Machine learning for flash caching in bulk storage systems

Daniel Lin-Kit Wong

CMU-CS-24-152
September 2024

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Gregory R. Ganger, Chair
David G. Andersen
Nathan Beckmann
Daniel S. Berger (Microsoft Research & University of Washington)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work is supported in part by the National Science Foundation under grant CNS1956271 and the Parallel Data Lab consortium members (Amazon, Datadog, Google, Hitachi, Honda, IBM Research, Intel Corporation, Jane Street, Meta, Microsoft Research, Oracle, Pure Storage, Salesforce, Samsung, Two Sigma, Western Digital and VMware.)

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: flash caching, machine learning for caching, machine learning for systems, bulk storage systems

Soli Deo gloria
Glory to God alone

Abstract

Flash caches are used to reduce peak backend load for throughput-constrained data center services, reducing the total number of backend servers required. Bulk storage systems are a large-scale example; backed by high-capacity but low-throughput hard disks, they use flash caches to provide a cost-effective storage layer underlying everything from blobstores to data warehouses.

However, flash caches must address flash's limited write endurance by limiting the number of flash writes to avoid premature wear-out. Thus, most flash caches rely on admission policies to filter cache insertions and maximize the workload-reduction value of each write.

This dissertation evaluates and demonstrates potential uses of ML in place of traditional heuristic cache management policies for flash caches in bulk storage systems. The most successful elements of my research are embodied in a flash cache system called Baleen, which uses coordinated ML admission and prefetching to reduce peak backend load. After learning painful lessons with early ML policy attempts, I exploit a new cache residency model (*episodes*) to guide model training. I focus on optimizing an end-to-end metric (*Disk-head Time*) that measures backend load more accurately than IO miss rate or byte miss rate. Evaluation using 7-day Meta traces from 7 storage clusters shows that Baleen reduces *Peak Disk-head Time* (and hence backend hard disks required) by 12% over state-of-the-art policies for a fixed flash write rate constraint.

I present a TCO (total cost of ownership) formula quantifying the costs of additional flash writes against reductions in Peak Disk-head Time in terms of flash drives and hard disks needed. Baleen-TCO chooses optimal flash write rates and reduces estimated TCO by 17%.

Workloads change over time, requiring that caches adapt to maintain performance. I present a strategy for peak load reduction that adapts selectivity to load levels. I also evaluated workload drift and its impact on ML policy performance on 30-day Meta traces.

Baleen is the result of substantial exploration and experimentation with ML for caching. I present lessons learned from additional strategies considered and explain why they saw limited success on our workloads. These include enhancements for ML-based eviction, more complex ML models, and optimizing the use of DRAM in hybrid caches. I also present lessons from ML production deployments.

Code and traces are available via https://www.pdl.cmu.edu/CILES/. These include our 7-day traces which were the most extensive public collection of traces from a production bulk storage system at the time of writing.

Acknowledgments

The acknowledgments section is my favorite part of a dissertation to read. The rest of this dissertation tells you a story about my research; the acknowledgments offer a glimpse into the human side of the PhD and all that went into it.

The advisor The PhD is an apprenticeship, and it was my honor to do mine with Greg. If you have not met him, imagine a really tall guy standing on a table in a lecture hall wearing a blue Michigan polo, khaki shorts and a denim PDL jacket. I am continually learning from him: not just how to do research and ask the right questions, but also from his personal example. Greg is not by nature showy (unless basketball or Michigan is involved), which meant that I only learned how smart he was a couple of years in. He's hands-on without needing to see the code: more than once, he diagnosed some bug in my code off a small anomaly in a graph I showed him. He has mastered the art of nuance, of showing off work while being humble, of being firm in a gentle manner. Perhaps, then, it is fitting that his trademark answer to questions in class is: "It depends." (I now catch myself saying that to junior students, in one of the many habits I have unconsciously picked up from him.) I am glad, however, that I could always depend on Greg over the past 7 years, whether through personal losses, a global pandemic, and (literally) fracturing my spine. Greg never shies away from doing the right things in the right way, and he does it so smoothly that outsiders might think it effortless. But the truth is that Greg shows up even more than his students: during our first paper deadline, he came to the office and left only 2 days later, while we mere mortals had to take breaks to sleep. Despite being a busy senior professor, he makes sure to make time for his students. You can take Greg out of a classroom, but you can never take the teacher out of him. He always has a lesson for you, whether you are a high school student, an undergrad, a master's student, a PhD student, a young professor, or a successful software engineer who graduated 20 years ago. Greg, I am so glad that I got the chance to spend the last 7 years with you, and I can imagine you'll be giving me career and parenting tips at a FAST reunion many years down the road.

The committee I first worked with Dave Andersen when we were part of the ISTC-VCS working on systems for machine learning. He has been an inspiring figure who always injects energy into the discussion and the room. It was during his OS class that I first started dabbling in ML for caching. Thank you too, Dave, for letting me join FAWNLunch, where I found solace and company during a challenging part of my PhD.

That same class project led me to Daniel Berger, who has been a relentless source of optimism and who was instrumental to the research that became Baleen (and came up with the name too). I am grateful for how he helped me work out the details of Baleen and for the different perspectives that he brings to the discussion. Thanks, Daniel, for our walk in the woods in Redmond.

Nathan Beckmann helped me gain a deeper appreciation of the intricacies of systems and got me to develop a habit of doing back-of-the-envelope calculations to check through my work and develop my own internal mental model.

