed-rate sampling algorithm [24] is used to decide which objects are represented in the MiniStacks, thus significantly reducing the MiniStacks' memory overheads. A MiniStack is used to temporarily decide which objects to evict, if necessary, while the cache is switching policies.

Finally, metadata of each access is streamed to disk in a *metadata log* (**MDL**). Information in the MDL is used to reconstruct the full stack of a target eviction policy when PaperCache switches policies.[6] This reconstruction is performed by a separate temporary thread. The MDL contains a sliding window of access metadata (default = 7 days) to limit stack reconstruction time.

When an existing object is accessed, or when a new object is added to the cache, the object's metadata is asynchronously sent to the EPM which then: (i) updates the full stack of the current eviction policy, (ii) updates all MiniStacks if sampled, and (iii) pushes the access metadata to the MDL. The EPM also asynchronously evicts objects from the cache (along with their metadata) when the cache's used size exceeds its configured size. Similar to prior techniques [9], PaperCache performs evictions lazily, which may cause short periods of time wherein the cache's used size exceeds its configured size (although we found this to be negligible when processing real-world access traces, which we demonstrate in §4.3.2). However, performing these operations asynchronously reduces access latencies as eviction policy stack operations occur off the main thread serving client accesses.

When the eviction policy is switched from policy P to policy P', PaperCache performs the following actions:

1. Pauses writes to the MDL and buffers subsequent access' metadata in memory.
2. Starts to use the MiniStack of P' for eviction decisions.
3. Releases the full stack of P from memory.
4. Reconstructs the full stack of P' using the MDL.
5. Starts to use the full stack of P' for eviction decisions.
6. Flushes buffered access' metadata to P' and the MDL, and resumes subsequent writes to the MDL.

These actions are performed asynchronously while the cache is servicing new accesses. If cache evictions must occur while the stack is being populated (i.e., before step 5), objects are evicted using the MiniStack of P'. The cache therefore has a short period of approximate behavior until

[5]We note that a MiniStack does not directly relate to an MRC generation algorithm, such as MiniSim; however, a MiniStack is equivalent to a single cache simulation within MiniSim. This will prove useful in allowing PaperCache to automatically determine if it is beneficial to switch eviction policies, described later.

[6]In the common case, we only store each access's key as its metadata in the MDL, however, some eviction policies may require more information about each access to perform reconstruction (e.g., the size of the access). In such cases, all required metadata of each access is saved to the MDL.

the full stack is reconstructed. In §4.3.6, we show that a cache subject to evictions from a policy's MiniStack exhibits a miss ratio within 1% of that when it is subject to the policy's full stack for a period of up to 2.7 hours, on average.

To handle TTLs, PaperCache uses a priority queue of object expiry times. This queue is managed by a separate time-to-live manager (**TTLM**) running on its own thread. The queue is checked every millisecond, and expired objects are removed from the cache and the MiniStacks. (Expired objects are never returned to clients.)

Automatically switching eviction policies

Each PaperCache MiniStack is analogous to a single simulation in MiniSim [48]. Waldspurger et al. showed that the miss ratio M of a MiniStack subject to a sampling rate R can be accurately calculated as the ratio between the observed number of misses to the expected number of accesses (i.e., $M = N_{misses}/(R * N_{total})$), where N_{misses} is the observed number of misses and N_{total} is the total number of accesses [48]. The MiniStacks can therefore be used, at no additional cost, to periodically identify which eviction policy incurs the lowest miss ratio. This allows PaperCache to support *auto eviction policy switching* where it automatically switches to the eviction policy with the lowest miss ratio. PaperCache can therefore be configured to support many eviction policies, including multiple configurations of the same eviction policy for policies which require configuration parameters (e.g., 2Q [20] or S3-FIFO [10]), and will automatically select the eviction policy with the lowest miss ratio. We found that identifying the policy with the lowest miss ratio, and if different than the current active eviction policy, switching to that eviction policy every 1 hour is effective.

Implementation

PaperCache was implemented in Rust v1.89.0 and uses jemalloc as its memory allocator.[7] Our implementation of PaperCache is open-sourced at `https://papercache.io`.

4.3 Evaluation

In this section, we evaluate PaperCache by examining:

(i) the accuracy of its policy implementations compared to ideal implementations after switching between policies (§4.3.1),

(ii) the ability of its lazy eviction scheme to keep the memory consumed by the cached objects within the configured cache size (§4.3.2),

(iii) its metadata memory overhead (§4.3.3) and latency performance (§4.3.4) compared to Redis,

(iv) its CPU usage when switching between eviction policies (§4.3.5),

(v) the duration for which MiniStacks provide reasonably accurate policy evictions (§4.3.6), and

(vi) its behavior when automatically switching eviction policies (§4.3.7).

In all experiments, PaperCache is configured with 8 eviction policies: LFU, FIFO, LRU, MRU, 2Q [20], S3-FIFO [10], CLOCK, and SIEVE [51].[8] We compare PaperCache to Redis in our evalu-

[7] PaperCache uses the same version of jemalloc as Redis v8.0.1.
[8] We use $K_{in} = 0.25$ and $K_{out} = 0.5$ for 2Q, and $|S| = 10\%$ for S3-FIFO.

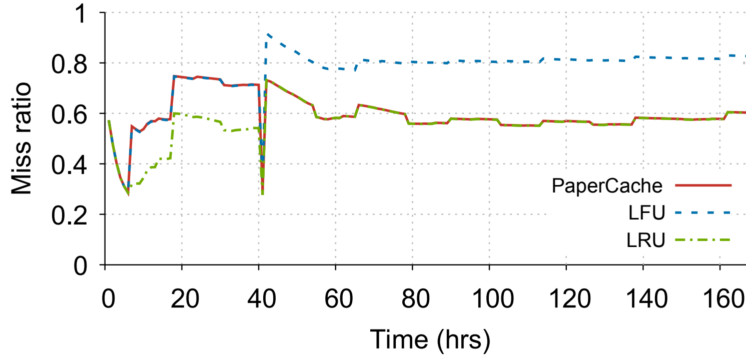


Figure 4.4: Miss ratios of LFU, LRU, and PaperCache switching from LFU to LRU at 40 hrs for the Cloudphysics `w96` trace [24].

ation as it is, to our knowledge, the only in-memory cache that supports eviction policy switching. For our evaluations, we used 873 access traces from Twitter [3], Cloudphysics [24], IBM [25], Tencent CBS [26], Alibaba [27], and Wikipedia [28] with over 189 billion total accesses.[9]

All experiments were done on Ubuntu 24.0.2 with an Intel i9-13900KS (24 cores) with 128GB of DDR5–4200MHz DRAM.

4.3.1 Policy switching accuracy

To examine the effectiveness of PaperCache's policy switching, we configured an instance of PaperCache of size 500MiB with the LFU eviction policy and show its performance compared to exact LFU and LRU caches of the same size for the `w96` trace in the Cloudphysics dataset [24], similar to the experiment we performed for Redis in §4.1.1. At time 40 hours, we switch PaperCache's policy from LFU to LRU and reset the hit and miss counters of the caches. Figure 4.4 shows the results. We observe that before 40 hours, PaperCache closely tracks the miss ratio of LFU. After 40 hours, it switches to LRU and immediately closely tracks the miss ratio of LRU as well. (Compared to the results obtained from Redis and shown in Figure 4.1, PaperCache tracks the miss ratios of LFU and then LRU far better than Redis.) As PaperCache's eviction policy switching occurs off the main thread handling client accesses, we noticed no measurable effects on access throughput or latency when switching the eviction policy at runtime.

4.3.2 Lazy eviction performance

PaperCache's lazy eviction scheme (where evictions occur off the main thread) may result in short periods of time wherein the memory used by cached objects exceeds the configured maximum cache size (as `SET` accesses are not blocked for evictions). Here, we evaluate the resulting overhead of using lazy evictions. We initialized a PaperCache instance of size 2GiB and configured it to use the LRU eviction policy. We then applied the accesses in the Cloudphysics `w96` access trace [24]. We measure the aggregate size of objects in the cache versus the access trace's working set size (**WSS**) over the duration of the experiment, where the WSS is the aggregate size of all unique objects accessed thus far. Figure 4.5 shows the results. Measured after the WSS exceeds 2GiB, PaperCache's used size is,

[9] We use the recommended traces in the Twitter dataset [3, 98].

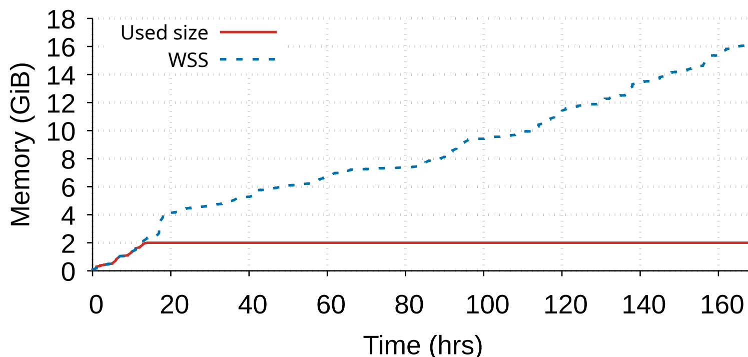


Figure 4.5: PaperCache used size versus WSS for the Cloudphysics `w96` trace [24].

Table 4.1: Memory overheads of Redis and PaperCache storing between 1,000 and 1,000,000 unique objects.

	1,000	10,000	100,000	1,000,000
Redis v8.0.1	15MiB	43MiB	176MiB	270MiB
PaperCache	12MiB	53MiB	195MiB	459MiB

on average, within 0.01% of the configured maximum size, up to a maximum of 0.15%. We repeated this experiment for several other access traces and found similar results.

4.3.3 Memory overhead

PaperCache's support of multiple eviction policies and the switching between them comes at the cost of higher memory overhead to maintain the full metadata stack (for the currently active eviction policy) and N MiniStacks, where N is the number of configured eviction policies. To measure the extent of this overhead, we compare PaperCache with Redis v8.0.1 [16], both configured to use 2GiB and LRU. We inserted between 1,000 and 1,000,000 unique objects, using object sizes such that the cached objects consume 1GiB in aggregate (so no evictions would occur), and then measured the high water mark (**HWM**) of the cache's residency set size. Table 4.1 shows the overheads as the HWM minus 1GiB.[10] PaperCache has between 10.8% and 70% higher memory overhead than Redis when storing between 100,000 and 1,000,000 unique objects, respectively, with 8 configured MiniStacks and a MiniStack sampling rate of 0.1%. Interestingly, we found that only roughly 1.6% of the memory overhead is attributed to storing the MiniStacks. We note that PaperCache uses slightly less memory than Redis when storing 1,000 objects which is attributed to smaller initial allocations for its data structures.

4.3.4 Latency performance

We experimentally compare the performance of PaperCache and Redis by measuring the latencies of `GET` and `SET` accesses. For this, we instantiated PaperCache and Redis caches of size 2GiB under

[10]These results demonstrate a rough metric for PaperCache's memory overhead compared to that of Redis. Because these caches do not have similar implementations and rely on different internal data structures (e.g., different hash table implementations), the presented metrics may not fairly compare the caches. To perform a fair comparison, PaperCache's eviction policy strategy may be implemented using the publicly available Redis codebase [16], though we leave this for future work.

Table 4.2: Access latency percentiles of Redis and PaperCache for all traces in the Cloudphysics dataset [24].

	p99 (μs)		**p99.9** (μs)		**p99.99** (μs)	
	GET	**SET**	**GET**	**SET**	**GET**	**SET**
Redis v8.0.1	116	595	298	1,327	600	1,889
PaperCache	66	523	177	806	417	1,070

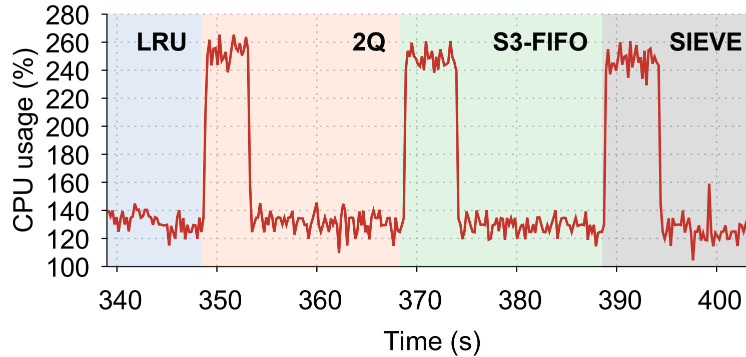


Figure 4.6: Total CPU usage of PaperCache (initially under LRU) when switching to the 2Q, S3-FIFO, and SIEVE eviction policies for the Cloudphysics `w07` trace [24]. Note that 100% CPU usage represents the full utilization of a single processor core.

LRU and performed the accesses in all access traces in the Cloudphysics dataset [24] (106 access traces comprising over 2 billion total accesses) using 4 concurrent clients for each cache issuing cache accesses as quickly as possible. Table 4.2 shows the p99, p99.9, and p99.99 percentile access latencies. We found that PaperCache has 30.5% and 43.4% lower p99.99 latencies for `GET` and `SET` accesses, respectively. These lower latencies are due to evictions in PaperCache occurring off the main thread.

4.3.5 CPU usage

We examine PaperCache's increased CPU usage while reconstructing the stack of a new policy by performing policy switches when processing the `w07` trace in the Cloudphysics dataset [24]. Figure 4.6 shows the total CPU usage of PaperCache (configured with 64GiB of memory) as it switches from LRU to 2Q, to S3-FIFO, then to SIEVE. The CPU usage presented in this figure includes that of all of PaperCache's active threads. During periods where PaperCache is not reconstructing an eviction policy's stack, we notice the CPU usage is typically between roughly 120% and 140%. We attribute this CPU usage to the main thread handling cache accesses, the thread sampling the accesses and maintaining the MiniStacks, and the thread processing each access' metadata to be stored in the MDL. These findings are in-line with our theoretical expectations and when repeating this experiment on other access traces and with different eviction policies, we found similar results. Although PaperCache has increased CPU usage when a policy stack is being reconstructed after a policy switch, we note that the extra CPU overhead occurs off the main thread, so there is no impact on access latency.

To examine the effects of switching eviction policies on Redis, we repeat the experiment on a Redis cache configured with 64GiB of memory as it switches from LRU to LFU when processing the

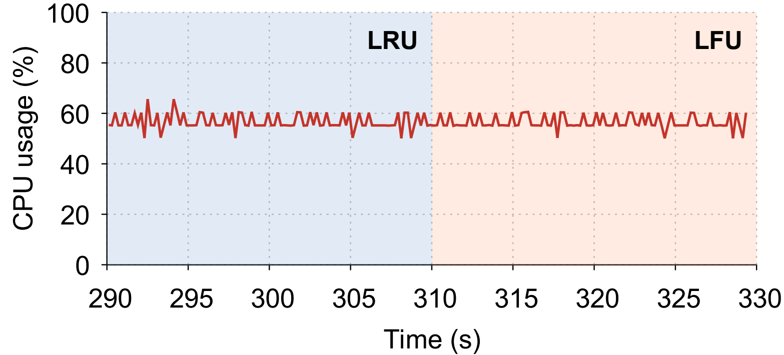


Figure 4.7: Total CPU usage of Redis (initially under LRU) when switching to the LFU eviction policy for the Cloudphysics `w07` trace [24]. Note that 100% CPU usage represents the full utilization of a single processor core.

`w07` trace in the Cloudphysics dataset [24]. Figure 4.7 shows the total CPU usage of Redis from 20s before to 20s after the policy switch. As Redis performs its policy switching lazily (an object's policy metadata is updated only after it is reaccessed), we notice no increase in CPU usage immediately after a policy switch.

4.3.6 MiniStack efficacy duration

While there is an ongoing eviction policy switch, eviction decisions are made using the MiniStack of the newly configured eviction policy. As a result, the miss ratio of the cache can deviate from that of the exact eviction policy because only objects represented in the MiniStack are selected for eviction and the MiniStack only contains objects that were sampled.

To evaluate the efficacy of a MiniStack in approximating the evictions of its corresponding policy, we initialized two LRU caches of equal size: one subject to evictions from a full LRU stack and the other from an LRU MiniStack. We applied the same access trace to each cache and measured the duration between when the first eviction occurred to when the miss ratios of the two caches differed by $\geq$ 1%, which we refer to as the MiniStack's *efficacy duration*. Each cache was allocated 10% of the access trace's WSS.

Figure 4.8 demonstrates the results of this experiment for 873 workloads that include over 189 billion accesses. We found the average and median efficacy durations to be 2.7 and 1.1 hours, respectively. The largest access trace is Twitter's `cluster18` trace [3] which includes 12.6 billion accesses. Experimentally, we found that PaperCache reconstructs a full stack at a rate of 18.2 million accesses per second, on average, and thus PaperCache reconstructs the full stack for `cluster18` in 11.5 minutes. As this is significantly less than the median MiniStack efficacy duration, PaperCache's miss ratio will track that of exact implementations of its eviction policies during a policy switch reasonably accurately.

In our findings, we observed some access traces exhibit low MiniStack efficacy durations, which we consider to be outlier cases. Table 4.3 summaries examples of these cases from the Twitter, Cloudphysics, IBM, and Alibaba datasets. The Twitter `cluster15` access trace has a MiniStack efficacy duration of < 1 second; however, 98.5% of its accesses are to unique objects and therefore any cache under this workload will have a considerably high miss ratio, regardless of its eviction

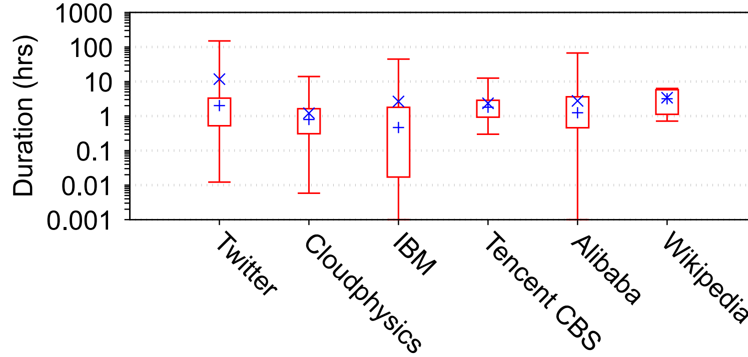


Figure 4.8: MiniStack efficacy durations for all considered datasets. The bottom and top lines identify the minimum and maximum results, respectively. The bottom and top of each box are the 25th and 75th percentile results, respectively. The $\times$ and $+$ symbols indicate the mean and median results, respectively. Note the logarithmic scale of the y-axis.

Table 4.3: Access traces with relatively small efficacy durations which we consider outlier cases.

Workload	**MiniStack efficacy duration**	**Unique accesses**	**Total accesses**
Twitter `cluster15`	< 1 second	6,290,874	6,387,217
Cloudphysics `w82`	183 seconds	75,902	4,926,610
IBM `ibm79`	156 seconds	12,585	202,751
Alibaba `ab367`	3 seconds	91,907	1,400,000

Table 4.4: % time each policy achieves the lowest and strictly lowest miss ratios, where "strictly lowest" means it is not tied, for the Cloudphysics `w42` access trace [24] when the cache is configured to be of size 10% of the trace's WSS.

Policy	**% time min. MR**	**% time strictly min. MR**
LFU	43.2	29.6
FIFO	10.1	0
LRU	11.8	0
MRU	24.9	13
2Q	10.1	0
S3-FIFO	42	31.4
CLOCK	11.9	0
SIEVE	25.4	12.4

policy. In the case of the Cloudphysics `w82`, IBM `ibm79`, and Alibaba `ab367` access traces which have relatively few accesses to unique objects, the MiniStack sampling rate of 0.1% (which we used in our evaluation) will lead to the MiniStacks containing few entries. Increasing the MiniStack sampling rate in such cases will result in higher MiniStack efficacy durations. In cases where this is not possible, the cache will temporarily notice degraded performance during an eviction policy switch, until the stack of the newly configured eviction policy is fully reconstructed.

4.3.7 Automatic policy switching behavior

We also evaluated PaperCache's ability to automatically switch eviction policies. For this, we configure PaperCache to switch every 1 hour to the corresponding policy of the MiniStack with the lowest miss ratio over the previous hour. We used the Cloudphysics `w42` access trace [24] which

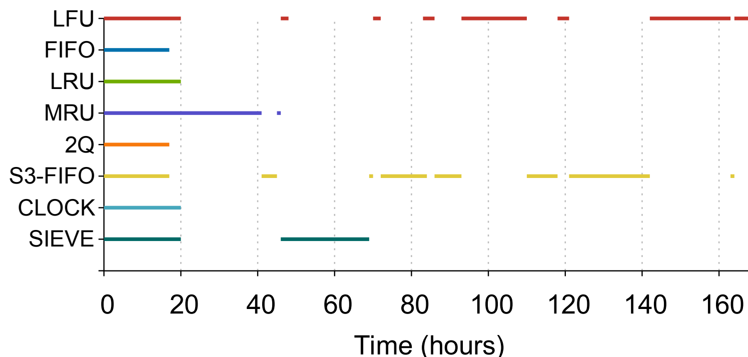


Figure 4.9: Instances where each configured PaperCache eviction policy achieves the lowest miss ratio for the `w42` trace in the Cloudphysics dataset [24].

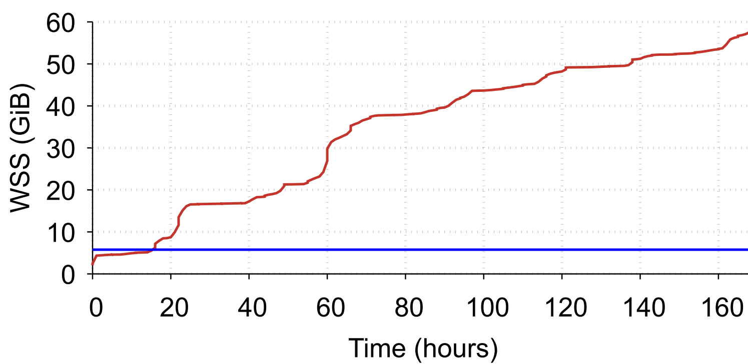


Figure 4.10: WSS over time for the `w42` trace in the Cloudphysics dataset [24]. The blue horizontal line is the configured cache size.

spans 168 hours, and we configured PaperCache with a size of 10% of the trace's WSS (this trace has a WSS of 57.7GiB). Throughout this access trace, PaperCache (initially under LRU) performs 17 policy switches, reducing the miss ratio compared to LRU by up to 9.7%. Table 4.4 shows the percentages of the total trace duration where each eviction policy achieves the lowest and strictly lowest miss ratio (where achieving the "strictly lowest" miss ratio indicates it achieves a miss ratio lower than that of any other eviction policy). The LFU eviction policy has the best performance for the largest duration of the access trace, achieving the lowest and strictly lowest miss ratio for 43.2% and 29.6% of the total trace duration, respectively.

Figure 4.9 shows the instances where each policy achieves the lowest miss ratio over the duration of the trace. We note that in Figure 4.9, all eviction policies have equal performance for the first 16 hours of the access trace. This is because no evictions occur during that period. The reason for this is shown in the first 16 hours of Figure 4.10 that depicts the WSS over time for the duration of the trace. The blue horizontal line shows the configured cache size (10% of the maximum WSS). The WSS starts to exceed the configured cache size at time roughly 16 hours.

Between times 20 hours and 165 hours, PaperCache alternates between the LFU and other eviction policies. To further examine this behavior, Figure 4.11 shows the Zipfian α over time for the duration of the trace. At hours 19, 47, 70, 94, 118, 142, and 166, we notice increases in the trace's Zipfian α. These times correlate to times at which PaperCache switches to the LFU eviction

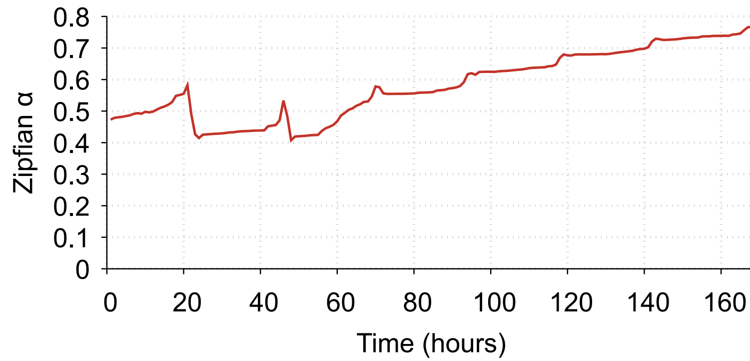


Figure 4.11: Zipfian α over time for the `w42` trace in the Cloudphysics dataset [24].

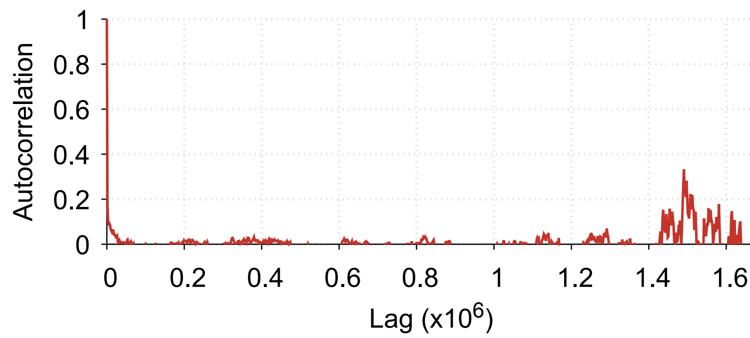


Figure 4.12: Autocorrelation coefficient between hours 20 and 41 for the `w42` trace in the Cloudphysics dataset [24].

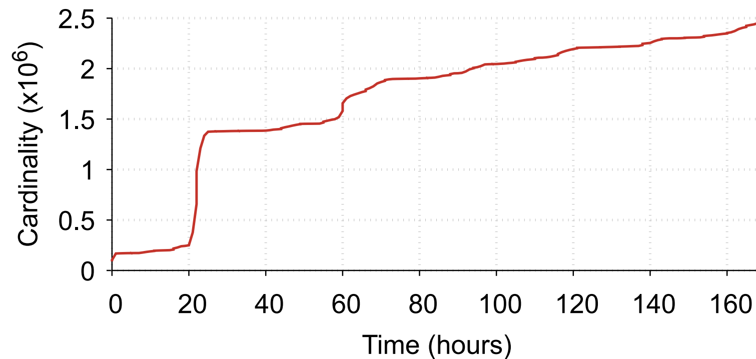


Figure 4.13: Cardinality over time for the `w42` trace in the Cloudphysics dataset [24].

policy (or is already configured to the LFU eviction policy in the case of hour 19). These results match previous works' findings which indicate that the LFU eviction policy excels when an access pattern follows a Zipfian distribution [30, 59].

Between times 20 hours and 41 hours, PaperCache switches to the MRU eviction policy. Prior work has suggested that the MRU eviction policy excels under cyclical workloads [58]. To examine the behavior of the workload during this period, we examine the autocorrelation coefficient, shown in Figure 4.12. During this period, 1,675,976 accesses occur, and we therefore find the autocorrelation

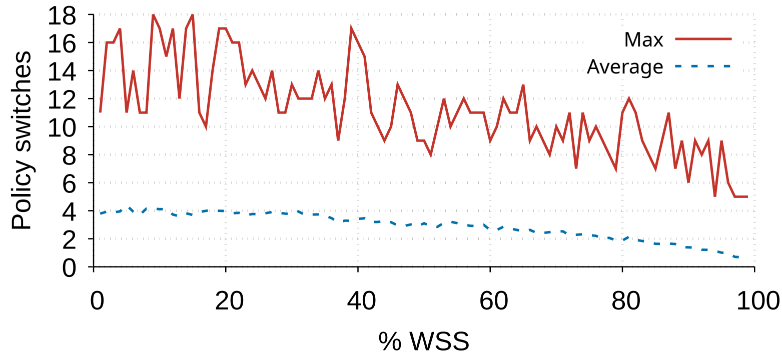


Figure 4.14: Number of eviction policy switches (initially LRU) versus cache size (as a percentage of the workload's WSS) for all traces in the Cloudphysics dataset [24].

coefficient for lag values in the range $[0, 1{,}675{,}976)$. The autocorrelation coefficient peaks at roughly 0.33 for a lag value of roughly 1,490,750 (we exclude the initial peak which occurs for small lag values). The consistently low autocorrelation coefficients and the relatively low peak of 0.33 indicates that there is not a strong cyclic access pattern during this time period. Interestingly, examining the cardinality of the accesses over time (shown in Figure 4.13) shows that at roughly 20 hours, there is a large number of accesses to unique objects, indicating a scanning pattern. Here, one would expect an eviction policy specifically designed for scanning pattern resistance (e.g., 2Q or S3-FIFO) to have the lowest miss ratio. However, we find that MRU outperforms these policies. These findings indicate that the "rules of thumb" used to select an eviction policy based on prior information of a workload's access patterns may not always result in the selection of the most performant eviction policy.

We found the behavior of the Cloudphysics `w42` workload to be representative of many workloads. Figure 4.14 shows the average and maximum number of eviction policy switches performed by PaperCache over the durations of all traces in the Cloudphysics dataset [24]. To examine the effects of the size of the cache on the number of policy switches, we vary the size of the cache from 1% to 99% of each access trace's WSS. We found that the maximum number of observed policy switches was between 5 and 18. A point of interest in this figure is when the average number of policy switches is ≤ 1, which we observed to be when the cache is sized $\geq 96\%$ of the WSS. This indicates that caches of these sizes will, on average, not observe miss ratio benefits from performing policy switches. Intuitively (and experimentally observed), as the cache size increases, the expected number of policy switches decreases as fewer evictions occur and different policies have more similar miss ratios. A cache of size $\geq WSS$ will not switch eviction policies as no evictions occur in the cache.

Although the average number of evictions for caches of size $\geq 96\%$ of the WSS is ≤ 1, to examine the effect of varying the cache size on noticing a miss ratio reduction from an eviction policy switch, we measure the ratio of traces in the Cloudphysics dataset [24] which observe at least 2 policy switches for cache sizes between 1% and 99% of each access traces's WSS. Figure 4.15 shows the results. We found that 20.8% of access traces with caches of size 96% of the WSS observe at least 2 policy switches. Further, we found 17% of access traces with caches of size 99% of the WSS also observe at least 2 policy switches. These findings indicate that in a significant number of cases, there is an observable benefit to dynamically modifying the cache's eviction policy as the ideal eviction

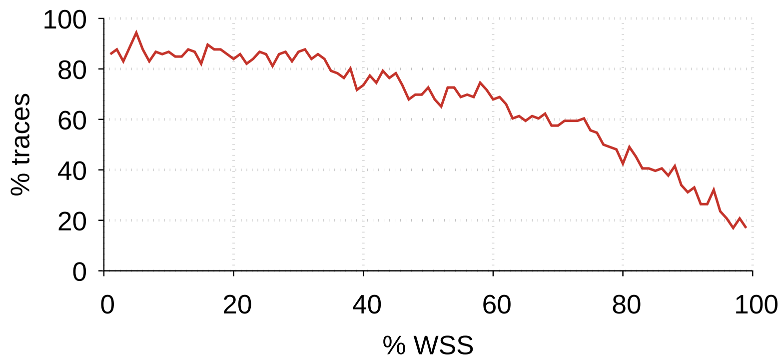


Figure 4.15: % traces in the Cloudphysics dataset [24] which observe at least 2 policy switches (initially LRU) as a function of the cache size (percentage of the workload's WSS).

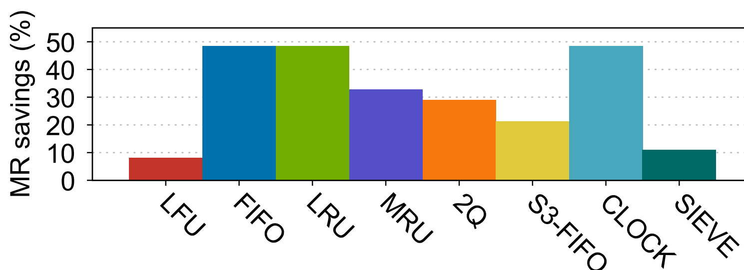


Figure 4.16: Miss ratio savings by performing eviction policy switching compared to statically assigned eviction policies for the Cloudphysics `w02` trace [24].

policy cannot be statically configured when the cache is first initialized.

PaperCache's automatic eviction policy switching can considerably reduce the miss ratio of the cache. Figure 4.16 shows the miss ratio savings achieved by PaperCache when performing automatic eviction policy switching compared to statically configured eviction policies for the `w02` access trace in the Cloudphysics dataset [24]. We configured PaperCache with an allocated size of 10% of the access trace's WSS (this trace has a WSS of 19.5GiB). PaperCache reduced its miss ratio by between 8.2% and 48.5%. These results demonstrate the maximum miss ratio reduction we observed by PaperCache which we present as the user of the cache will always notice either a decrease or no change in the miss ratio as a result of PaperCache's automatic eviction policy switching.

4.4 Conclusion

In this chapter, we introduced PaperCache, a novel in-memory cache design which supports the dynamic switching between any eviction policy at runtime. We demonstrated that a workload's optimal eviction policy can change over time, reinforcing the need for PaperCache. We introduced our novel eviction policy switching technique through the use of MiniStacks and show how they can be leveraged to perform automatic eviction policy switching, which we showed can reduce PaperCache's miss ratio by up to 48.5%. As future work, we plan to reduce PaperCache's memory overhead and extend it to support admission policies.

Chapter 5

Flux: Multi-Cache, Multi-Host Online Orchestration

Online MRC algorithms, such as Kosmo (Chapter 3) allow for the real-time, detailed understanding of the effects of modifying a cache's configuration parameters, such as its size and eviction policy, on its miss ratio. In-memory caches, such as PaperCache (Chapter 4) can modify such configuration parameters in real-time. However, to realize the benefits of such tools, one must have an online cache orchestrator which can use MRCs to periodically inform decisions about the configurations of in-memory caches.

The primary thesis of this chapter is that the resource footprint of a cache fleet can, in many circumstances, be reduced considerably, without significantly affecting miss ratios. The secondary thesis is that this can be done in large part automatically, by periodically adjusting the configurations of the caches in the fleet according to the needs of their workloads. In pursuit of these two theses, we developed *Flux*, the first online multi-host cache fleet orchestrator. The primary objective of Flux is to reduce the amount of memory used in aggregate while maintaining cache hit ratios, with a secondary objective to reduce the number of hosts required to run the caches being managed.

Flux assigns each new cache being instantiated to a specific host, selected based on the amount of unallocated memory available at the hosts, with multiple caches potentially assigned to the same host. Flux periodically (re-)configures the caches it manages with respect to cache size, cache eviction policy, and cache (co-)location. Flux may decide to migrate individual caches, either to accommodate increasing memory requirements of co-located caches or to consolidate the number of hosts needed. Reconfiguration is driven by the current demands of each cache's workload. We cast this cache fleet orchestration problem as a constrained optimization problem [111–113].

Flux focuses on in-memory caches that run as independent processes (e.g., within Docker containers), aligning with the design of widely used systems like Memcached and Redis. Flux is designed to excel in two primary deployment environments: (i) public cloud platforms, where fleets of hosts are dedicated to serving caches and cache instances can be flexibly instantiated or retired based on client demand; and (ii) corporate datacenters, which operate cache fleets ranging from a few dozen to several thousand nodes. Flux is well suited for workloads where dynamic cache reconfiguration can deliver meaningful performance benefits. For workloads that are predominantly memory-bound, traditional static approaches may already perform well.