The other research mentors and collaborators The Parallel Data Lab has been an amazing home and a unique experience. Thank you to Michael Kozuch, Michael Kaminsky, Babu, Majd, George Amvrosiadis and Rashmi Vinayak for all the help and support you've given me along the way. The PDL has opened doors to collaborations including with Meta, without which the work presented in this dissertation would have been impossible. I owe a big thank you to Hao Wu for working closely with me to get me the data that I needed. Thank you too to Sathya Gunasekar, Jimmy Lu, Qing Zhao, Snehal Khandkar, and Abhinav Sharma from Meta.

My Google internship was also an important part of my PhD. Thank you to Peter Ma, Sudip Roy, Yanqi Zhou, Mangpo Phothilimthana and James Laudon for hosting me first as an intern and then a Student Researcher.

It's been a long road getting to here. Thank you Prof Leong Hon Wai, Ben Leong, Robert Watson, and Ng See Kiong for giving me my first steps into research: of being crushed, of experiencing the moment "Why am I here? What am I doing here?", and eventually feeling the irresistible pull to do the PhD.

The students My fellow students at Carnegie Mellon walked with me through the PhD. Angela was the first PhD student that I worked with at CMU and my closest student collaborator. Within a month of me arriving at CMU, we were working on a NSDI submission, the first of many. Paper deadlines with Angela and Greg remain one of the highlights of my PhD experience. Thank you, Angela, for always being so encouraging and supportive of my endeavors. Thomas was my second collaborator, and his wit and sense of humor were so endearing that I had no choice but to follow him on to his startup. Giulio and Chris rounded out the ISTC-VCS gang. We all spent countless hours spent together in the CIC and then later in GHC 9009, memories that will remain with me for a long time.

I flitted between reading groups before finding a home in FAWN. Anuj's jokes never failed to crack me up, while I enjoyed Sol's company and got to learn about

Rust. Thank you Saurabh for taking time to listen to me and help me with my research. Thank you to Aaron, Andrew, Junwoo and Rajat for your help and for paving the way ahead for me to follow. Thank you Sara and Juncheng for listening to my talks and giving me feedback, and to Jack, Lilia and Carson. Thank you Ankush, Daiyaan, Hojin, Sanjith, Sarvesh, Suhas, Tim, Ziyue and Theo for being fellow members of Greg's gang. I thank my fellow PDLers for the retreats and poster sessions, and all the time that we spent preparing for them.

The Pittsburgh friends Thank you Tze Hui, Akash, Kat and Darby for being my 'normal friends' and the times snowboarding and climbing together. Thank you to my church in Pittsburgh, OIF: the Dongs, the Chengs, the Lews, the Khoos, the Wus, Isaac, the Fus, the McLaughlins, the Chiangs, the Orrs, Shining, Elynna & Michael, Alex, Dan, Eva & Jason, Fred & Jenny, and everyone else that I missed out.

The staff Thank you Karen Lindenfelser for watching out for me right from the start and letting me in on that crucial secret: Greg is the best boss. Thank you Joan Digney for bearing with my late poster submissions and that poster stuck together from pieces of paper like a serial killer letter. Thank you, Deb Cavlovich, for being so empathetic and a motherly presence in the whirlwind that is CSD. Thank you Matthew Stewart for your help in getting me through the last few hoops of my PhD. Thank you to all the PDL staff who helped me with things little and big, including Chuck Cranor, Chad Dougherty, Jason Boles, and Mitch Franzos.

My friends To my dearest friends around the world, thank you for always rooting for me. Thank you Yi Le and Sarah, Yingda and Yuqun, Stacey and Daniel, Jia Ying and Lumpy, Rachel, Geok Ting, Clement, Yen Seow and all the 09S6K folks. Thank you to my Pandan church family, including Ivan and Ruth, Gabriel and Okta, William and Eunice, and Ben and Carmen.

My family I thank my family and the sacrifices they made to support me in my PhD so far from home. I would like to acknowledge my grandparents, including Nainai and Gong Gong who passed on during my PhD.

Besides finding Greg, the other big match of my PhD was getting engaged to Tze Hui. Tze Hui, I am so thankful that God led us together and so excited for the future we have together.

Finally There are so many more people I would like to thank than there is time and space to write. I hope that one day, I will be able to thank you in person.

Last but not least, I thank God for his almighty grace and mercy upon me, and for granting me the breath and strength to complete this dissertation.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Flash has become an integral part of storage systems, as it offers IO performance orders of magnitude more than hard disks (HDDs) while offering storage densities orders of magnitude greater than dynamic random-access memory (DRAM) for the same cost. Hyperscalars have consolidated their storage needs into bulk storage systems that are backed by HDDs (due to their low cost per GB) and fronted by flash caches that absorb demand and compensate for HDDs' low IOPS and bandwidth capacity. However, flash has one major weakness distinguishing it from other storage media: its low write endurance. Any flash caching policy must be designed with this in mind to avoid premature flash wear-out. We define the flash caching problem as determining which times to fetch, admit and evict items to minimize backend load given a flash write rate limit.

Bulk storage systems are provisioned according to peak demand for Disk-head Time. If the system has insufficient IO capacity, requests queue up and slowdowns occur. If sustained, clients retry requests and failures occur. Thus, bulk storage IO requirements are defined by peak load, which in turn affects total storage cost.

Replacing heuristics with machine learning (ML) in cache management policies is appealing as it offers the opportunity to tailor strategies to the workload and incorporate offline insights into online decisions in a scalable manner. However, applying ML to caching has been difficult since: 1) caching does not map well to problems in other fields such as computer vision or natural language processing that are well-solved by supervised learning, and 2) the nature of caching (in terms of the delayed rewards and how each decision affects subsequent decisions) makes it challenging for reinforcement learning [6, 36, 42, 49, 51, 98]. Furthermore, systems practitioners prioritize understanding the decisions made by ML policies and their failure modes; ML policies need to be introspectable, not just performant [96].