We argue that Flux constitutes a major improvement in the state of the art, which consists in many cases of manual static configuration of individual caches at the time they are instantiated. Flux instead dynamically adapts to changes in workload and makes decisions for the entire cache fleet, not for an individual cache. In doing so, it can in many cases avoid the overprovisioning that is common with static manual configuration, while maintaining cache miss ratios.

Our design of Flux was motivated by the findings of a comprehensive analysis of over 900 publicly available, real-world cache access traces from Twitter [3], Cloudphysics [24], IBM [25], Tencent CBS [26], Alibaba [27], and Wikipedia [28] containing over 346 billion accesses, with a combined total duration of over 55 years of cache operations. In §5.1, we present our findings and show for example that:

1. ***Most workloads require cache sizes substantially lower than the typical amount of DRAM available in today's hosts to achieve their minimal possible miss ratios.*** In fact, a majority of the workloads require less than 32GiB of cache memory to achieve their minimal possible miss ratios. This implies that multiple cache instances can often be run simultaneously on a single host, potentially reducing the number of hosts needed.

2. ***Adjusting cache sizes to the needs of their workloads every hour results in a 36% lower aggregate memory footprint on average*** compared to statically setting the cache size to the workloads' working set size (which results in the minimal possible miss ratio).[1] In the extreme, the savings can be as high as 95% for some workloads.

3. ***Switching a cache's eviction policy periodically can lead to a miss ratio that is up to 60% lower compared to statically configuring the eviction policy when the cache is instantiated.*** In fact, every workload we considered benefited from switching eviction policies dynamically at run time. Further, switching the eviction policy can also reduce the required cache size over an interval of time; e.g., for a few workloads the average memory savings achieved by dynamically switching eviction policies can be as high as 50%.

A key implication of the above findings is that the standard practice of configuring the size and eviction policy of each cache at instantiation, and keeping the configuration static throughout the cache's lifetime, is in many cases suboptimal in terms of resource consumption. Instead, periodically reconfiguring the caches can lead to significant improvements in efficiency.

Our work on Flux follows the footsteps of prior work on cache orchestration systems [73, 78, 79, 81]. Flux differentiates itself from these prior systems in that (i) it is capable of managing a fleet of caches assigned to a fleet of hosts while previous ochestrators only support single hosts; (ii) considers additional resources beyond just memory (e.g., network bandwidth); (iii) uses a lighter-weight heuristic algorithm (instead of, say, an exhaustive search algorithm such as dynamic programming) to configure the caches, which improves scalability and enables caches to be reconfigured more frequently (e.g., every hour); and (iv) is able to support more complex objective functions.

Our evaluation on real-world workloads in §5.4 demonstrates Flux's ability to significantly reduce resource requirements. In comparison to a fleet of caches statically configured according to their working set and with Least Recently Used (LRU) as their eviction policy, the same fleet of caches managed by Flux uses 55%-95% less memory and 14%-79% fewer hosts with marginal effects on

[1] Although this appears to be an extreme option, caches are often initially configured to be significantly larger than their actual WSSes for fear of degraded performance.

Table 5.1: Access trace datasets used in our analysis.

Dataset	# workloads	Dataset	# workloads
Twitter [3]	54	Cloudphysics [24]	106
IBM [25]	98	Tencent [26]	40
Alibaba [27]	609	Wikipedia [28]	3

the cache miss ratios. We also demonstrate Flux's ability to manage the dynamic instantiation and removal of caches during operation.

Contributions

This chapter makes the following contributions:

1. We present the results of our analysis of 900+ publicly available cache access traces, focusing on dynamic cache resizing and eviction policy switching. (§5.1)
2. We introduce the design and implementation of Flux, the first in-memory cache orchestration system capable of dynamically adjusting the configurations of a set of caches running on a set of hosts. (§5.3)
3. We evaluate Flux with a number of case studies and show its effectiveness in conserving memory and host usage with a marginal impact on the caches' miss ratios. (§5.4)

Limitations

Our work has the following limitations:

1. Our analysis is based only on publicly available workloads and hence implications drawn from the analysis may not be valid for other real-world workloads.
2. The in-memory cache workloads of some organizations (e.g., Meta [31]) are primarily memory-bound where dynamic cache configuration adaptations are unlikely to be effective, so Flux will not be of much use in such cases.
3. Flux uses a heuristic optimization algorithm to configure caches running on a fleet of hosts, and hence does not claim to be optimal.

5.1 Analysis of Caching Workloads

In this section we present the results of a comprehensive analysis of over 900 real-world cache workloads represented by publicly available cache access traces as they relate to cache sizing and choice of eviction policy. The need for Flux, and its design, was motivated by this analysis. Table 5.1 lists the traces we used in our analysis. They contain over 346 billion cache accesses and represent over 55 years of cache operations in aggregate. None of the workloads used for our analysis has a duration of less than a week.

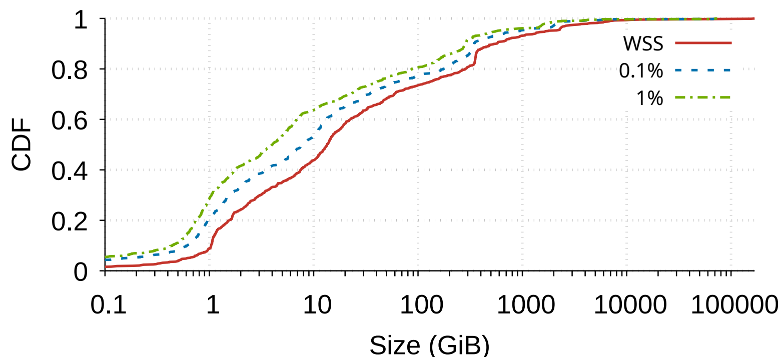


Figure 5.1: CDF of the WSS and the minimum cache sizes required to achieve a miss ratio increase of at most 0.1% and 1% for all traces in Table 5.1. Note the logarithmic scale of the x-axis.

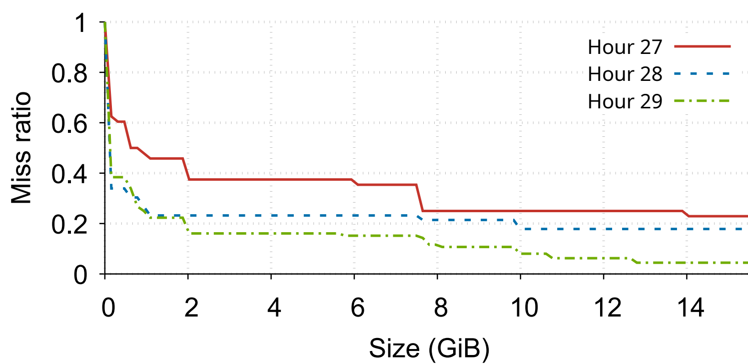


Figure 5.2: MRCs for the LRU eviction policy for three consecutive hours of the `w77` trace in the Cloudphysics dataset [24].

5.1.1 Working set size analysis

If a cache's size is statically configured and set to the target workload's WSS, then the cache will achieve the *minimal possible miss ratio* (**MPMR**). Our first, perhaps somewhat surprising finding was that many cache workloads require cache sizes substantially lower than the 210GiB/318GiB/419GiB of physical memory available in today's hosts.[2] For example, if the size of each cache were set to its workload's WSS — thus guaranteeing the minimum possible miss ratio — then 63% of the cache sizes for the 900+ workloads we considered would be under 32GiB. The red curve in Figure 5.1 depicts the *cumulative distribution function* (**CDF**) of the cache sizes needed to achieve the MPMR for each workload.

Insight. Multiple cache instances can often be run simultaneously on a single host, potentially reducing the number of hosts used.

5.1.2 Miss ratio curve analysis

An analysis of the MRCs for the 900+ workloads we considered indicates that if a slight increase in a workload's miss ratio is acceptable, then the required cache size for each workload is often

[2]These values were obtained from AWS's hosted Redis offerings [36].

substantially lower than the workload's WSS. For example, for a target miss ratio of just 0.1% higher than the minimum possible miss ratio, a workload's cache will need 31.3% less memory than the workload's WSS, on average. Moreoever, the cache can be sized 51.2% smaller on average if a 1% higher miss ratio is acceptable. This is because the miss ratio curves of many workloads tend to have long tails with a slight negative slope before reaching the minimum steady-state (when the WSS is reached). The CDF of the cache sizes required to achieve a miss ratio 0.1% higher than the minimal possible miss ratio across all 900+ workloads is depicted as the blue dashed line in Figure 5.1, and the CDF of the cache sizes required to achieve a miss ratio 1% higher than the MPMR is depicted as the green dotted line.

If a workload's cached objects have TTL limits associated with them, then the cache size required to achieve the workload's minimum possible miss ratio is also dramatically lower [95]; e.g., 69% lower than the WSS for the Twitter workloads, on average.[3]

Insight. Substantial memory savings can be achieved by accepting slight increases in the miss ratio and by accounting for TTLs.

5.1.3 Dynamic adjustment of cache sizes

For many cache workloads, cache size requirements change over time. Consider as a typical example the MRCs of the `w77` workload in the Cloudphysics dataset [24] under LRU shown in Figure 5.2. The three curves are the MRCs obtained over hours 27, 28, and 29, respectively. Each MRC identifies a different optimal minimal cache size. For hour 27, the cache reaches its minimum miss ratio at a size of roughly 14GiB. For the next hour, the optimal size reduces to roughly 10GiB. Finally, for the next following hour, the size increases to roughly 13GiB.

This pattern suggests that memory savings can potentially be obtained by periodically changing the workload's cache size. Figure 5.3 shows the cache memory requirements for the `w77` workload over the length of the entire trace under LRU for two scenarios: (1) when the cache size is adjusted to workload's WSS as measured from time 0 to the current time and (2) when the cache size is adjusted each hour to achieve the minimum miss ratio as indicated by the MRC generated over the previous hour. The integral over the adjusted size is 65.7% of that over the static size, representing an average savings of 34.3%.[4] We repeated this for all 900+ access traces considered and found the average savings to be 35.7%. We also found that similar savings can be achieved for caches operating under eviction policies other than LRU. Figure 5.4 depicts the possible savings for caches operating under LFU, FIFO, CLOCK, 2Q [20], ARC [50], S3-FIFO [10], SIEVE [51], LRFU [49], and MRU.

The memory savings through periodic cache size adjustment comes at a cost, however, in that temporarily downsizing a cache may cause an increase in a cache's miss ratio, as some of the cache's objects may be evicted as part of the downsizing, only to be accessed again later. We found that adjusting the cache size each hour resulted in a median increase in the miss ratio of about 1%.

While a 1% median increase in miss ratio may be tolerable, some workloads incur an unacceptably large increase, primarily because their caches downsize considerably during periods of low activity, causing a significant portion of the cache's objects to be evicted, only to be accessed later. One potential way to mitigate this is to apply a *downsize limit* which limits how much a cache can be

[3] The Twitter traces are the only publicly available cache workloads that include TTL attributes.

[4] The savings would be even larger if compared against statically setting the cache size to the WSS as measured over the entire workload.

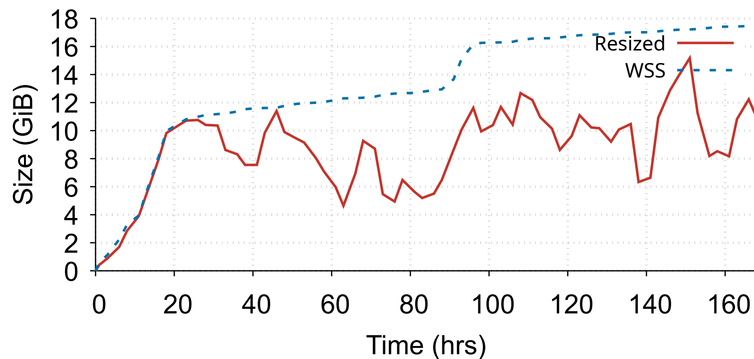


Figure 5.3: Cache size over time when resized to the minimum required each hour and the WSS for the Cloudphysics `w77` trace [24].

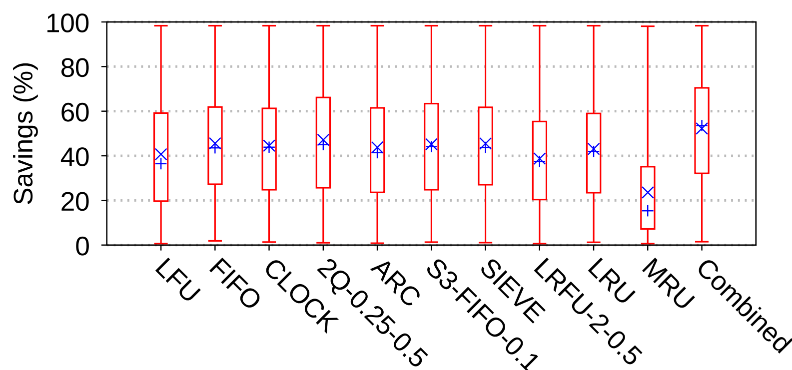


Figure 5.4: Average memory savings across all workloads for different eviction policies. For each policy, the bottom and top whisker identify the minimum and maximum savings achieved by one of the workloads; the bottom and top of the box identify the 25th and 75th percentile result; the $\times$ and $+$ indicate the mean and median savings achieved, respectively.

downsized each hour. For example, using a downsize limit of 50%, such that the cache size cannot be decreased by more than 50%, reduces the median miss ratio increase to 0.27% while still resulting in an average memory savings of 33.7%. Even with this limit, a few workloads continued to have unacceptably high miss ratio increases. Further investigation revealed that these workloads had very large re-access times (i.e., the time between subsequent accesses to the same object). The remedy for these workloads is to identify this behavior and then set their downsize limit to 100% (i.e., no downsizing).

Insight. Substantial additional memory savings can be achieved by dynamically adjusting the cache size every hour;[5] however, a few workloads require special consideration with regard to downsizing because they would otherwise incur unacceptably high miss ratios.

5.1.4 Periodic eviction policy switching

It has been well established that when a cache is operating at a size less than its workload's WSS, then the eviction policy that results in the lowest miss ratio will be workload dependent [3, 9, 25,

[5] We justify using the one hour threshold in §5.3.

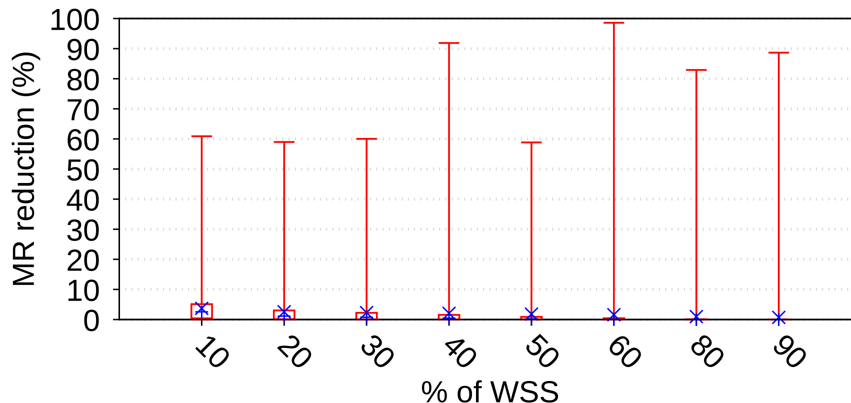


Figure 5.5: Miss ratio reduction when actively switching eviction policies when keeping the size of the workloads cache to a percentage of the workload's WSS. The eviction policies supported were: LRU, LFU, FIFO, S3-FIFO, Clock, 2Q, ARK, Sieve, LFRU, and MRU. See Figure 5.4 for a description of the candlesticks.

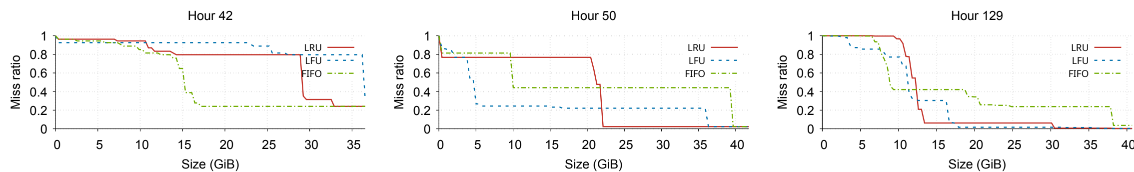


Figure 5.6: Three MRCs for the LFU, LRU, and FIFO eviction policies in 1-hour epochs for the `w15` trace in the Cloudphysics dataset [24].

30, 49, 59].[6] This is relevant for workloads with very large WSSes that do not fit in a host's memory (see Figure 5.1) but also when hosts experience memory pressure so that some caches must operate at a size smaller than their workload's WSS.

Our analysis indicates (somewhat to our surprise) that most workloads only marginally benefit from switching to the eviction policy offering the lowest miss ratio each hour when keeping the cache size constant (however certain workloads achieve significant reductions in miss ratio). Figure 5.5 shows the range of miss ratio reductions achieved across all considered workloads when switching the eviction policy of each cache to the policy offering the lowest miss ratio each hour while maintaining the cache size to a percentage of the WSS of the cache's workload. The median and average miss ratio improvement tops out at just a few percentage points. However, the figure indicates that nevertheless some workloads benefit significantly (as shown by the top whiskers).

A more significant benefit can be realized when combining eviction policy switching with cache sizing. This is because different eviction policies reach their lowest miss ratios at different cache sizes. Consider the example shown in Figure 5.6. Focusing on hour 42, if the cache is operating with a cache size of 35GiB under LRU (which has the lowest miss ratio at that size amongst the three eviction policies considered in the figure), switching from LRU to FIFO allows the cache size to be reduced to 17GiB while achieving (almost) the same miss ratio. Similarly, in hour 129, if the cache is operating at 40GiB under FIFO (which has the lowest miss ratio at that size amongst the

[6] If the cache is operating at size equal to or larger than the WSS, then all eviction policies will have the same miss ratio, namely the MPMR.

three eviction policies considered), switching to LFU allows the cache size to be reduced to roughly 18GiB while achieving the same miss ratio.

The right-most candlestick in Figure 5.4 shows the benefit of combining eviction policy switching with cache resizing each hour. The median and average memory savings achieved across all workloads is 5-10% higher than when not switching eviction policy.

Insight. Combining eviction policy switching with memory sizing can lead to memory savings that are higher than just memory sizing. Eviction policy switching may improve the workload's miss ratio as well.

5.2 Cache Orchestration

The key insights in the previous section foreshadow the potential benefits of running multiple caches on a single host, dynamically adjusting cache sizes based on MRCs, and switching eviction policies. *Cache orchestration* allows us to exploit these insights in a system whose primary aim is to reduce the memory and host footprints of managed caches. Our approach is to formulate cache orchestration as a constrained optimization problem. The solution to the optimization problem is recomputed periodically, once per *epoch*, resulting in a potential reconfiguration of the cache fleet once per epoch.

5.2.1 Optimization problem

Optimization problems involve decision variables, constraints, measurements, and objective functions. Let us consider each in the context of cache orchestration. *Decision variables* are the independent quantities that the optimization algorithm manipulates to achieve its goal of optimizing the objective function. For the cache orchestration problem considered in this chapter, the decision variables form a tuple given by

$$C_i = (S_i, E_i, H_i)\ . \tag{5.1}$$

Here C_i represents the configuration of cache i, for $i \in \{1, \ldots, N\}$, where N is the number of caches being managed. It consists of three elements: $S_i \geq 0$ represents the cache size; E_i identifies an eviction policy; and H_i identifies the host to which cache i is assigned in the current epoch. The configuration of the entire cache fleet is denoted as the tuple $C = (C_1, \ldots, C_N)$.

Constraints are restrictions on the allowable values of the decision variables. One immediately discerns several constraints in the cache orchestration problem. Foremost, the sum of the sizes of the caches allocated on a given host must not exceed the amount of memory on that host. Let $h_j \geq 0$ denote the memory capacity of host j, where $j \in \{1, \ldots, N_{host}\}$, then these constraints are:

$$\left(\sum_{\{i\,:\,H_i\,=\,j\}} S_i \right) \leq h_j\ , \qquad j = 1, ..., N_{host}\ . \tag{5.2}$$

Further constraints may be specified by the cache fleet operator; for instance, specifying that a given cache must be allocated a minimum amount of memory or must be run on an isolated host.

Measurements in the cache orchestration problem include all exogenous data that must be provided to the optimization algorithm to evaluate the objective function(s). These measurements may include: (i) a histogram of stack distances for each eviction policy supported, (ii) the number of accessed objects in each epoch; (iii) the average size of accessed objects, and (iv) the peak resource consumption.

Finally, the *objective function* is a quantity that is specified by the cache fleet operator. Typical optimization functions include minimization of the number of cache misses, the number of hosts, or the total memory usage of the cache fleet. For example, prior cache management systems targeting a single host [73, 79, 114] utilized an objective function that quantifies the aggregate number of misses over all caches:

$$J_1(C) := \sum_{i=1}^{N} \mathrm{MRC}_i(S_i) * A_i. \tag{5.3}$$

Here, $\mathrm{MRC}_i(S_i)$ is the miss ratio of cache i at size S_i (as obtained from the miss ratio curve constructed from the stack distance histogram); and A_i is the number of accesses on cache i. (The miss ratio times the number of accesses is the number of misses.)

5.2.2 Optimization method

We selected to use a genetic algorithm to solve the optimization problem required at the end of each epoch. *Genetic algorithms* (**GA**s) are heuristic algorithms that provide approximate solutions to complex, high dimensional optimization problems. They utilize ideas from natural selection and genetics to search a high-dimensional space of parameters [115–118]. Genetic algorithms have previously been applied in cloud environments to, for example, optimize microservice migrations, optimize resource allocation for virtualized network functions, or optimize load balancing [119–122].

We decided on a genetic algorithm because (i) it can handle complex objective functions, (ii) it has relatively low overhead, and (iii) it allows the operator to dynamically switch/modify the objective function at any time with no overhead. We did not pursue other resource optimization approaches that have been used in prior resource allocation schemes, such as linear programming (e.g., [123, 124]), dynamic programming (e.g., [73, 125]), and machine learning (e.g., [126, 127]). Linear programming formulations are not suitable for our domain because our objective functions rely on MRCs that are fundamentally nonlinear and in some cases not concave, and because the optimization problem we use primarily is multi-level. Dynamic programming has high computational overheads limiting the frequency at which optimization can occur. Machine learning requires a new learning period each time the operator decides to adjust or switch the objective function.[7]

Genetic algorithm background

Genetic algorithms solve both constrained and unconstrained optimization problems over a set of parameters. They generally consist of the following elements:

- a population of *individuals*, each representing a possible solution to the optimization problem;

[7] In practice, we do not expect the objective function to frequently change; however, this may be useful in some cases. For example, if certain servers are experiencing a degradation in performance (e.g., in the event of a hardware failure), the objective function may be updated to migrate caches off of and not place future caches onto these servers.

- *genes* that represent the properties of an individual; specifically, each gene represents the value of one parameter;
- *constraints* (if any) on the set of possible solutions; and
- an *objective function* used to rank the *fitness* of individuals; i.e., distinguish which individuals are closer to optimal.

A genetic algorithm is typically initialized with a random population of individuals, which may represent the user's first guesses of possible solutions. The algorithm then performs a sequence of iterations, with a new generation of individuals produced in each iteration. In each iteration, pairs of individuals, called *parents*, are selected to *mate*.[8] The two parents are selected through a combination of random selection as well as based on those with higher fitness. The *offspring* produced by parents contain genes that are randomly selected from one or the other parent. During the mating process, there is a probability that a *mutation* may occur in which a new gene is quasi-randomly generated for the offspring, rather than copying a gene from a parent.

The genetic algorithm continues iterating until either (i) one or more individual's fitness, as measured by the objective function, is within a pre-specified tolerance of the optimal fitness, or (ii) the configured maximum number of iterations has been reached. When the algorithm completes its iterations, the genes of the fittest individual from the final generation are selected as the solution to the optimization problem.

Genetic algorithm for cache orchestration

For cache orchestration, a gene is a specific configuration, C_i of a cache. An individual consists of a set of genes, one for each cache being managed, and hence corresponds to the tuple C. The operator is expected to provide the objective functions. This can come in the form of a binary comparison function that returns the fitter of two individuals. The operator specifies the constraints on each host, such as its memory configuration or its available network bandwidth. Further, the operator can specify constraints on individual caches, for example to statically set the size of the cache.

5.3 Flux

In this section we present details on Flux, an in-memory cache orchestration system. Flux performs three basic operations: (i) periodically updating the configuration of its currently managed caches, (ii) servicing cache instantiation requests from its clients, and (iii) servicing cache tear-down requests.

Periodic updates of cache configurations occur at the end of each *epoch*, and they involve determining whether and how to adjust the cache size, whether to switch the eviction policy, and whether to reassign a cache to a new host (triggering a cache migration). To make informed decisions balancing the tradeoffs involved when optimizing host resources, Flux requires that each cache provide it with certain information at the end of each epoch: (i) a histogram of stack distances for each supported eviction policy, from which Flux generates corresponding MRCs, (ii) the number of objects accessed as well as their average size, and (iii) peak resource consumption (e.g., network bandwidth consumed).

[8] This is sometimes also referred to as *crossover*. [115]

Flux also services cache instantiation and tear-down requests. By default, each newly instantiated cache is configured with 2GiB of memory and LRU as the eviction policy, and the cache is assigned to a host with sufficient memory. If the client request also identifies the workload for the cache to be instantiated, then Flux exploits this information by incorporating the workload's historical miss ratio curves to more suitably configure the cache size and eviction policy.

5.3.1 Cache requirements

Flux is largely agnostic to the caches it can manage. However, the caches must adhere to several requirements. For instance, as stated earlier, Flux requires that each cache it manages sends it certain information on the behavior of its workload at the end of each epoch. Current caches do not typically compile such information, but we generally found it straightforward to add this functionality with modest effort. For example, we have implemented a custom Redis module capable of generating stack distance histograms for multiple eviction policies using a background thread (thus not affecting the performance of the cache itself) with less than 100MiB of memory overhead and no impact on the cache's miss ratio.[9] (We were able to significantly reduce the overhead of gathering this information by using SHARDS sampling [24]).

As a minor but important detail, it is necessary that the cache clears its stack distance histograms at the beginning of each epoch in order to accurately capture the stack distances incurred during the epoch, but we have found it critical that the eviction stacks themselves not be cleared; otherwise, the stack distances obtained will reflect those of an empty cache and can therefore be misleading.

Moreover, for Flux to be effective, the caches it manages need to be able to modify their size. Modern caches like Memcached and Redis are already able to do this on receiving the appropriate command. For example, with Redis, this is the `CONFIG SET maxmemory` command. When the size is increased, the newly available space will be filled as new objects are added to the cache, and the *residency set size* (**RSS**) of its process will increase accordingly. On a size decrease, the cache will need to evict an appropriate number of objects (using the eviction policy in place) and the cache memory will have to be defragmented. Redis, for example, is surprisingly good at defragementing memory when the `activedefrag` flag is set and `jemalloc` is being used as the underlying memory allocator. For instance, we experimentally ran Redis with the Cloudphysics `w38` access trace until its RSS reached 8GiB and then set `maxmemory` to 1GiB. It took Redis about 4 seconds to defragment the memory and it took jemalloc about 10 seconds to reduce the RSS to 1GiB.[10]

Flux does not strictly require its caches to support eviction policy switching, but memory savings will be more limited if not. Unfortunately, popular open-source in-memory caches are rather limited in their support for eviction policy switching. Memcached [17] only supports a single eviction policy (an optimized variant of LRU). CacheLib [31] supports multiple eviction policies but cannot switch between them at runtime. Redis, on the other hand, supports LRU and LFU and can switch between them [16]. PaperCache is the only in-memory cache we are aware of that is capable of switching between a wider variety of eviction policies [52].

[9] Redis itself does not maintain eviction stacks.
[10] Measurements obtained on an Intel i9-13900KS with 128GiB of DDR5 4200MHz DRAM running Ubuntu 24.04

5.3.2 Constraints

Flux, through its genetic algorithm, is able to ensure that both host and cache constraints are adhered to. To ensure the physical limitations of the hosts are not exceeded, the current Flux implementation takes into account each host's available memory and network bandwidth. (Interestingly, we found that network bandwidth was never a bottleneck when running the workloads listed in Table 5.1: the median peak data access rate was only 83MiB/s.)

Regarding cache constraints, Flux allows the operator to impose restrictions on their cache configurations on instantiation. At this time, Flux supports three types of cache constraints: elastic, target, and fixed. An *elastic cache* implies that the cache's size can be modified to be any value. This treats the cache as a fully-managed elastic cache. A *target cache* is specified with a target cache size and a miss ratio epsilon, which specifies that the cache's size can be modified such that its miss ratio is not higher than epsilon greater than the cache's miss ratio at the target size. This is useful when a client is paying for a specific cache size, as Flux will modify the cache's allocated size such that its miss ratio will at worst be marginally worse (by epsilon) than if it were allocated the full target size. A *fixed* cache is specified with a target size that must be adhered to. This is useful for high priority clients who pay for a specific cache size and must receive exactly that size. Other cache constraints could easily be added.

5.3.3 Genetic algorithm

Genetic algorithms come in many variants and thus require a number of parameters to be selected. For our implementation we selected these parameters according to the recommendations of Rabinovitch et al. [116]. Specifically, we set the size of the population to $1,000$ and the probability of a mutation to $1/N$, where N is the number of caches being managed. Parents are selected for mating as follows: the fittest of three randomly chosen individuals is selected as one of the parents, and the fittest of another three randomly chosen individuals is selected as the second parent.

At the start of the the optimization process, 1,000 individuals are generated, with one individual corresponding to the current cache configurations and the other 999 randomly generated with one gene for each cache being managed. Each newly generated gene is configured as follows: (i) the host is selected randomly, (ii) the memory is randomly set to a value between zero and the minimum of (a) the host's available memory and (b) the smallest cache size which achieves the workload's MPMR (as determined by the previous epoch's MRC), and (iii) the eviction policy is selected as the policy that has the lowest miss ratio at the selected size. (If a generated individual violates the given constraints then the generation of the individual is restarted from scratch.)

The genetic algorithm then starts iterating to produce successive generations. In our implementation, each generation produces 1,000 offspring to form a new generation. If a mutation occurs during a mating, then the mutated gene is generated in the same way the genes were generated at the beginning of the process (as described above). Iterations continue for 60 seconds, which typically results in approximately 8,700 generations on our hardware. At the end of the iteration process, the fittest individual is selected and the caches are instructed to update accordingly.[11]

Cache constraints affect how gene mutations occur in Flux's genetic algorithm. The sizes of

[11] The fitness of each individual is calculated using the cost function in place. The cache's MRC is used to determine the miss ratio at each gene's configured cache size.

caches with the *target*, or *fixed* configurations are not selected randomly during mutation. Instead, for the *target* configuration, the smallest cache size which achieves the constraint is selected based on the cache's MRC.

5.3.4 Cache migration

Occasionally, Flux will find it necessary to migrate a cache from one host to another. This may occur if Flux decides a cache needs to increase its size but there is not enough free physical memory available on the host it is running on. Another reason Flux will migrate a cache is to enable cache serving host consolidation. To prevent the costly migration of caches between servers on different racks, we recommend a separate instance of Flux be used to orchestrate the caches for each server rack.

To ensure clients are not impacted by the migration of their cache, cache migration should not: (i) create downtime for the cache, or (ii) result in a higher miss ratio of the cache during or after migration. A number of cache migration mechanisms with zero downtime have previously been proposed [88–90, 128]. For example, Rocksteady is able to migrate in-memory caches at 758 MB/s between hosts under high load [88, 129].

5.3.5 Flux implementation

Flux is expected to run as a separate process, typically on a dedicated server. Flux is implemented in Rust v1.89.0-nightly in roughly 3K LoC. Random individual creation when initializing Flux's genetic algorithm as well as individual mating is done in parallel. We intend to make the implementation available as an artifact.

Flux has several configurable parameters. These include the epoch size (1hr default), the cache downsize limit (50% default), and the default cache instantiation configuration (2GiB, LRU default). In addition, the GA also has a number of configuration parameters, as discussed in §5.3.3, including the number of individuals used, the limit on how long iterations should continue,[12] and the probability of a mutation occurring.

Flux relies on the caches it orchestrates to manage their own MRCs and access statistics, which are sent to Flux at the end of each epoch. This allows Flux's implementation to be entirely stateless, simplifying its fault tolerance – if the Flux service notices downtime during an epoch and is restarted before the epoch ends, there will be no change in its management of the caches. If the Flux service notices a longer downtime (e.g., spanning multiple epochs), the caches will not be reconfigured and will operate using their last assigned configuration settings.

5.4 Case Studies

We conducted a number of case studies to evaluate the potential of Flux. Our experiments ranged from small-scale to large-scale with hundreds of concurrently running caches. We relied entirely on real-world cache workloads for which access traces are publicly available (for reproducibility). The results of our experiments indicate that multiple concurrently running caches orchestrated by Flux

[12]Typically, to prevent unbounded optimization, GAs are limited in their number of generations. However, as we require the GA to produce a solution within a limited time window, we limit the GAs maximum runtime instead.

Table 5.2: Host hour and GiB hour reductions attained by Flux.

Experiment	Host capacity	Host hour reduction	GiB hour reduction
Small-scale	256GiB	55%	74%
Small-scale	1,024GiB	14%	55%
Cloudphysics	256GiB	42%	70%
Cloudphysics	1,024GiB	49%	67%
Alibaba	256GiB	48%	90%
Alibaba	1,024GiB	79%	95%

require significantly fewer hosts and significantly less memory, while having only marginally higher miss ratios, when compared to reasonable baselines (described below). Table 5.2 summarizes the key results of our experiments: with Flux, host usage was reduced by 14-79% and memory usage was reduced by 55-95%.

Baselines. Our baseline caches were configured and managed as follows. First, baseline caches were configured to use the LRU eviction policy throughout their operation, as LRU is often the default eviction policy for in-memory caches. Second, a first-fit bin-packing algorithm was used to assign newly instantiated caches to hosts with the objective of minimizing the number of hosts used. However, once a cache for a given workload was assigned to a host, it remained on the host until the cache was torn down. Finally, baseline cache sizes were statically configured to be equal to the minimum of (i) the working set size (WSS) of the workload the cache is running, assuming the WSS of the *entire* workload is oracularly known when the cache is instantiated, and (ii) the host's memory capacity. That is, if the workload's WSS is less than or equal to the host's memory capacity, then the cache size will be configured to be equal to the WSS, thus incurring only compulsory misses and the lowest possible minimal miss ratio. We note that this baseline configuration will tend to use less memory than configurations used in practice because the workload's WSS is typically not known a priori, which is why operators tend to over-provision caches so as not to compromise performance. As a result, the memory savings we report for Flux are reasonably conservative.