In this thesis, we propose an approach for applying ML to flash caching and gather evidence in support of this statement:

Thesis statement: *ML flash caching policies can reduce total cost in bulk storage systems, but in order to outperform heuristics in well-tuned production systems, they must have a flexible and principled design that can adapt to diverse workloads.*

1.1 The case for ML in flash caching

Large-scale storage continues to be predominantly done with hard disks (HDDs), which provide much more cost-effective storage than flash. However, HDDs have low throughput, and each can generally only perform about 100 IOs per second (IOPS). Modern storage systems rely heavily on flash caches to absorb a substantial fraction of requests and thereby reduce the number of disks needed to satisfy the IO workload.

Although a functional cache can be realized using traditional approaches, which assume items can be admitted to the cache arbitrarily, it is important to consider the differing natures of HDDs and flash SSDs. In particular, the IOPS and bandwidth of HDDs has not kept up with increases in their capacity, making disk time a key goal of flash caching more than average IO latency. Flash, on the other hand, provides orders of magnitude higher IOPS, but it wears out as it is written. As a result, expected SSD lifetime projections assume relatively low average write rate limits, such as "three drive-writes per day", meaning 3N TB of writes to a N TB SSD each day. Manufacturers offer SSDs with even lower endurance (e.g., 1 drive write per day) with correspondingly lower prices. All of this translates to a need for smart admission policies to decide which items get written into cache [5, 21]. Eviction policies cannot substitute for admission policies, since the cost of writes is experienced at admission time, even if the item is immediately evicted afterwards. Popular policies have included random admission and history-based policies that reject items without sufficient recent usage.

Machine learning (ML) policies for flash cache admission have been proposed as a solution for avoiding excessive flash writes. However, caching does not easily map to well-trodden problems in computer vision or natural language processing. In particular, a policy's decision is often affected by its past decisions, and can have synergistic or antagonistic effects on other parts of the system. While in theory this can be addressed with end-to-end and reinforcement learning techniques, in practice, such models require large amounts of human capital and computing resources, and do not necessarily outperform a typical well-tuned production system [6, 36, 42, 49, 51, 98].

Making ML policies introspectable is key to their adoption by systems practitioners [96]. While accurate models are desirable, success also hinges on the correct decisions being posed to the models. *How* one uses ML is key: how to generate training examples from traces, how to arrive at optimal decisions for ML to learn from, which subproblems ML should be applied to, and how to optimize end-to-end systems performance without sacrificing introspectability, debuggability, and efficiency.

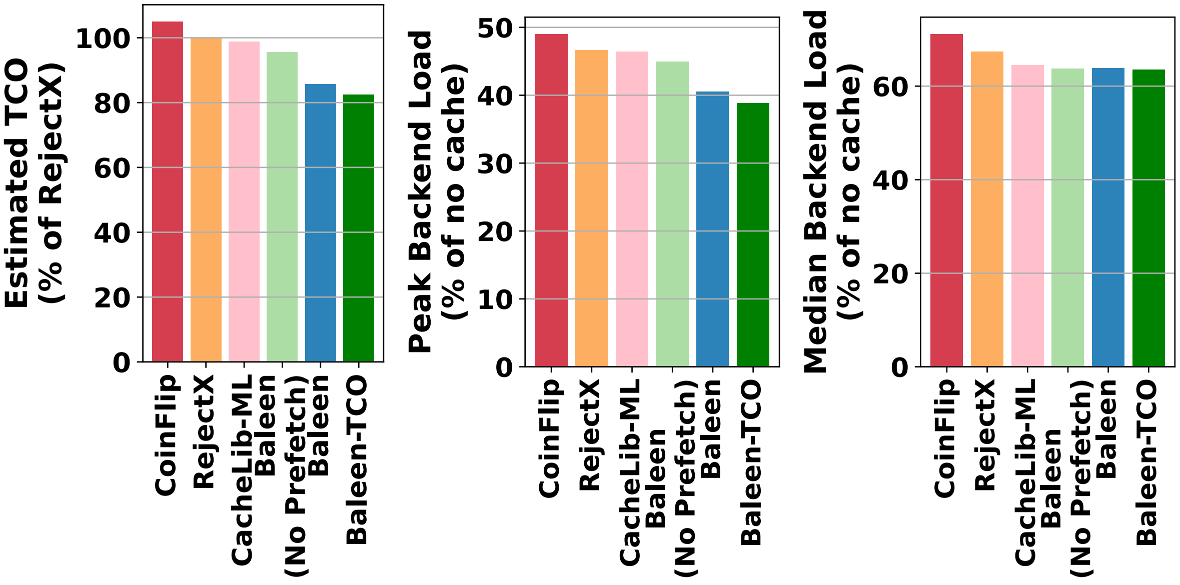
1.2 Baleen: ML for admission and prefetching that optimizes peak HDD load and storage TCO (HDDs+cache)

In this dissertation, we evaluated different areas of cache management in which ML could be applied where there would traditionally be heuristics in a flash cache serving a bulk storage system. We composed the most successful elements of our research, including ML admission and ML prefetching, and embodied them in a flash cache called Baleen.