5.4.1 Experimental Setup

Hosts. We ensured that a sufficient number of hosts were available to accommodate all concurrently running caches. We further assumed all hosts in the fleet had equal memory capacity,[13] but repeated each experiment with host memory capacities of 256GiB and 1,024GiB of memory, mirroring the capacities of the `x2iedn.2xlarge` and `x2iedn.8xlarge` EC2 instances offered by AWS [36], respectively. Flux ran on a dedicated host with an Intel i9-13900KS and 128GiB of DDR5 4200MHz DRAM running Ubuntu 24.04.

Caches. We selected PaperCache [52] as the cache server, because it supports multiple eviction policies and can switch between eviction policies efficiently at any time. Besides modifying PaperCache to provide Flux with the information required by Flux, we also modified PaperCache in two additional ways to reduce the time needed to run our experiments. First, we modified PaperCache to process each workload's incoming access requests in sequence as quickly as possible without waiting

[13] This is not a limitation of Flux which can easily orchestrate caches on hosts with heterogeneous capacities.

for the wall clock to reach the access time associated with each access; instead we had PaperCache track time using the access timestamps (for gathering metrics and for synchronizing the caches at the end of each Flux epoch). Second, we modified PaperCache such that it does not store objects in memory, but rather only keeps track of the memory consumed by all objects in the cache; however, it still fully processes incoming cache access requests, including maintaining the index, looking up the key of the accessed object in the index and maintaining eviction structures (e.g., LRU queue). This greatly reduced the residency set size (RSS) of the cache processes as we "simulated" a large number of caches running in parallel. We ran both the original PaperCache and its modified version on numerous cache workloads to verify that the key statistical measurements were identical.

Flux configuration. Each newly instantiated cache under Flux was initially configured to use 2GiB of memory with LRU as the eviction policy. All caches sizes were configured to be "elastic" (as described in §5.3.2); i.e., with no contraints on its size. The downsizing limit was set to 50%. Epochs of 1hr were used; i.e., every hour, each cache sends to the Flux server its current configuration, the average object size, statistics on the network bandwidth used, and the miss ratio curves of each eviction policy the cache supports (in our case LRU, LFU, MRU, FIFO, S3-FIFO, 2Q, CLOCK, and SIEVE). With this information, Flux determines the configuration of each cache for the next epoch with its genetic algorithm (GA). The GA used a population of 1,000 individuals and produced new generations for 60 seconds before reconfiguring the caches.

Metrics. The primary metrics we focus on are (i) the number of hosts required (in host hours), (ii) the amount of memory required (in GiB hours), and (iii) the difference in the miss ratio relative to the baseline.

5.4.2 Choice of Objective Function

The starting point for the construction of our objective function is the objective function used in prior work, J_1 defined in Equation 5.3. We refined J_1 in several ways. First, we optimize on capacity misses instead of all misses so as not to discriminate against workloads that naturally have a high miss ratio. Capacity misses can be estimated using the cache's MRC: $(\mathrm{MRC}_i(S_i) - \mathrm{MRC}_i(\infty)) \cdot A_i$, where $\mathrm{MRC}_i(S_i)$ is the miss ratio of cache i at size S_i, $\mathrm{MRC}_i(\infty)$ is the compulsory miss ratio, and A_i is the number of accesses during the previous epoch.

Second, to better account for the load on backend storage caused by cache misses, we then multiply the number of capacity misses by the average size of the objects accessed in the previous hour, given that misses to large objects incur higher overheads than misses to small objects [9]: $(\mathrm{MRC}_i(S_i) - \mathrm{MRC}_i(\infty)) \cdot A_i \cdot O_i$, where O_i is the average size of objects accessed in cache i in the previous hour.

Third, we take into account the overhead of caches migrating from one host to another, as we found that not doing so causes caches to bounce from host to host, resulting in a high level of inefficiency. For this, we use the size of the cache being migrated as a proxy for the migration overhead. Thus, in our objective function, the migration cost for cache i is approximated by $M_i \cdot S_i$, where $M_i \in \{0, 1\}$ denotes whether cache i migrates or not.

Together, this leads to the following objective function:

$$J_2 := \sum \left[\, \big(\mathrm{MRC}_i(S_i) - \mathrm{MRC}_i(\infty)\big) \cdot A_i \cdot O_i + M_i \cdot S_i \, \right] \tag{5.4}$$

An additional objective function aims to minimize the number of hosts running caches:

$$J_3 := \#\text{ hosts running caches} \tag{5.5}$$

A final objective is to minimize the total allocated size of all caches:

$$J_4 := \sum S_i \tag{5.6}$$

to allow the hosts to maintain available space, if possible, to accommodate tenants' changing workloads or the introduction of new tenants.

In summary, we have the Flux genetic algorithm solve the following mutli-level optimization:

$$\begin{aligned} Y_2 &= \underset{C}{\operatorname{argmin}}\ \ J_2(C) \quad \textit{s.t. Equation } 5.3 \textit{ holds} \\ Y_3 &= \underset{C \in Y2}{\operatorname{argmin}}\ \ J_3(C) \\ &\min_{C \in Y_3}\ \ J_4(C) \end{aligned} \tag{5.7}$$

We would like to emphasize that there are many other objective functions one might wish to consider, but J_2, J_3, and J_4 seems like a reasonable starting point. Defining objective functions that achieve desired objectives while being fair is complex, especially if constraints are involved [130]; however evaluating tradeoffs between different objective functions is beyond the scope of this work.

Further, the optimization Flux implements is more involved than what is described above. It selects the eviction policy based on miss ratio achieved and possible memory savings. It supports constraints on caches that might be set by an operator, including specifying a fixed static size or specifying that a cache should be sized so that it's expected miss ratio is within ϵ of the miss ratio of a cache sized at S_i. It also takes into account network bandwidth constraints.

5.4.3 Small-scale experiments

In this set of experiments, we evaluated Flux using four randomly selected access traces from each of the Twitter [3], Cloudphysics [24], Tencent CBS [26], and Alibaba [27] datasets. We staggered their entry and exit times to simulate a dynamic, real-world hosting environment where clients instantiate and tear-down cache instances dynamically. Table 5.3 shows a summary of the workloads we used as well as their start and end times.

We present the results of our evaluation for two different host configurations: one with 256GiB of available memory and 25Gb/sec of available network bandwidth, and the other with 1,024GiB of available memory and 25Gb/sec of available network bandwidth.

256GiB hosts. Figure 5.7 depicts key metrics over time for the first host configuration. These are (i) the number of running caches and hosts being used in the baseline and by Flux; (ii) the aggregate cache sizes of the currently running workloads, (iii) the average difference between miss ratios obtained by the Flux-configured caches and the baseline caches (referred to as the "miss ratio difference") and (iv) the extra network costs incurred by the Flux-configured caches due to capacity misses and migrations relative to the baseline caches.

Over the duration of the experiment, Flux reduced the number of host hours relative to the

Table 5.3: Workloads used in our experiments. Entry and exit times are shown for each, as well as their WSS.

Dataset	Trace	Entry (hrs)	Exit (hrs)	WSS (GiB)
Alibaba	ab131	0	793	15
Alibaba	ab326	0	793	7
Alibaba	ab547	0	793	6
Alibaba	ab733	0	793	80
Twitter	cluster15	0	168	0.5
Twitter	cluster22	0	168	0.2
Twitter	cluster26	0	168	9
Twitter	cluster41	0	168	2.5
Cloudphysics	w03	168	336	1,779
Cloudphysics	w12	192	360	8
Cloudphysics	w83	216	384	20
Cloudphysics	w102	240	408	8
Tencent CBS	node5	504	672	155
Tencent CBS	node15	528	696	659
Tencent CBS	node28	552	720	156
Tencent CBS	node29	576	744	293

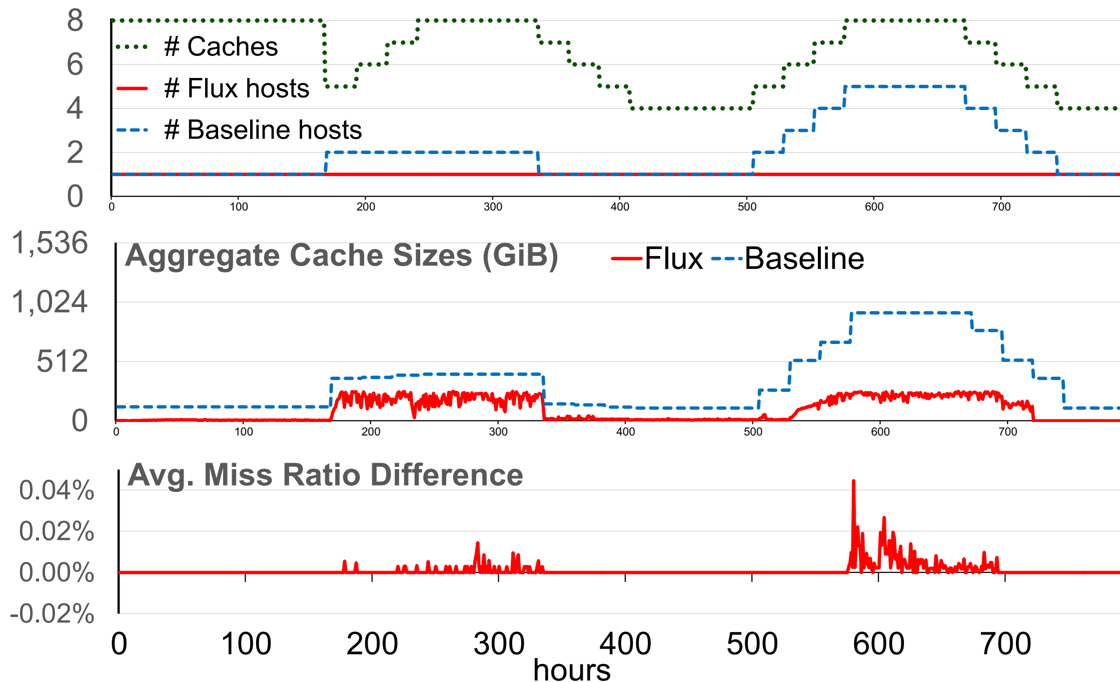


Figure 5.7: Stats from small-scale experiment with 256GiB hosts. *Top graph*: Number of active caches and hosts; *middle graph*: aggregate cache sizes; *bottom graph*: average miss ratio difference between Flux-managed and baseline caches.

baseline from 1,628 to 793, corresponding to a 51% reduction, and it reduced the amount of memory used in terms of GiB hours by 74%. Flux performed a total of 1,365 and 1,630 cache size increases and decreases, respectively, and 498 eviction policy switches. Notably, the average miss ratio difference

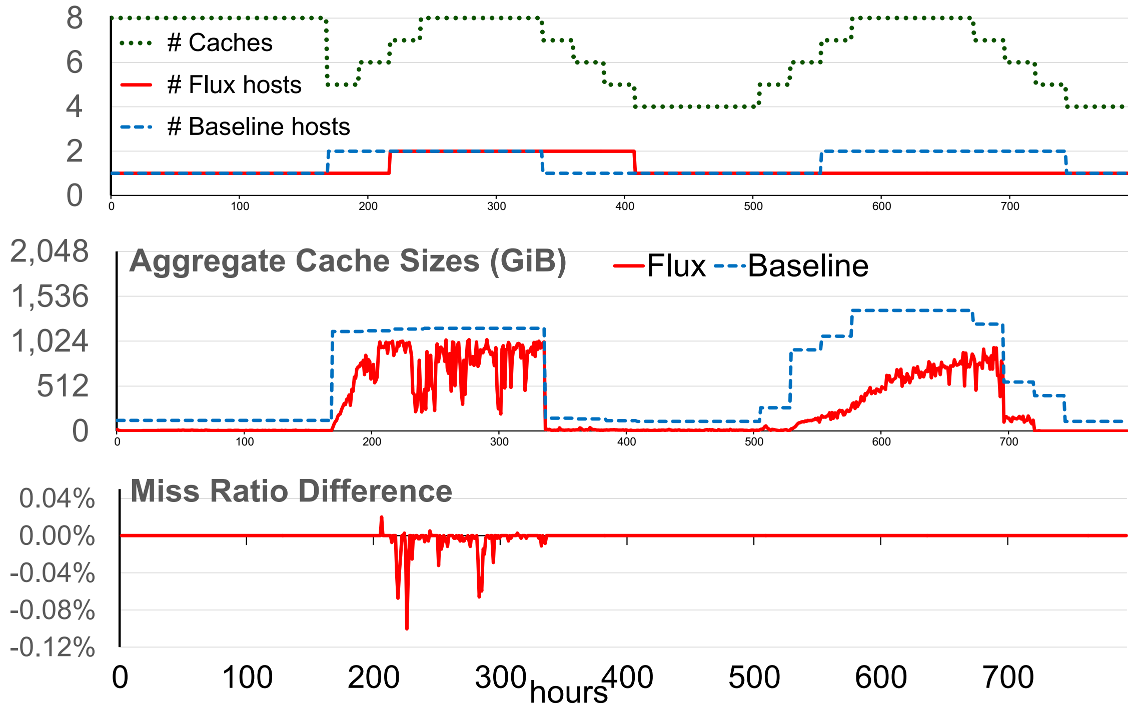


Figure 5.8: Stats from small-scale experiment with 1,024GiB hosts. See Figure 5.7 caption for description of graphs.

remained well below 0.1% throughout.

The baseline used as many as five hosts simultaneously. Flux never used more than one host, because Flux determined from the MRCs that the miss ratio for none of the concurrently running workloads would improve substantially if their cache were upsized to 256GiB. This is confirmed by the fact that the caches continuously perform well in that the average increase in each cache's observed miss ratio relative to that of the baseline remained low.

1,024GiB hosts. Figure 5.8 depicts the same metrics for the second host configuration. Over the duration of the experiment, Flux reduced the number of host hours relative to the baseline from 1,151 to 984, corresponding to a 14% reduction, and it reduced the amount of memory used in terms of GiB hours by 55%. Flux performed a total of 1,230 and 1,281 cache size increases and decreases, respectively, and 379 eviction policy switches. We note that the miss ratio difference curve goes negative at points. This is because the baseline statically used LRU as the eviction policy throughout, while Flux could switch to a more advantageous eviction policy at times.

The baseline made use of a second host twice: when the cache for workload `w03`, with a WSS of 1,787GiB, was instantiated and when the cache for workload `node28` was instantiated. Somewhat surprisingly, Flux also made use of a second host at time 216 when the cache for workload `w83` was instantiated. This is because at time 216, the first host did not have 2GiB of unallocated memory, so an additional host was needed for the new cache.

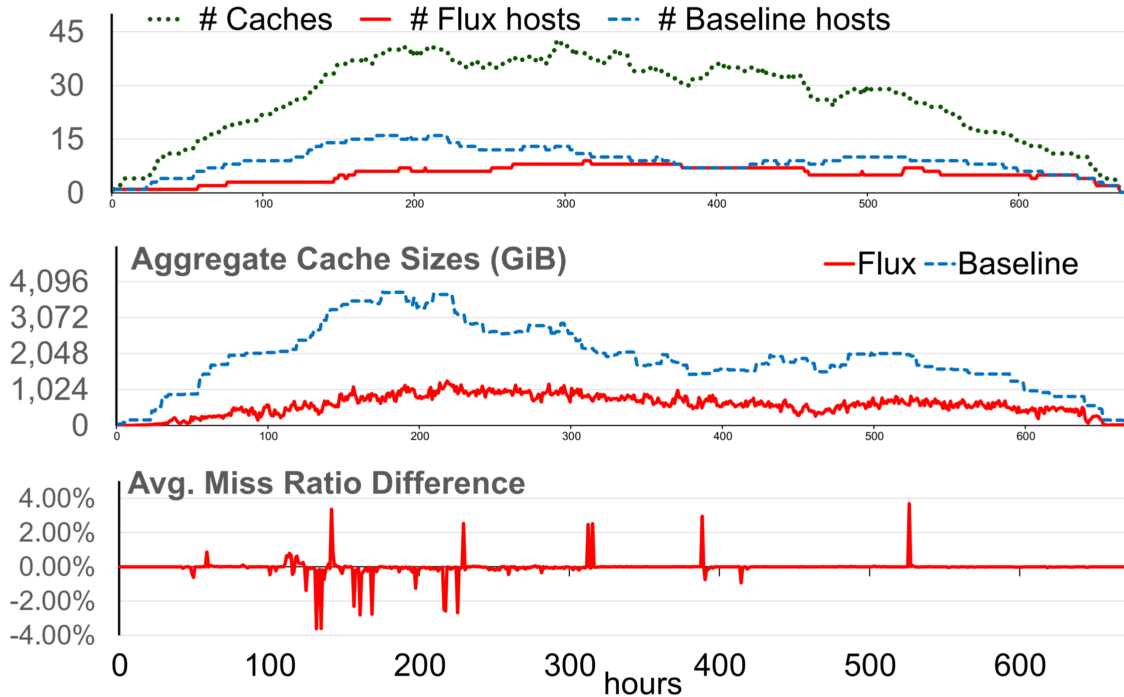


Figure 5.9: Stats from Cloudphysics experiment with 256GiB hosts. See Figure 5.7 caption for description of graphs.

5.4.4 Cloudphysics workloads

We evaluated Flux's ability to manage a medium-scale hosting environment using all (106) access traces in the Cloudphysics dataset [24]. We selected this dataset for this experiment because it was the second largest dataset available to us. The Cloudphysics workloads include over 2 billion accesses and a combined WSS of almost 22TiB. We randomly staggered the entry times of the cache instances to simulate a dynamic real-world hosting environment. The workload for a newly instantiated cache was selected randomly from the set of 106 access traces that had not been previously selected. Each of the 106 workloads ran to completion over the course of the experiment. During that time, the peak aggregate WSSes of concurrently-running caches was 9.5TiB. We again present our results for hosting environments with 256GiB and 1,024GiB hosts.

256GiB hosts. Figure 5.9 depicts key metrics over time. The baseline required as many as 16 hosts, while Flux required at most 8. Flux reduced the number of host hours relative to the baseline from 6,079 to 3,508, corresponding to a 42% reduction, and it reduced the amount of memory used in terms of GiB hours by 70%. Flux performed a total of 6,518 and 8,467 cache size increases and decreases, respectively, and 6,397 eviction policy switches.

Flux initiated a total of 660 cache migrations, which corresponds to almost one every hour. However, the size of the caches being migrated was relatively small, with the median (average) size being just 65MiB (198MiB); the largest cache that was migrated was under 4GiB in size.

One will note that the Miss Ratio Difference curve has several spikes that reach 4%. Further investigation revealed that each spike was caused by one workload that experienced an extremely high miss ratio during the epoch, pulling up the average. (Each spike was caused by a different

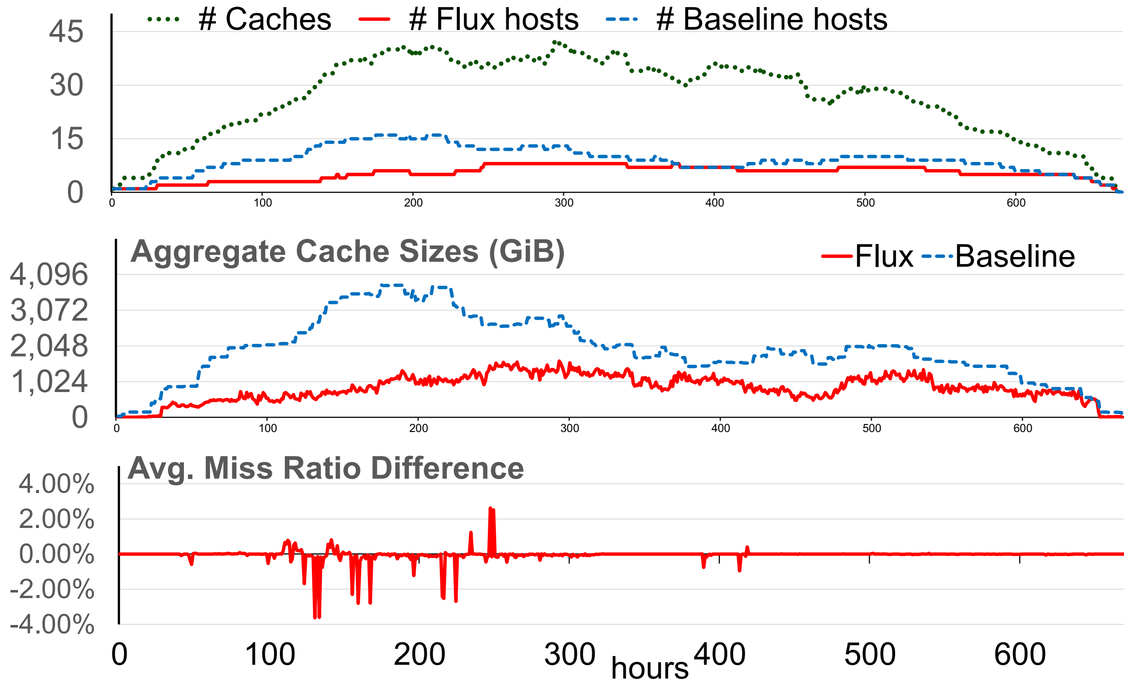


Figure 5.10: Stats from Cloudphysics experiment with 256GiB hosts with 5 fixed-size caches. See Figure 5.7 caption for description of graphs.

workload.) The workloads that caused these spikes all have large WSSes and large reuse distances, yet Flux downsized their caches. To illustrate why, we focus on one of these cases. In the epoch prior to the spike, the cache size was set to 20GiB. It's MRC identified that the miss ratio could be reduced significantly by increasing the cache size to 170GiB. But its host only had 34GiB free memory available, and Flux determined that increasing the cache size by that amount would offer no benefits. To the contrary, Flux actually reduced the size of the cache to 10GiB because it's MRC was flat up until 170GiB. Moreover, Flux did not decide to migrate that cache to a different host, primarily because the J_2 objective function disincentivizes migrations, particularly for large caches.

One remedy is to configure the caches for these workloads to fixed, static sizes. We reran this experiment with 5 of the 106 workload caches set to have a size equal to their WSS, and the spikes were either eliminated or significantly diminished; see Figure 5.10. However, the host hours savings was reduced from 42% to 41% and the memory savings was reduced from 70% to 57%.

1,024GiB hosts. Figure 5.11 depicts key metrics over time for the 1,024GiB host configuration. Over the duration of the experiment, the baseline required as many as 10 hosts, while Flux requires at most 3. Flux reduced the number of host hours relative to the baseline from 3,336 to 1,714, corresponding to a 49% reduction, and it reduces the amount of memory used in terms of GiB hours by 67%. In total, Flux performed 7,301 and 8,721 cache size increases and decreases, respectively, and 6,298 eviction policy switches.

Flux initiated 336 cache migrations, which corresponds to one migration every two hours. The median (average) size of the migrated caches was 64MiB (131MiB); all migrated caches were under 1GiB in size, with one exception where the migrated cache was 4GiB in size.

We again observe several Miss Ratio Difference spikes. The same potential remedy as used in

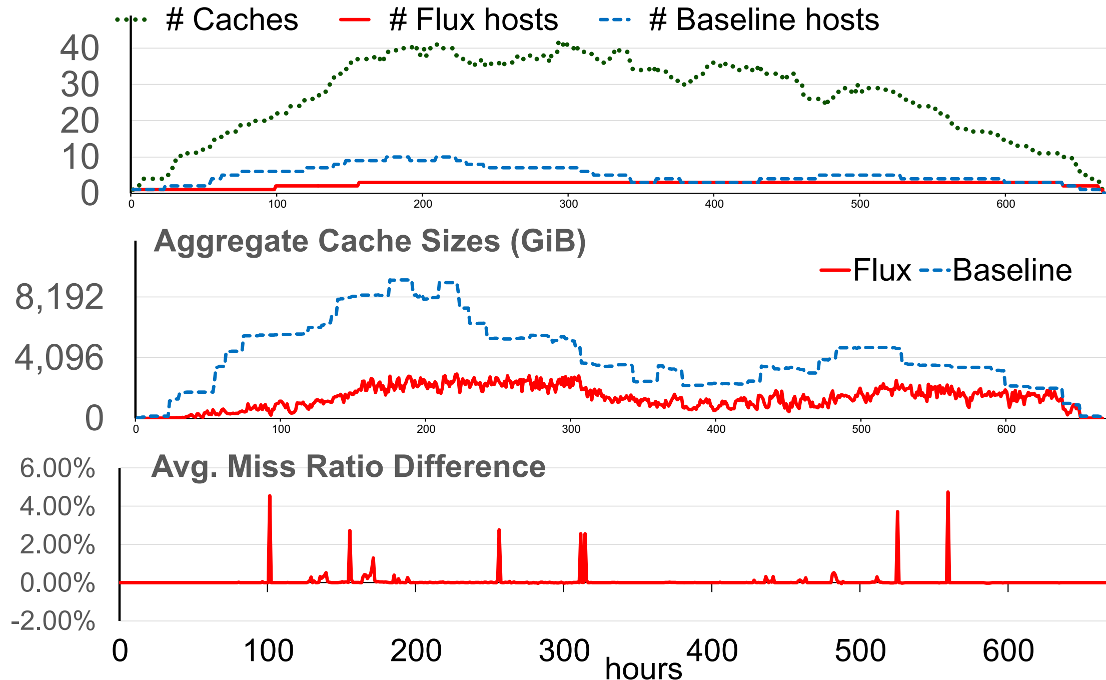


Figure 5.11: Stats from Cloudphysics experiment with 1,024GiB hosts. See Figure 5.7 caption for description of graphs.

the 256GiB host configuration applies.

5.4.5 Alibaba workloads

We evaluated Flux's ability to manage a larger-scale hosting environment using all (609) access traces in the Alibaba dataset [27]. We chose this particular dataset for these experiments because it was the largest publicly available dataset available to us. These workloads entail over 20 billion accesses and an aggregate WSS of over 112TiB. For this experiment, we launched all 609 workloads simultaneously.

Figure 5.12 and 5.13 show the key metrics gathered over the first four days of operation. With 256GiB hosts, the baseline caches occupied 596 hosts, while the Flux caches occupied between 300 and 340 hosts after a warmup period. Flux reduced the number of host hours by 48%, from 51,852 to 27,166, and the number of GiB hours by 90%. It initiated 10,713 and 11,966 cache size increases and decreases, respectively, as well as 539 eviction policy switches.

With 1,024GiB hosts, the baseline caches occupied 536 hosts, while the Flux caches occupied between 110 and 113 hosts after a warmup period. Flux reduced the number of host hours by 79%, from 46,632 to 9,702, and the number of GiB hours by 95%. It initiated 12,034 and 12,502 cache size increases and decreases, respectively, as well as 685 eviction policy switches.

In this set of experiments, Flux initiated a total of 14,291 and 11,074 cache migrations under the 256GiB and 1,024GiB hosting environments, respectively. The median and average sizes of migrated caches was 1,017MiB and 2.3GiB iunder 256GiB hosts, and 1.3GiB and 6.2GiB under 1,024GiB hosts, respectively. Unlike in the previous experiments with fewer concurrently-running caches, here, Flux had to perform more frequent migrations of larger caches due to increasing memory pressure. These

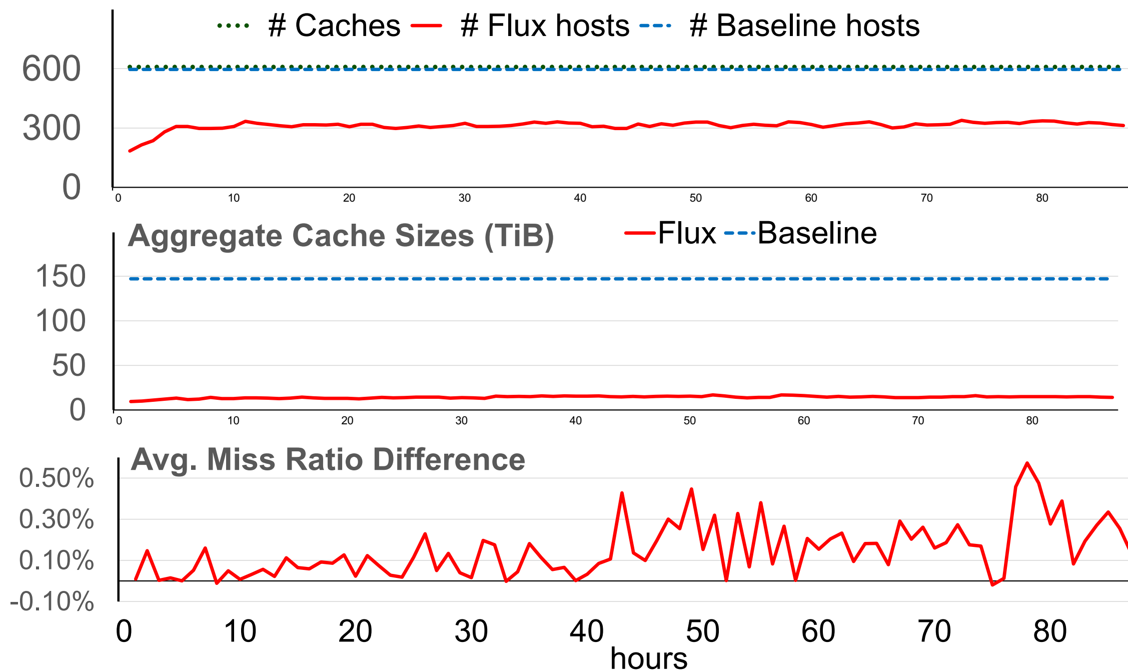


Figure 5.12: Stats from Alibaba experiment with 256GiB hosts. See Figure 5.7 caption for description of graphs.

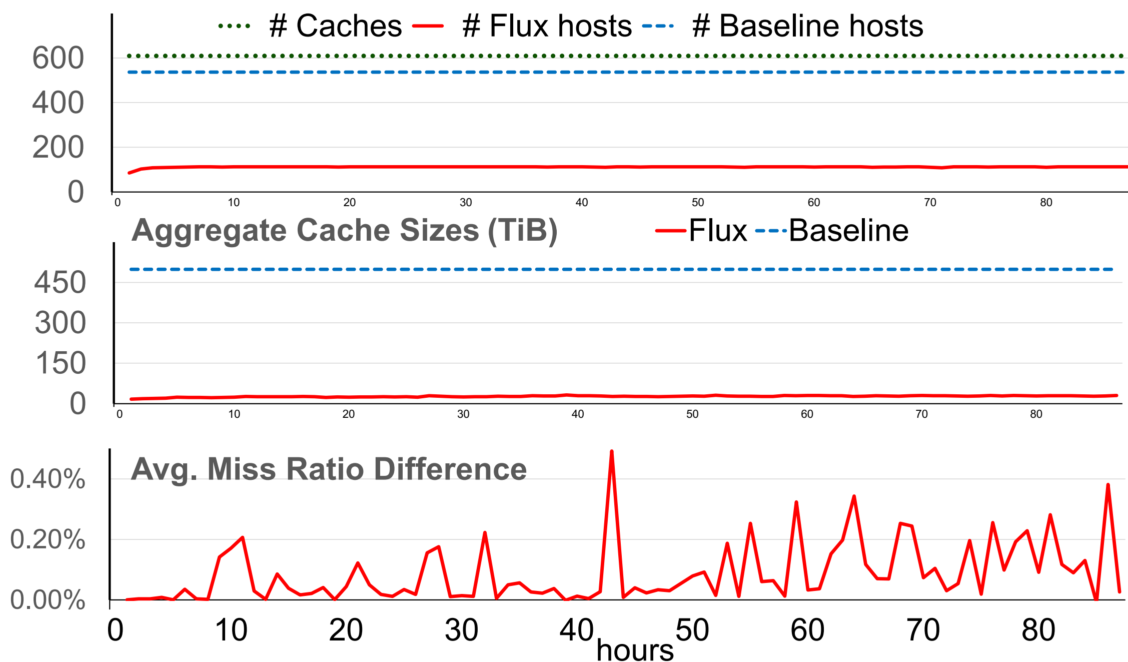


Figure 5.13: Stats from Alibaba experiment with 1,024GiB hosts. See Figure 5.7 caption for description of graphs.

exceedingly high number of migrations highlight the importance of more strategic cache placement of newly instantiated caches. When a new cache is initialized, Flux places it onto an existing host

with the most available unused memory, or a new host if no existing host has sufficient memory available. However, in this case, where a large number of caches are instantiated simultaneously, congestion occurs, wherein the caches are packed onto a few hosts and must later be migrated when their WSSes expand (the Alibaba workloads [27] used in these experiments have particularly large WSSes). In such scenarios, a more suitable host selection algorithm may reduce the number of cache migrations. We leave the exploration of such algorithms for future work.

5.4.6 Choice of Flux parameters

We evaluated various Flux parameter configurations to find suitable values to use in our evaluation. To identify a suitable epoch length, we compared, for each workload in Table 5.1, (1) the miss ratio when adjusting its cache size at the end of each epoch to the optimal size according to the epoch's MRC, to (2) the miss ratio of its cache when sized to be equal to the workload's WSS (thus only incurring capacity misses). We considered epoch lengths ranging from 1 minute to 12 hours. For 1min epochs, the median increase in miss ratio was 5%; for 12hr epochs it was 0.22%. The median increase in miss ratio for 1hr epochs was roughly 1% which reduced to 0.27% when configured with a downsize limit of 50%. We selected 1hr as a suitable epoch length as it does not cause a significant increase in the cache's miss ratio while allowing Flux to adapt to changing workloads at a reasonable frequency.

To determine a suitable maximum runtime for Flux's GA optimization, we compared the J_2 cost function values of Flux to those of an ideal exhaustive search approach using dynamic programming. We experimented with scenarios with between 2 and 106 caches, each running a different Couldphysics workload, and 4 256GiB hosts. We recorded with maximum runtime values of 10s, 30s, and 60s and found that Flux achieves a maximum cost value increase over the ideal configuration of 0.59%, 0.42%, and 0.33%, respectively. For 106 caches, the dynamic programming approach took roughly 1.1hrs to arrive at a solution. Although all our evaluated maximum runtimes yielded acceptable cost increases, we use 60s in our evaluation as it is suitable for larger numbers of caches and is well within our epoch length of 1hr.

5.5 Related work

Several past works have examined resource allocation in multi-tenant, multi-server environments [131–133]. Ballani et al. use virtual networks to model inter-VM network bandwidth and match tenants with host resources [131]. Guo et al. describe a data center abstraction, called a *virtual data center*, to allow for improved elasticity under changing tenant workloads [132]. Rai, Bhagwan, and Guha use GPUs to perform data center optimizations, such as VM placement and network bandwidth allocation [133].

Many studies have focused on cache memory allocation [73, 78, 79, 81]. Cidon et al. describe a optimization technique called *Dynacache* which minimizes aggregate cache misses on a shared host [78]. Byrne, Onder, and Wang apply configurable allocation bounds to individual tenants subject to the cost function introduced in Dynacache [78, 79]. Zhang et al. introduce an efficient modeling technique which improves Dynacache's optimization technique [73]. Cidon et al. describe a memory re-distribution technique that dynamically adjusts the memory allocations of caches on a shared host to improve hit ratios [81].