In Baleen, we decompose the flash caching problem into admission, prefetching, and eviction

(§2.4). This helped us align policy decisions to well-understood and efficient ML techniques for supervised learning. We do, however, want to co-design these different components to reap the full benefits. One may depend on the other to be effective, as we found to be true for ML prefetching and ML admission.

Baleen explores ML policies for flash caches in bulk storage systems. We introduce a new analytic approach for access pattern analysis, based on a cache residency model we call *episodes* (§4.2), which groups accesses that correspond to an item's cache residency if admitted. Our approach provides a more complete view of end-to-end flash caching policy performance, and enables us to efficiently model policy behavior under multiple constraints. This is especially useful for flash caches given that the resource burden of an admission is dominated by its flash writes, which is the same whether the item is admitted at the start or end of the episode. From our approach, we develop *OPT* (4.3), an episode-based approximation of optimal admission and train ML admission policies to imitate OPT. We benchmark them against OPT and other baseline admission policies on seven recent real-world storage cluster traces collected over 3 years.



(a) Estimated TCO

(b) Peak load

(c) Median load

Figure 1.1: Baleen-TCO reduces (estimated) TCO by 17% and peak load by 16% over the best baseline on 7 Meta traces by choosing the optimal flash write rate. IO and byte miss rates were reduced by 14% and 2%. For the default flash write rate, Baleen reduces peak load by 12% over the best baseline.

Baleen is our resulting ML-guided Flash cache policy. We evaluate it by its savings in *Peak Disk-head Time* (§4.1), a measure of peak backend load, and we found that a combination of ML-guided admission and ML-guided prefetching provides the largest improvement. In deploying ML, we learned that determining the right optimization metric is not an easy task; an earlier version of Baleen improved IO hit ratio but had worse end-to-end performance (disk-head time). Optimizing for the right metric in the ML policy improved both introspectability and system performance. We also developed a variant Baleen-TCO, which chooses the optimal flash write rate to optimize our estimate of the total cost of ownership (TCO). This also results in improvements to traditional metrics, reducing IO miss rate by 14% and byte miss rate by 2%.

1.3 Optimizing for peak load, workload drift and other ML caching explorations

Changes in the request distribution over time pose two challenges to ensuring the continued performance of admission policies. First, demand changes over shorter time scales (hours, days, weeks) that give rise to peaks in backend load. Second, workload drift over longer time scales (months, years) that cause model performance to regress over time. We describe strategies to optimize Peak Disk-head Time by choosing the right static parameters, and more advanced strategies for doing by adaptively varying the admission policy selectivity threshold. We perform the first (to the best of our knowledge) analysis of drift in a production caching system, showing how policy performance is affected over time if a model is not retrained.

Caching is an age-old problem, and yet the problem of predicting what will be accessed in the future remains a difficult one to solve. Part of the difficulty is that there are many possible ways in which to approach the problem and that the tractability of caching solutions is dependent on the workload. We evaluated numerous other solutions, such as attempts to improve ML eviction policies, ML for DRAM placement, more advanced models, segment-aware prefetching, prefetching on PUT, and explain why they did not work for our workloads. In exploring the space for improvement, quantifying the room for possible improvement is also important and we sought to break down the gap between Baleen and OPT and attribute Baleen's benefit over existing policies.

1.4 Contributions

We propose and evaluate a principled design for ML flash caching to reduce peak backend load (and thus storage costs), consisting of:

1. **Analytical models to present the right decision options to ML (§4)** We propose the episodes model (§4.2) as a new cache residency model that uses offline information to guide the development and training of ML models. This model simplifies the decision space into decisions on episodes, to the benefit of both humans (increasing introspectability) and ML.
2. **Oracle policies for ML to imitate** We propose OPT (§4.3), an episode-based approximation of optimal admission, and OPT-Range (§4.4), an episode-based approximation of optimal prefetching. This allows us to apply well-understood supervised learning techniques and to understand when performance is limited by ML performance versus other factors.
3. **Metrics and simulators that accurately assess the impact of ML on systems performance** We propose *Peak Disk-head Time* (§4.1) as an end-to-end metric for ML model design, training and evaluation. We show it can be successfully used as an optimization goal during

training and that it matches up with systems metrics (disk utilization) collected in production (§3.4).

4. **TCO (HDDs+cache) formula for flash caching systems (§6)** In flash caching, the reduction in backend load (Peak Disk-head Time), and thus the number of HDDs needed, must be balanced against the number of flash writes which is a cost factor due to additional SSDs required. We derive a TCO (total cost of operation) formula using public data that allows configurations with different flash write rates to be compared. We also extend our system to pick the best target flash write rate for each workload in order to minimize this cost metric.
5. **Decomposition of the flash caching problem into smaller subproblems for ML (§2.4).** We propose a heuristic decomposition of the flash caching problem into admission, prefetching, and eviction. Breaking it down makes it easier for humans to understand the individual components. ML solutions can be evaluated against comparable heuristics or oracle policies for each of these subproblems. We propose that these ML solutions be co-designed and coordinated as they can have synergistic effects on each other.