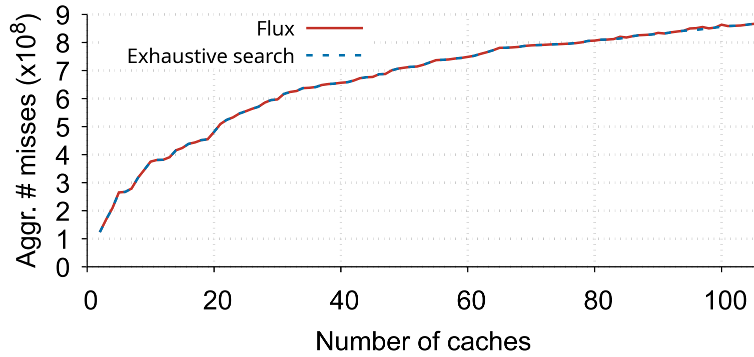


Figure 5.14: Aggregate number of misses of caches managed by Flux and an exhaustive search approach for between 2 and 106 managed caches.

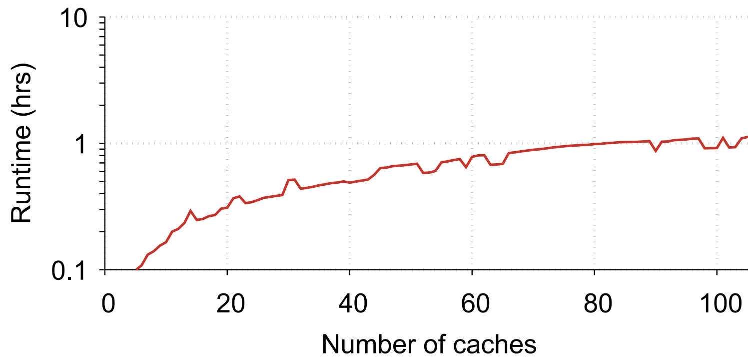


Figure 5.15: Runtime of an exhaustive search orchestrator for between 2 and 106 managed caches.

5.5.1 Comparison to prior work

Although Flux's problem domain differs from that of prior work (e.g., Flux optimizes caches running on fleets of hosting servers whereas prior techniques are limited to a single host), for completeness, we present a comparison of Flux's optimization method to the exhaustive search technique used by previously proposed cache orchestrators [73, 78, 79]. Here, we use the J_1 cost function (Equation 5.3) as a direct comparison with prior techniques to demonstrate the accuracy of Flux's heuristic optimization approach. Figure 5.14 shows the computed cost (i.e., aggregate number of misses) when using Flux compared to the ideal, exhaustive search approach when managing between 2 and 106 caches on a host with 1TiB of available memory. We found that the maximum cost increase noticed by Flux is only 0.9%. Figure 5.15 shows the measured runtime of the exhaustive search approach to calculate each optimal configuration.[14] We found that when managing a large number of caches, while Flux has a fixed maximum runtime of 60s, the high computational cost of prior exhaustive search techniques results in runtimes exceeding 1hr, making them unsuitable for use in real-world cache orchestrators.

[14]We employed the same dynamic programming techniques as described in prior work [73, 79].

5.6 Concluding remarks

In this chapter, we introduced Flux, a novel online in-memory cache orchestrator which periodically reconfigures the size, eviction policy, and host placement of caches running on a fleet of hosts. Flux migrates caches between hosts to consolidate, or to improve performance with its primary objective being to reduce the number of host servers in use while adhering to host limitations, such as network bandwidth. Flux is agnostic to its objective function and allows for user-definable objective functions which can be modified at runtime. We demonstrated Flux's ability to manage fleets of between 16 and 609 caches and showed that it can reduce the memory resource requirements by 55%-95% and host resource requirements by 14%-79%.

In the future, we plan on investigating more robust objective functions (e.g., with explicit fairness requirements) as well as additional host limitations (e.g., CPU limitations, memory bandwidth, etc.) for Flux to consider during its optimizations. We further plan to examine the limits of Flux's scalability (e.g., maximum number of managed caches).

Chapter 6

Concluding Remarks

In this dissertation, we examined methods to model, improve the performance of, and automatically manage application-level in-memory caches. We showed that a cache's eviction policy can have significant effects on its performance, though modern modeling techniques only model caches under LRU. We demonstrated that the optimal eviction policy for a cache under a workload can change over time, though in-memory caches are currently limited in their abilities to switch between eviction policies at runtime. Finally, we examined how the configurations of in-memory caches running on a fleet of hosting servers can be modified over time to reduce resource usage and improve cache performance.

6.1 Contributions

We made the following major contributions:

- We introduced **Kosmo** (Chapter 3), an MRC generation algorithm which allows for the efficient generation of MRCs for various eviction policies simultaneously, online. We showed that Kosmo can generate MRCs using 3.6 times less memory, on average, and up to 36 times in the extreme case, than MiniSim, the current state-of-the-art algorithm for generating MRCs for non-LRU eviction policies. We also demonstrated that an eviction policy's inclusion property is an important consideration when generating MRCs and presented the first real-world examples of violations of the inclusion property for the LFU, FIFO, 2Q, LRFU, and MRU eviction policies.
- We introduced **PaperCache** (Chapter 4), a novel in-memory cache design which can switch between any eviction policy instantaneously, at runtime. We demonstrated that Redis, a popular in-memory cache (and, to our knowledge, the only in-memory cache that supports switching between eviction policies at runtime), is extremely limited in the eviction policies it supports and observes miss ratios that deviate from ideal implementations of its eviction policies. We showed that PaperCache supports any eviction policy and achieves miss ratios within 1% of ideal implementations of its eviction policies.
- We introduced **Flux** (Chapter 5), an in-memory cache orchestration framework which manages in-memory caches running on a fleet of hosting servers. We showed that previous cache orchestration techniques only model caches running on a single hosting server and only support the

adjustment of each cache's allocated size. We demonstrated that Flux is an open framework, allowing administrators to supply their own optimization criteria, and automatically adjusts the caches' allocated sizes and eviction policies, and migrates caches between hosting servers for consolidation or performance improvements. We also described interesting insights into the frequency with which a cache's allocated size and eviction policy should be evaluated and adjusted over time.

6.2 Future research directions

The work presented in this dissertation leads to many interesting future research directions.

Eviction policy inclusion property violations

In §3.1 we demonstrated examples of inclusion property violations for many popular eviction policies, including policies that were previously thought to adhere to the inclusion property (e.g., MRU [48, 57, 93, 94]). MRCs for these eviction policies may, in some cases, exhibit non-monotonic behavior where, counterintuitively, increasing the cache's size also increases its miss ratio. The implications of the inclusion property on periodic cache reconfigurations for performance benefits is not well-trodden as we were the first to demonstrate violations for the LFU, FIFO, 2Q, LRFU, and MRU eviction policies.

Workload-specific eviction policies

PaperCache's (Chapter 4) ability to periodically automatically evaluate the miss ratios of multiple eviction policies and switch to the most performant policy at runtime creates an interesting opportunity for eviction policy development. Designers of eviction policies can now target more specific workloads (e.g., eviction policies which perform well for scanning or cyclic access patterns, though do not perform well for other patterns, such as MRU) and rely on PaperCache to monitor and switch to said policies only when beneficial at runtime. This allows designers to not make compromises on the performance of their eviction policies to better handle general cache access patterns.

Cache orchestration optimization criteria

In Chapter 5, we introduced a method of multi-cache, multi-host cache orchestration called Flux that leverages genetic algorithms to optimize the configurations and placements of caches running on a fleet of hosting servers. This technique, designed as an open framework, allows system administrators to supply their own objective functions which Flux uses to make optimization decisions. Although we demonstrated two possible objective functions, how to best create these is not well researched. Prior work has simply attempted to minimize the aggregate number of misses to the caches which can lead to cache starvation.

Bibliography

[1] Jhonny Mertz and Ingrid Nunes. "Understanding Application-Level Caching in Web Applications: A Comprehensive Introduction and Survey of State-of-the-Art Approaches ". In: *ACM Computing Surveys* 50.6 (Nov. 2017), pp. 1–34. ISSN: 0360-0300. DOI: 10.1145/3145813.

[2] Ao Wang et al. "InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache". In: *Proc. Conf. on File and Storage Technologies (FAST'20)*. Feb. 2020, pp. 267–281. ISBN: 978-1-939133-12-0.

[3] Juncheng Yang, Yao Yue, and K. V. Rashmi. "A Large-Scale Analysis of Hundreds of In-Memory Key-Value Cache Clusters at Twitter". In: *ACM Transactions on Storage* 17.3 (Aug. 2021), pp. 1–35. ISSN: 1553-3077. DOI: 10.1145/3468521.

[4] Rajesh Nishtala et al. "Scaling Memcache at Facebook". In: *Proc. Symp. on Networked Systems Design and Implementation (NSDI'13)*. Apr. 2013, pp. 385–398. ISBN: 978-1-931971-00-3.

[5] Juncheng Yang, Yao Yue, and Rashmi Vinayak. "Segcache: A memory-efficient and scalable in-memory key-value cache for small objects". In: *Proc. Symp. on Networked Systems Design and Implementation (NSDI'21)*. Apr. 2021, pp. 503–518. ISBN: 978-1-939133-21-2.

[6] Yue Cheng, Aayush Gupta, and Ali R. Butt. "An In-Memory Object Caching Framework with Adaptive Load Balancing". In: *Proc. of the European Conf. on Computer Systems (EuroSys'15)*. Apr. 2015, pp. 1–16. ISBN: 9781450332385. DOI: 10.1145/2741948.2741967.

[7] Jaewon Kwak et al. "In-Memory Caching Orchestration for Hadoop". In: *Proc. Intl. Symp. on Cluster, Cloud and Grid Computing (CCGrid'16)*. May 2016, pp. 94–97. DOI: 10.1109/CCGrid.2016.73.

[8] Kia Shakiba, Sari Sultan, and Michael Stumm. "Kosmo: Efficient Online Miss Ratio Curve Generation for Eviction Policy Evaluation". In: *Proc. Conf. on File and Storage Technologies (FAST'24)*. Feb. 2024, pp. 89–105.

[9] Hojin Park et al. "Reducing Cross-Cloud/Region Costs with the Auto-Configuring MACARON Cache". In: *Proc. Symp. on Operating Systems Principles (SOSP'24)*. Nov. 2024, pp. 347–368.

[10] Juncheng Yang et al. "FIFO Queues Are All You Need for Cache Eviction". In: *Proc. Symp. on Operating Systems Principles (SOSP'23)*. Oct. 2023, pp. 130–149. ISBN: 9798400702297. DOI: 10.1145/3600006.3613147.

[11] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. “LHD: Improving Cache Hit Rate by Maximizing Hit Density”. In: *Proc. Symp. on Networked Systems Design and Implementation (NSDI'18)*. Apr. 2018, pp. 389–403. ISBN: 978-1-939133-01-4.

[12] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. “Hyperbolic Caching: Flexible Caching for Web Applications”. In: *Proc. USENIX Annual Technical Conf. (USENIX ATC'17)*. July 2017, pp. 499–511. ISBN: 978-1-931971-38-6.

[13] Bin Fan, David G. Andersen, and Michael Kaminsky. “MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing”. In: *Proc. Symp. on Networked Systems Design and Implementation (NSDI'13)*. Apr. 2013, pp. 371–384. ISBN: 978-1-931971-00-3.

[14] Ankita Kejriwal et al. “SLIK: Scalable Low-Latency Indexes for a Key-Value Store”. In: *Proc. USENIX Annual Technical Conf. (USENIX ATC'16)*. June 2016, pp. 57–70. ISBN: 978-1-931971-30-0.

[15] Bojie Li et al. “KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC”. In: *Proc. Symp. on Operating Systems Principles (SOSP'17)*. Oct. 2017, pp. 137–152. ISBN: 9781450350853. DOI: `10.1145/3132747.3132756`.

[16] Redis Labs. *Redis*. `https://redis.io`.

[17] Memcached. *Memcached*. `https://memcached.org`.

[18] Timothy Zhu et al. “Saving Cash by Using Less Cache”. In: *Proc. Workshop on Hot Topics in Cloud Computing (HotCloud'12)*. June 2012.

[19] Nathan Beckmann and Daniel Sanchez. “Talus: A simple way to remove cliffs in cache performance”. In: *Proc. Intl. Symp. on High Performance Computer Architecture (HPCA'15)*. Feb. 2015, pp. 64–75. DOI: `10.1109/HPCA.2015.7056022`.

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

[21] J. Alghazo, A. Akaaboune, and N. Botros. “SF-LRU cache replacement algorithm”. In: *Proc. Workshop on Memory Technology, Design and Testing*. Aug. 2004, pp. 19–24. DOI: `10.1109/MTDT.2004.1327979`.

[22] P. Ranjan Panda et al. “A data alignment technique for improving cache performance”. In: *Proc. Intl. Conf. on Computer Design VLSI in Computers and Processors (ICCD'97)*. Oct. 1997, pp. 587–592. DOI: `10.1109/ICCD.1997.628925`.

[23] D. Thiebaut, J. Wolf, and H. Stone. “Improving Disk Cache Hit-Ratios Through Cache Partitioning”. In: *IEEE Transactions on Computers* 41.06 (June 1992), pp. 665–676. ISSN: 1557-9956. DOI: `10.1109/12.144619`.

[24] Carl A. Waldspurger et al. “Efficient MRC Construction with SHARDS”. In: *Proc. Conf. on File and Storage Technologies (FAST'15)*. Feb. 2015, pp. 95–110. ISBN: 978-1-931971-201.

[25] Ohad Eytan et al. “It's Time to Revisit LRU vs. FIFO”. In: *Proc. Workshop on Hot Topics in Storage and File Systems (HotStorage'20)*. July 2020.

[26] Tencent. *Tencent CBS*. `https://intl.cloud.tencent.com/product/cbs`.

[27] Alibaba. *Alibaba Block Traces*. https://github.com/alibaba/block-traces.

[28] Wikimedia Foundation. *Wikipedia CDN Traces*. https://wikitech.wikimedia.org/wiki/Analytics/Data_Lake/Traffic/Caching.

[29] George Kingsley Zipf. "Relative frequency as a determinant of phonetic change". In: *Harvard Studies in Classical Philology* 40 (1929), pp. 1–95.

[30] L. Breslau et al. "Web caching and Zipf-like distributions: Evidence and implications". In: *Proc. Conf. of the IEEE Computer and Communications Societies (INFOCOM'99)*. Mar. 1999, pp. 126–134. DOI: 10.1109/INFCOM.1999.749260.

[31] Benjamin Berg et al. "The CacheLib Caching Engine: Design and Experiences at Scale". In: *Proc. Symp. on Operating Systems Design and Implementation (OSDI'20)*. Nov. 2020, pp. 753–768.

[32] Brian F. Cooper et al. "Benchmarking cloud serving systems with YCSB". In: *ACM Symposium on Cloud Computing*. Association for Computing Machinery, June 2010, pp. 143–154.

[33] Syed Hasan et al. "Trade-offs in optimizing the cache deployments of CDNs". In: *Proc. Conf. of the IEEE Computer and Communications Societies (INFOCOM'99)*. Apr. 2014, pp. 460–468.

[34] Qi Huang et al. "An analysis of Facebook photo caching". In: *Proc. Symp. on Operating Systems Principles (SOSP'13)*. Nov. 2013, pp. 167–181.

[35] Kunwadee Sripanidkulchai, Bruce Maggs, and Hui Zhang. "An analysis of live streaming workloads on the internet". In: *Proc. ACM SIGCOMM Conf. on Internet Measurement (IMC'04)*. Oct. 2004, pp. 41–54.

[36] Amazon. *Amazon Web Services*. https://aws.amazon.com.

[37] Microsoft. *Microsoft Azure*. https://azure.microsoft.com.

[38] Google. *Google Cloud*. https://cloud.google.com.

[39] IBM. *IBM Cloud*. https://www.ibm.com/cloud.

[40] Cloud Native Computing Foundation. *Kubernetes*. https://kubernetes.io.

[41] Docker. *Docker*. https://www.docker.com.

[42] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. "Write Off-Loading: Practical Power Management for Enterprise Storage". In: *ACM Transactions on Storage* 4.3 (Nov. 2008), pp. 1–23. ISSN: 1553-3077. DOI: 10.1145/1416944.1416949.

[43] R.L. Mattson et al. "Evaluation Techniques for Storage Hierarchies". In: *IBM Systems Journal* 9.2 (1970), pp. 78–117. DOI: 10.1147/sj.92.0078.

[44] F. Olken. *Efficient Methods for Calculating the Success Function of Fixed-Space Replacement Policies*. Tech. rep. LBL-12370. Lawrence Berkeley National Lab (LBNL), May 1981. DOI: 10.2172/6051879.

[45] Qingpeng Niu et al. "PARDA: A Fast Parallel Reuse Distance Analysis Algorithm". In: *Proc. Intl. Parallel and Distributed Processing Symp. (IPDPS'12)*. Aug. 2012, pp. 1284–1294. DOI: 10.1109/IPDPS.2012.117.

[46] Jake Wires et al. "Characterizing Storage Workloads with Counter Stacks". In: *Proc. Symp. on Operating Systems Design and Implementation (OSDI'14)*. Oct. 2014, pp. 335–349. ISBN: 978-1-931971-16-4.

[47] Xiameng Hu et al. "Kinetic Modeling of Data Eviction in Cache". In: *Proc. USENIX Annual Technical Conf. (USENIX ATC'16)*. June 2016, pp. 351–364. ISBN: 978-1-931971-30-0.

[48] Carl Waldspurger et al. "Cache Modeling and Optimization using Miniature Simulations". In: *Proc. USENIX Annual Technical Conf. (USENIX ATC'17)*. July 2017, pp. 487–498. ISBN: 978-1-931971-38-6.

[49] S. Min et al. "LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies". In: *IEEE Transactions on Computers* 50.12 (Dec. 2001), pp. 1352–1361. ISSN: 1557-9956. DOI: `10.1109/TC.2001.970573`.