We evaluate this design through the following studies:

- **Baleen: ML-driven admission and prefetching for flash caches (§5)**

  Baleen is a flash cache that uses coordinated ML admission and ML prefetching to reduce backend load in bulk storage systems. After learning painful lessons from early ML policy attempts, we exploit a new cache residency model (*episodes*) to guide model training, and focus on optimizing the end-to-end metric Disk-head Time which balances IO hit rate and byte hit rate. We also propose *OPT*, an episode-based approximation of optimal admission, which we use both to train ML admission on and as a benchmark. We developed both our BCacheSim simulator and our CacheLib testbed against production system counters. Evaluation on Meta traces from seven storage clusters collected over three years shows that Baleen reduces Peak Disk-head Time (and backend capacity required) by 12% over state-of-the-art policies.
- **Baleen-TCO: balancing the costs of flash writes against reductions in HDD load (§6)**

  We derive a TCO (total cost of operation) formula that can be used to evaluate cost reductions from reducing Peak Disk-head Time against the cost of additional flash writes. We evaluated an extension, Baleen-TCO, which chooses the best flash write rate for each workload (whereas a static flash write rate target is manually set in the status quo). This additional flexibility allows Baleen-TCO to save additional Peak Disk-head Time on some workloads while minimizing flash writes where it is not beneficial, reducing TCO by 17% compared to existing ML policies.
- **Optimizing Peak Disk-head Time (§7)** Ideally, backend capacity is provisioned to match peak load. It thus makes sense to devote special attention to reducing peaks. We describe the importance of optimizing peak load and show a case study that illustrates how optimizing average load can lead to the wrong outcome. We proceeded to evaluate strategies to explicitly optimize peak load. One such strategy was redistributing flash write rate budget from off-peak periods to peak periods. where we varied the ML policy to be much more selective during off-peak periods, saving 8.0% of flash writes with a reduction of 1.4% in peak load.

- **Analyzing workload drift in caching (§8)** Workload drift refers to changes over time in access patterns and the popularity of items, which can cause ML model performance to decrease due to a mismatch between data at training and inference time. We collected additional longer traces and show that drift is substantial and results in a decrease in ML policy performance even over a period of 3 months. We evaluate the efficacy of the common strategy of retraining.
- **ML for eviction (§9.1)** We study the third subproblem in our heuristic decomposition: eviction policies. Existing ML eviction policies have been evaluated in the context of DRAM caches. In contrast, the conventional policy in flash caches has been to use a simple eviction policy (such as LRU or FIFO) in favor of having a complex admission policy. However, given that the time between an item's first and last access before eviction can range from minutes to hours in a flash cache, using a simple eviction policy such as LRU leaves significant potential savings on the table. If items could be evicted immediately after they are last accessed, instead of waiting to leave the cache, this would result in a potential reduction of 11% in average Disk-head Time. We evaluated policies that use episode timespan and episode maximum interarrival time for early eviction.
- **ML for DRAM placement (§9.2)** Conventional wisdom in hybrid cache design is to promote an object to DRAM when there is a hit on it in flash, and to insert DRAM evictions into flash. We consider how DRAM might be used selectively to reduce flash writes, instead of letting every item pass through DRAM. DRAM should be used to gain more information on episodes the policy is unsure about, and to bypass flash entirely for episodes with a very short timespan, or that have a scan or churn pattern.
- **Cache Transformer: exploring more advanced ML models (§9.3)** We rely on Gradient Boosting Machines (GBMs) in Baleen, which are relatively simple models compared to the range of advanced deep neural network architectures available today. We design a Transformer-based caching model and evaluated it, showing that the GBM performs just as well on our workloads.
- **Segment-aware admission and prefetching (§9.4).** We explore how more fine-grained admission policies might further reduce DT, and describe our attempts at doing so and the challenges in achieving those benefits.
- **Prefetching on PUT (§9.6)** We explore the viability of prefetching an item upon PUT, and describe ML models we trained to do so. We show why the problem is challenging and of limited benefit given our workloads.
- **Attributing Baleen's benefit over previous policies and breaking down the gap to OPT (9.5)** We analyze how Baleen was able to achieve better results than previous ML policies (e.g., by introducing size awareness). We show that Baleen still has significant room for improvement, as quantified by the gap of 16% in DT between Baleen and OPT. We attribute some of these to late admissions, and also examine how Baleen is limited by its labels by looking at how a Bayes Optimal classifer would do.

1.5 Outline

The rest of this dissertation is organized as follows:

- Chapter 2 *(Background)* gives background on bulk storage systems in the context of modern data centers, the flash caching problem (balancing absorption of HDD load against limited flash write endurance), and related work.
- Chapter 3 *(Experimental setup)* goes over our datasets, simulator and testbed.
- Chapter 4 *(Episodes & OPT: modeling flash caching and exploring savings in Disk-head Time)* introduces our episodes model for flash cache residencies, our approximation of of optimal for flash admission (OPT) and prefetching (OPT-Range), and our analytical flash cache model.
- Chapter 5 *(Baleen: Training ML policies for flash caching)* describes how we train ML policies for flash admission and prefetching in Baleen, and validated it on production workloads.
- Chapter 7 *(Optimizing for peak load)* investigates the optimization of peak load by choosing algorithm parameters and varying admission selectivity by system load.
- Chapter 8 *(Workload drift in caching)* introduces the concept of workload drift and evaluates ML performance in response to drift over months, as well as analyzing different aspects of retraining, a common mitigation measure.
- Chapter 9 *(Lessons learned from other ML-guided caching explorations)* presents a number of case studies such as ML for flash eviction, ML for DRAM placement, and Cache Transformer.
- Chapter 10 *(Conclusions, lessons learned and future directions)* summarizes the contributions and lessons learned in the course of this dissertation.

Chapter 2

Background

A cache[1] is a smaller and faster storage placed in front of a larger and slower backend storage. Over time, caching has grown to encompass not just setups with different memory speeds (CPU cache and DRAM), but also the same memory at different locations (CDNs) and different storage mediums (flash drives and hard disks). In this dissertation, we focus on flash caching, which has become an integral part of bulk storage systems that are found in hyperscalar data centers in which much of the world's data resides.

2.1 Bulk storage systems in data centers

Tectonic is an example of a bulk storage system, which aggregates persistent storage needs in data centers (e.g., from blobstores and data warehouses). Flash caches are used to reduce the load on the backing HDDs and meet throughput requirements. Other systems have a similar design [28, 58, 69].

Accesses are made to byte ranges within **blocks**. Blocks are mapped to a location on backing HDDs, and subdivided into smaller units called **segments** that can be individually cached. (Tectonic has 8MB blocks and 128 KB segments.) Upon an access, the cache is checked for all segments needed to cover the request byte range. If any are missing, an IO is made to the backing store to fetch them, at which point they can be admitted into the cache.

Each cluster has 10,000s of storage nodes independently serving requests. Each node has 378 TB in HDDs [58], 400 GB in flash cache, and 10 GB in DRAM cache (37,800:40:1). We focus on the scope of the individual node.

2.2 Bulk storage limited by disk-head time (DT)

At scale, hard disks (HDDs) remain the choice of backing store as they are cheaper by 10X per TB over SSDs [54]. Newer HDDs offer increased storage density, resulting in shrinking throughput

[1] The term *cache*, as used in computing, originated at the IBM Systems Journal in 1967 to replace the term "High Speed Buffer" in papers describing the IBM System/360 Model 85 [70]. This was a year after Belady's well-known paper was published, in which the term cache does not actually appear [4].

(IOPS and bandwidth) per GB as more GBs are served by the same disk head.

Disk-head time (defined in §4.1) on backing HDDs is a premium resource, especially with workloads that are more random than sequential. The mechanical nature of HDDs results in a high, size-independent access time penalty (e.g., 10 ms) for positioning the read/write head before bytes are transferred. With a high read rate (e.g., 5.5 ms/MB), a request could take 10 to 70 ms (Fig 2.1).

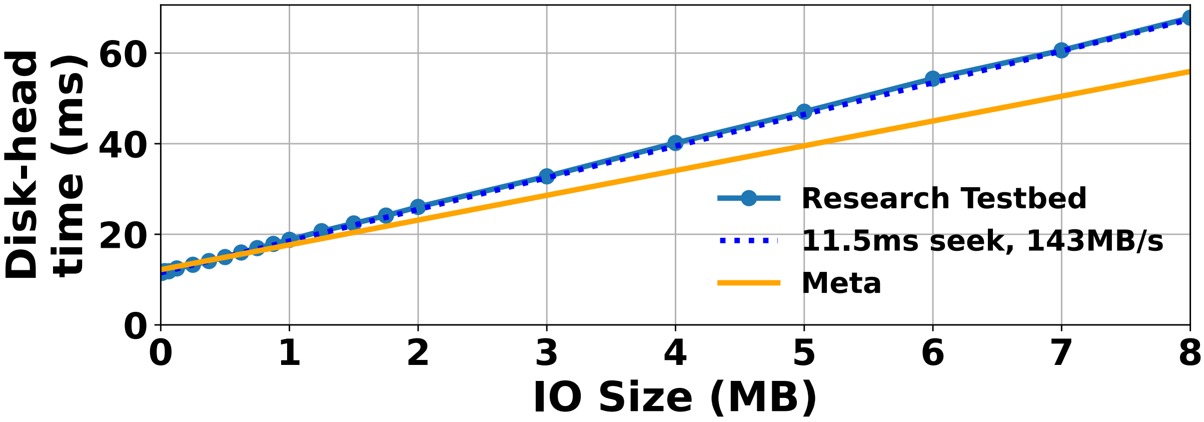


Figure 2.1: Disk-head Time (DT) for one IO. When a HDD performs an IO, the disk head **seeks** before it **reads** data. For tiny IOs, throughput is limited by *IOPS*; for large IOs, by *bandwidth*. DT encompasses both metrics and generalizes to variable-size IOs.

In provisioning bulk storage, peak demand for disk-head time matters most. If the system has insufficient IO capacity, requests queue up, and slowdowns occur. If sustained, clients retry requests and failures occur, affecting user experience. Thus, bulk storage IO requirements are defined by peak load, which in turn affects storage costs.

We will describe in §4.1 the details of our proposal to use Disk-head Time as a metric, and evaluate it using production traces.

2.3 Flash caches absorb HDD load but have limited write endurance

Flash caching plays an important role in absorbing backend load, compensating for disk-head time limitations of the underlying HDDs. This setup enables resource-efficient storage for workloads that exceed the throughput requirements of HDDs but which are infeasible to store using flash alone. With the trends towards higher density HDDs and fewer bytes per HDD spindle, flash caches unlock more usable bytes per spindle.

While managing throughput is the primary goal of flash caching, tail latency can improve as a result of reduced backend contention [97]. Flash caches also add flexibility for matching system throughput to ever-growing demand, as it is easier to enlarge flash caches than swap out existing HDDs. When AI training put pressure on storage bandwidth at Meta, the solution was to add a disaggregated flash caching tier [103].

Flash does not have access setup penalties, but does have wearout that translates into long-term average-write-rate limits. SSD manufacturers rate their drives' endurance in terms of drive-writes per day (DWPD) over their warranty period. Caching is an especially challenging workload for flash, since items will have widely varying lifetimes, resulting in a usage pattern closer to random I/Os than large sequential writes. Items admitted together may not be evicted at the same time, worsening write amplification. Writing every miss into flash would cause it to wear out prematurely. Admitting everything requires up to 492 MB s^{-1} or 43 DWPD for our traces; for an SSD rated at 3 DWPD over 5 years, this means a reduced lifetime of just 4 months (i.e., 14 × as fast). One solution is SSD capacity overprovisioning, but this can rapidly become a dominant part of the total storage costs [5, 96].