[50] Nimrod Megiddo and Dharmendra S. Modha. "ARC: A Self-Tuning, Low Overhead Replacement Cache". In: *Proc. Conf. on File and Storage Technologies (FAST'03)*. Mar. 2003, pp. 115–130.

[51] Yazhuo Zhang et al. "SIEVE is Simpler than LRU: an Efficient Turn-Key Eviction Algorithm for Web Caches". In: *Proc. Symp. on Networked Systems Design and Implementation (NSDI'24)*. Apr. 2024, pp. 1229–1246.

[52] Kia Shakiba and Michael Stumm. "PaperCache: In-Memory Caching with Dynamic Eviction Policies". In: *Proc. Workshop on Hot Topics in Storage and File Systems (HotStorage'25)*. July 2025.

[53] GNU. *The GNU Allocator*. `https://www.gnu.org/software/libc/manual/html_node/The-GNU-Allocator.html`.

[54] jemalloc. *jemalloc*. `https://jemalloc.net`.

[55] Laszlo A. Belady. "A study of replacement algorithms for a virtual-storage computer". In: *IBM Systems Journal* 5.2 (1966), pp. 78–101.

[56] Rabin A Sugumar and Santosh G Abraham. "Efficient simulation of caches under optimal replacement with applications to miss characterization". In: *Proc. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'93)*. June 1993, pp. 24–35.

[57] 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)*. June 2011, pp. 43–54.

[58] Shaul Dar et al. "Semantic data caching and replacement". In: *Proc. Intl. Conf. on Very Large Data Bases (VLDB'96)*. Sept. 1996, pp. 330–341.

[59] Gerhard Hasslinger et al. "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)*. May 2018, pp. 1–6. DOI: `10.23919/WIOPT.2018.8362880`.

[60] Dhruv Matani, Ketan Shah, and Anirban Mitra. "An O(1) Algorithm for Implementing the LFU Cache Eviction Scheme". In: *arXiv preprint arXiv:2110.11602* (Oct. 2021).

[61] Fernando J Corbato. *A Paging Experiment with the Multics System*. Tech. rep. Massachusetts Institute of Technology, 1968.

[62] Song Jiang, Feng Chen, and Xiaodong Zhang. "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement." In: *Proc. USENIX Annual Technical Conf. (USENIX ATC'05)*. Apr. 2005, pp. 323–336.

[63] Song Jiang and Xiaodong Zhang. "LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance". In: *Proc. Intl. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'02)*. June 2002, pp. 31–42. ISBN: 1581135319. DOI: 10.1145/511334.511340.

[64] Ramakrishna Karedla, J Spencer Love, and Bradley G Wherry. "Caching strategies to improve disk system performance". In: *Computer* 27.3 (Mar. 1994), pp. 38–46.

[65] Yinyin Wang et al. "LR-LRU: A PACS-oriented intelligent cache replacement policy". In: *IEEE Access* 7 (2019), pp. 58073–58084.

[66] Chunhua Li et al. "SS-LRU: a smart segmented LRU caching". In: *Proc. of the Design Automation Conf. (DAC'22)*. Aug. 2022, pp. 397–402.

[67] Yuanyuan Zhou, James Philbin, and Kai Li. "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches." In: *Proc. USENIX Annual Technical Conf. (USENIX ATC'01)*. June 2001, pp. 91–104.

[68] Zhenyu Song et al. "Learning relaxed belady for content distribution network caching". In: *Proc. Symp. on Networked Systems Design and Implementation (NSDI'20)*. Feb. 2020, pp. 529–544.

[69] Juncheng Yang et al. "GL-Cache: Group-level learning for efficient and high-performance caching". In: *Proc. Conf. on File and Storage Technologies (FAST'23)*. Feb. 2023, pp. 115–134.

[70] Redis Labs. *Redis serialization protocol specification*. https://redis.io/docs/latest/develop/reference/protocol-spec.

[71] Peter J. Denning. "The working set model for program behavior". In: *Proc. Symp. on Operating Systems Principles (SOSP'67)*. Jan. 1967, pp. 15.1–15.12.

[72] Bin Fan et al. "Cuckoo filter: Practically better than bloom". In: *Proc. Intl. Conf. on emerging Networking Experiments and Technologies (CoNEXT'14)*. Dec. 2014, pp. 75–88.

[73] Yu Zhang et al. "OSCA: An Online-Model Based Cache Allocation Scheme in Cloud Block Storage Systems". In: *Proc. USENIX Annual Technical Conf. (USENIX ATC'20)*. July 2020, pp. 785–798.

[74] David Eklov and Erik Hagersten. "StatStack: Efficient modeling of LRU caches". In: *Proc. Intl. Symp. on Performance Analysis of Systems & Software (ISPASS'10)*. Mar. 2010, pp. 55–65. DOI: 10.1109/ISPASS.2010.5452069.

[75] David K Tam et al. "RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations". In: *Proc. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. Mar. 2009, pp. 121–132.

[76] Xiameng Hu et al. “LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache”. In: *Proc. USENIX Annual Technical Conf. (USENIX ATC’15)*. July 2015, pp. 57–69. ISBN: 978-1-931971-225.

[77] Trausti Saemundsson et al. “Dynamic Performance Profiling of Cloud Caches”. In: *Proc. Symp. on Cloud Computing (SOCC’14)*. Nov. 2014, pp. 1–14. ISBN: 9781450332521. DOI: 10.1145/2670979.2671007.

[78] Asaf Cidon et al. “Dynacache: Dynamic Cloud Caching”. In: *Proc. Workshop on Hot Topics in Cloud Computing (HotCloud’15)*. July 2015.

[79] 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)*. June 2018, pp. 84–95. ISBN: 9781450358019. DOI: 10.1145/3210563.3210571.

[80] Asaf Cidon et al. “Cliffhanger: Scaling Performance Cliffs in Web Memory Caches”. In: *Proc. Symp. on Networked Systems Design and Implementation (NSDI’16)*. Mar. 2016, pp. 379–392. ISBN: 978-1-931971-29-4.

[81] Asaf Cidon et al. “Memshare: A Dynamic Multi-tenant Key-value Cache”. In: *Proc. USENIX Annual Technical Conf. (USENIX ATC’17)*. July 2017, pp. 321–334. ISBN: 978-1-931971-38-6.

[82] Ali Ghodsi et al. “Dominant resource fairness: Fair allocation of multiple resource types”. In: *Proc. Symp. on Networked Systems Design and Implementation (NSDI’11)*. Mar. 2011.

[83] Raj Parihar et al. “Protection and Utilization in Shared Cache through Rationing”. In: *Proc. Intl. Conf. on Parallel Architectures and Compilation (PACT’14)*. Aug. 2014, pp. 487–488.

[84] Sean Barker et al. ““Cut me some slack”: latency-aware live migration for databases”. In: *Proc. Intl. Conf. on Extending Database Technology (EDBT’12)*. Mar. 2012, pp. 432–443.

[85] John Ousterhout et al. “The RAMCloud Storage System”. In: *ACM Transactions on Computer Systems* 33.3 (Aug. 2015). ISSN: 0734-2071.

[86] Junbin Kang et al. “Remus: Efficient Live Migration for Distributed Databases with Snapshot Isolation”. In: *Proc. Intl. Conf. on Management of Data (SIGMOD’22)*. June 2022, pp. 2232–2245.

[87] Xingda Wei et al. “Replication-driven live reconfiguration for fast distributed transaction processing”. In: *Proc. USENIX Annual Technical Conf. (USENIX ATC’17)*. July 2017, pp. 335–347.

[88] Chinmay Kulkarni et al. “Rocksteady: Fast Migration for Low-latency In-memory Storage”. In: *Proc. Symp. on Operating Systems Principles (SOSP’17)*. Oct. 2017, pp. 390–405. ISBN: 9781450350853. DOI: 10.1145/3132747.3132784.

[89] Jiewen Hai et al. “Fulva: Efficient Live Migration for In-Memory Key-Value Stores with Zero Downtime”. In: *Proc. Symp. on Reliable Distributed Systems (SRDS’19)*. Oct. 2019, pp. 83–92. DOI: 10.1109/SRDS47363.2019.00019.

[90] Zeying Zhu, Yibo Zhao, and Zaoxing Liu. “In-Memory Key-Value Store Live Migration with NetMigrate”. In: *Proc. Conf. on File and Storage Technologies (FAST’24)*. Feb. 2024, pp. 209–224. ISBN: 978-1-939133-38-0.

[91] Xiaojun Guo et al. “FLOWS: Balanced MRC profiling for heterogeneous object-size cache”. In: *Proc. of the European Conf. on Computer Systems (EuroSys’24)*. Apr. 2024, pp. 421–440.

[92] Benjamin Reed and Darrell D. E. Long. “Analysis of Caching Algorithms for Distributed File Systems”. In: *SIGOPS Operating Systems Review* 30.3 (July 1996), pp. 12–21. ISSN: 0163-5980. DOI: 10.1145/230908.230913.

[93] Ailing Yu et al. “DFShards: Effective Construction of MRCs Online for Non-Stack Algorithms”. In: *Proc. Intl. Conf. on Computing Frontiers (CF’21)*. May 2021, pp. 63–72. ISBN: 9781450384049. DOI: 10.1145/3457388.3458810.

[94] Xiaoming Gu and Chen Ding. “A Generalized Theory of Collaborative Caching”. In: *Proc. Intl. Symp. on Memory Management (ISMM’12)*. June 2012, pp. 109–120. ISBN: 9781450313506. DOI: 10.1145/2258996.2259012.

[95] Sari Sultan et al. “TTLs matter: Efficient cache sizing with TTL-aware miss ratio curves and working set sizes”. In: *Proc. of the European Conf. on Computer Systems (EuroSys’24)*. Apr. 2024, pp. 387–404. ISBN: 9798400704376. DOI: 10.1145/3627703.3650066.

[96] U.S. Securities and Exchange Commission (SEC). *EDGAR Log File Data Sets*. https://www.sec.gov/about/data/edgar-log-file-data-sets.

[97] James Ryans. “Using the EDGAR log file data set”. In: (2017). URL: https://dx.doi.org/10.2139/ssrn.2913612.

[98] Twitter. *GitHub: twitter/cache-trace*. https://github.com/twitter/cache-trace.

[99] Xiameng Hu et al. “Fast Miss Ratio Curve Modeling for Storage Cache”. In: *ACM Transactions on Storage* 14.2 (Apr. 2018), pp. 1–34. ISSN: 1553-3077. DOI: 10.1145/3185751.

[100] The kernel development community. *The /proc Filesystem*. https://www.kernel.org/doc/html/latest/filesystems/proc.html.

[101] Nathan Beckmann and Daniel Sanchez. “Modeling cache performance beyond LRU”. In: *Proc. Intl. Symp. on High Performance Computer Architecture (HPCA’16)*. Mar. 2016, pp. 225–236. DOI: 10.1109/HPCA.2016.7446067.

[102] E. Berg and E. 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)*. Mar. 2004, pp. 20–27. DOI: 10.1109/ISPASS.2004.1291352.

[103] D. Thiebaut. “On the Fractal Dimension of Computer Programs and its Application to the Prediction of the Cache Miss Ratio”. In: *IEEE Transactions on Computers* 38.7 (July 1989), pp. 1012–1026. DOI: 10.1109/12.30852.

[104] Nam Duong et al. “Improving Cache Management Policies Using Dynamic Reuse Distances”. In: *Proc. Intl. Symp. on Microarchitecture (MICRO’12)*. Dec. 2012, pp. 389–400. DOI: 10.1109/MICRO.2012.43.

[105] Muhammad Bilal and Shin-Gak Kang. “Time Aware Least Recent Used (TLRU) cache management policy in ICN”. In: *Proc. Intl. Conf. on Advanced Communication Technology (ICACT’14)*. Feb. 2014, pp. 528–532. DOI: 10.1109/ICACT.2014.6779016.

[106] Gil Einziger, Roy Friedman, and Ben Manes. "TinyLFU: A Highly Efficient Cache Admission Policy". In: *ACM Transactions on Storage* 13.4 (Nov. 2017), pp. 1–31. ISSN: 1553-3077. DOI: 10.1145/3149371.

[107] Tiago B. G. Perez, Xiaobo Zhou, and Dazhao Cheng. "Reference-Distance Eviction and Prefetching for Cache Management in Spark". In: *Proc. Conf. on Parallel Processing (ICPP'18)*. 2018, pp. 1–10.

[108] Masamichi Takagi and Kei Hiraki. "Inter-Reference Gap Distribution Replacement: An Improved Replacement Algorithm for Set-Associative Caches". In: *Proc. Intl. Conf. on Supercomputing (ICS'04)*. June 2004, pp. 20–30. ISBN: 1581138393. DOI: 10.1145/1006209.1006213.

[109] John T. Robinson and Murthy V. Devarakonda. "Data Cache Management Using Frequency-Based Replacement". In: *Proc. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'90)*. Apr. 1990, pp. 134–142. ISBN: 0897913590. DOI: 10.1145/98457.98523.

[110] Sanle Zhao et al. "LAShards: Low-overhead and Self-adaptive MRC Construction for Non-stack Algorithms". In: *IEEE Transactions on Computers* (2025), pp. 1–14.

[111] Luis N Vicente and Paul H Calamai. "Bilevel and Multilevel Programming: A Bibliography Review". In: *Journal of Global optimization* 5.3 (1994), pp. 291–306.

[112] Athanasios Migdalas, Panos M Pardalos, and Peter Värbrand. *Multilevel Optimization: Algorithms and Applications*. Vol. 20. Springer Science & Business Media, 2013.

[113] Ryo Sato, Mirai Tanaka, and Akiko Takeda. "A gradient method for multilevel optimization". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 7522–7533.

[114] Asaf Cidon et al. "Dynacache: Dynamic Cloud Caching". In: *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud-15)*. July 2015.

[115] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[116] Yuri Rabinovich and Avi Wigderson. "Techniques for Bounding the Convergence Rate of Genetic Algorithms ". In: *Random Structures & Algorithms* 14.2 (Feb. 1999), pp. 111–138.

[117] Severino F. Galán, Ole J. Mengshoel, and Rafael Pinter. "A Novel Mating Approach for Genetic Algorithms". In: *Evolutionary Computation* 21.2 (2013), pp. 197–229.

[118] Brad L Miller, David E Goldberg, et al. "Genetic Algorithms, Tournament Selection, and the Effects of Noise ". In: *Complex systems* 9.3 (1995), pp. 193–212.

[119] Ka-Ho Chow et al. "Atlas: Hybrid Cloud Migration Advisor for Interactive Microservices". In: *Proc. of the European Conf. on Computer Systems (EuroSys'24)*. Apr. 2024, pp. 870–887. DOI: 10.1145/3627703.3629587.

[120] Zhongni Zheng et al. "An approach for cloud resource scheduling based on Parallel Genetic Algorithm". In: *Proc. Intl. Conf. on Computer Research and Development (ICCRD'11)*. Vol. 2. 2011, pp. 444–447.

[121] Jianhua Gu et al. "A new resource scheduling strategy based on genetic algorithm in cloud computing environment." In: *J. Comput.* 7.1 (2012), pp. 42–52.

[122] Windhya Rankothge et al. "Optimizing resource allocation for virtualized network functions in a cloud center using genetic algorithms". In: *IEEE Transactions on Network and Service Management* 14.2 (2017), pp. 343–356.

[123] Suhas Jayaram Subramanya et al. "COpter: Efficient Large-Scale Resource-Allocation via Continual Optimization". In: *Proc. 31st Symp. on Operating Systems Principles (SOSP'25)*. Oct. 2025, pp. 322–340.

[124] Deepak Narayanan et al. "Solving Large-Scale Granular Resource Allocation Problems Efficiently with POP". In: *Proc. 28st Symp. on Operating Systems Principles (SOSP'21)*. Oct. 2021, pp. 521–537.

[125] Yaoyao Ding et al. "IOS: Inter-Operator Scheduler for CNN Acceleration". In: *Proc. of Machine Learning and Systems*. Ed. by A. Smola, A. Dimakis, and I. Stoica. Vol. 3. 2021, pp. 167–180.

[126] Haoran Qiu et al. "FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices". In: *Symp. on Operating Systems Design and Implementation (OSDI'20)*. Nov. 2020, pp. 805–825.

[127] Romil Bhardwaj et al. "Cilantro: Performance-Aware Resource Allocation for General Objectives via Online Feedback". In: *Symp. on Operating Systems Design and Implementation (OSDI'23)*. July 2023, pp. 623–643.

[128] Dulcardo Arteaga et al. "CloudCache: On-demand Flash Cache Management for Cloud Computing". In: *Proc. Conf. on File and Storage Technologies (FAST'16)*. Feb. 2016, pp. 355–369.

[129] Chinmay Kulkarni et al. *Rocksteady artifact*. `https://github.com/utah-scs/RAMCloud/tree/rocksteady-sosp2017`. 2017.

[130] Violet Xinying Chen and John N Hooker. "A guide to formulating fairness in an optimization model". In: *Annals of Operations Research* 326.1 (2023), pp. 581–619.

[131] Hitesh Ballani et al. "Towards predictable datacenter networks". In: *Proc. ACM SIGCOMM Conf. (SIGCOMM'11)*. Aug. 2011, pp. 242–253.

[132] Chuanxiong Guo et al. "SecondNet: A data center network virtualization architecture with bandwidth guarantees". In: *Proc. Intl. Conf. on Emerging Networking Experiments and Technologies (CoNEXT'10)*. Nov. 2010, pp. 1–12.

[133] Anshul Rai, Ranjita Bhagwan, and Saikat Guha. "Generalized resource allocation for the cloud". In: *Proc. Symp. on Cloud Computing (SoCC'12)*. Oct. 2012, pp. 1–12.