2.3.1 Introducing admission policies and baselines (RejectX, CoinFlip)

Flash caches leverage **admission policies** (APs) to decide if items should be inserted into the flash cache or discarded, and have simple eviction policies (LRU, FIFO) to minimize write amplification [5]. Like eviction policies, admission policies weigh the benefit of hits from new items against lost hits from evicted items. They must also weigh the write cost of admitting the new item against other past or future items. Policies have an *admission threshold* that can be varied to achieve the target flash write rate. We provide some examples.

- **CoinFlip (baseline)** On a miss, segments for an access are either all admitted, or not at all, with probability p. This simple policy does not need tracking of past items seen.
- **RejectX (baseline)** rejects a segment the first X times it is seen. Past accesses are tracked using probabilistic data structures similar to Bloom filters. We use $X = 1$ and vary the window size of past accesses to achieve the desired write rate. Both Meta [5] and Google [96] used this prior to switching to more complex policies.
- **ML admission policies** use offline features to make decisions in addition to online features such as past access counts. An ML model can be trained offline based on a trace (as we do), or online using reinforcement learning.

2.4 Decomposing the caching problem

We define the caching problem as determining which times we should fetch, admit, and evict each segment to minimize the backend's DT given a flash write rate limit.

We propose a heuristic decomposition of this problem into three subproblems: admission, prefetching, and eviction. This makes it easier to reason about the optimal solutions to each sub-problem and the training and behavior of ML solutions for each part. Making ML solutions easier to train, understand, and debug mitigates production engineers' common criticism of their blackbox nature [67].

Admission: Whether to admit something into cache in anticipation of future hits that reduce DT Here, we trade off the disk-head time saved against the write rate used from caching an item. We model this as a binary classifier, where misses are admitted if the output probability exceeds the policy threshold. We also considered regression models (e.g., predicting no. of expected hits). Such models eliminate the threshold parameter, but we found they perform worse end-to-end, perhaps because their loss functions incentivize performance at all thresholds (write rates) rather than just those at the boundary.

Prefetching: Whether to prefetch extra segments outside the current access range (which was a miss) Here, we trade off DT saved from hits on the first accesses against the additional time spent in cache, and for incorrect prefetches, the DT wasted and the opportunity cost of the wasted flash write rate. We further decompose the prefetching problem into a) deciding what segments to prefetch and b) when to prefetch (whether the expected benefit exceeds the cost, taking into account the possibility of mispredictions).

Eviction: Which segment in the cache to pick for eviction upon an admission Here, one can employ existing approaches for non-flash caches, including ML-based policies. In Baleen, we employ a simple eviction policy (in our case, LRU) as is used in production systems. We evaluated ML-based flash-aware eviction policies but left them out of Baleen given the low savings in DT.

2.5 Limitations of existing systems

Existing works are often:

- **Not modular.** Without a modular design, the system can be oversimplified and miss out on key design considerations [21], or else veer towards too much complexity and be difficult to debug and reason about.
- **Optimizing for intermediate metrics.** Many systems optimize hit rate [8, 18, 21, 38, 63, 72], bandwidth [68, 69] or write rate without considering the larger system the cache is part of. This makes them less performant and robust.
- **Not focused on peak periods.** Almost all systems report averages, giving less accurate assessments of system performance, as bad performance at peak can be covered up by good (but ultimately unhelpful) off-peak performance. To our knowledge, only one other system evaluates load at peak [69].
- **Not co-designed.** Many systems focus on a single aspect like flash admission [5, 18, 21] or eviction [3, 8, 38, 44, 63, 68, 69, 72, 77, 106] without considering the effect of one part on another, in the belief that their benefits will be fully retained when applied with other techniques. To our knowledge, only two other systems evaluate multiple subproblems, such as admission and eviction [1] or admission and prefetching [97].

2.6 Related work

Production flash caching systems CacheSack [96, 97] optimizes admission policies for the flash cache in front of Google's bulk storage system, Colossus. This design shares Baleen's objectives of co-optimizing backend disk reads and flash write endurance. CacheSack partitions traffic into categories using metadata and user annotations, assigning probabilities to each of 4 simple admission policies for each category by solving a fractional knapsack problem. This offline approach has slower reaction times than Baleen, and does not evaluate load at peak. Meta's Tectonic bulk storage system uses a CacheLib-managed flash cache, with an ML admission policy that does not use episodes and does not perform prefetching (which we evaluate against as the CacheLib-ML baseline). § 5 shows that this approach is significantly less effective than Baleen. Kangaroo [53] improves CacheLib's small object cache, and is orthogonal to Baleen, which improves performance for large objects. Amazon's AQUA [2] also fills a similar role for Redshift (data warehouse), acting as an off-cluster flash caching layer with S3 as the backing store. Bulk storage systems backed by HDDs and fronted by cache servers can also be found at Alibaba Cloud [39] and Tencent [101].

Non-ML flash admission policies CacheLib [5] is Meta's general-purpose caching library and includes random and RejectX admission policies for flash caches. Section 2.3.1 introduces RejectX. Section 5.3 extensively compares Baleen to random (CoinFlip) and RejectX. LARC [30] is equivalent to RejectX and was the default admission policy used at Google prior to CacheSack. TinyLFU [18] proposed a frequency-based admission policy that leverages probabilistic data structures for compact history representation. Baleen adds ML, size-awareness, disk performance goals, and prefetching over TinyLFU.

ML-based flash caching policies Flashield [21] addresses the lack of information on flash admission candidates by putting them in a DRAM buffer first. The item's usage history is used to generate features for a support vector machine classifier. However, we found this approach infeasible as DRAM lifetimes are too short in practice (see § 5.4.2). More targeted applications of ML aim to exclude one-hit-wonders [80] or items that have no reads [102]. Reinforcement learning has also been used to train a feedforward neural network for admission policies on CDNs, given a broad set of features [33]. Baleen adds more flexible admission policies, size-awareness, disk performance goals, and prefetching over these works. Early work on flash caching focused on flash-friendly eviction policies [61]. Recent work instead uses simpler eviction policies such as CLOCK or FIFO, and leaves the heavy lifting to the admission policy [96]. Smart policies for data placement seek to reduce write amplification [14], and can be used in tandem with Baleen.

Prefetching policies CacheSack [97] incorporated static prefetching policies as choices for their optimization function. [107] implemented heuristic-based prefetching for photo stores, but found significant room for improvement relative to their offline optimal. Others have posed caching as a scheduling problem in the context of streaming video and incorporated aspects of prefetching [45, 65, 76]. In databases, Leaper trains a ML prefetcher to exploit reuse at the key range level [95].

Models for caching and offline optimal Bélády's MIN algorithm is the optimal eviction policy [4]. [68] introduces Relaxed Bélády for eviction which prunes the decision space like OPT does; however OPT makes stronger assumptions valid for flash admission and decides at a higher granularity (see § 4.3.1). Raven [29] is a probabilistic approximation of MIN. [16] sought to extend Bélády to admission with a container-optimized MIN that optimizes hit rate while minimizing flash erasures, but did not provide an online algorithm. Our proposed OPT policy is the only online policy that approximates the optimal flash admission policy, and which can easily optimize an arbitrary metric like DT, not just hit rate.

ML for eviction Some policies seek to learn from Bélády, such as LRB which learns a relaxed Bélády [68], and RL-Bélády [87]. A key challenge to using RL is the long delays for rewards. [6] Others seek to go beyond Bélády, such as LRU-BaSE [83]. MAT [88] reduces ML inference overhead by using a heuristic to filter out likely candidates. HALP [69] augments a heuristic with ML for the YouTube CDN. Deep learning has also been applied to learn forward reuse distance with LSTMs [40] and reinforcement learning [86]. [66] uses a support vector machine with features they derived from training an LSTM. [17] proposes that a classical caching policy be run in parallel with ML policies, allowing the implementation to switch to the better-performing policy dynamically. ML-based eviction is orthogonal to Baleen's contribution and cannot control flash write rates.

Metrics: byte miss rate, object miss rate and Disk-head Time We are not the first to recognize the need to balance object miss rate and byte miss rate in caching when object sizes vary [82]. Some have approached this through the lens of size-awareness [8, 20]. One policy, LRU-BaSE [82], tries to optimize both object miss ratio and byte miss ratio. Disk-head Time is known in the storage community [12, 47, 48, 78, 79] with a number of cluster systems and disk scheduling algorithms from the 2000s that optimized for it. To the best of our knowledge, Baleen is the first flash caching policy that optimizes for Disk-head Time.

Table 2.1: Summary of related work

System	Year	AP/ PF/E[1]	Metric	Peak[2]	Flash[3]	ML	Online	Size-aware	Real evaluation[4]	Application	Main contributions
Caches with admission policies											
Baleen [85]	2023	AP, PF	DT	✓	✓✓	GBM	✗	✓	TB (CL)	Bulk storage	ML imitates optimal approximation (based on episodes residency model)
CacheSack [96, 97]	2022	AP, PF	TCO (hits, flash writes)		✓✓	✗	✓	✓	Prod (Google)	Bulk storage	Greedy, computes best policy per category every 5 mins
CacheLib [5]	2020	AP	Hits		✓	GBM	✗		Prod (Meta)	Bulk storage, CDN, Key-value, Graph	Use recent history (in Bloom filters) as features
RL-Bélády [87]	2020	AP, E	Hits			FF (AP), GBM (E)	✓	✓	✗	CDN	FF trained using Monte Carlo, GBM predicts next request time, Autotune eviction threshold
Flashield [21]	2019	AP	Hits		✓	SVM	✗	✓	TB (mc)	Key-value	Use recent history (hits in DRAM) as features
AViC [1]	2019	AP, E	Hits, Bytes			GBM			✗	CDN	Predict future access time, rejects one-hit-wonders
TinyLFU [18, 20]	2017	AP	Hits			✗	✓	✓ [20]	TB (Caf)	Web services, Block I/O	Admits items above frequency threshold

continued on next page

[1] AP = Admission Policy, PF = Prefetching, E = Eviction Policy.

[2] ✓= Evaluates at peak.

[3] ✓= Evaluates flash writes, ✓✓= Explicitly co-optimizes for flash writes.

[4] TB = Testbed, Prod = Production, ✗= Simulator, CL = CacheLib, Caf = Caffeine, ATS = Apache Traffic Server, SC = SegCache, mc = memcached, V = Varnish.

System	Year	AP/ PF/E[1]	Metric	Peak[2]	Flash[3]	ML	On-line	Size-aware	Real evaluation[4]	Application	Main contributions
AdaptSize [8]	2017	AP	Hits			✗	✓	✓	TB (V)	CDN	Admits items below size threshold
Other flash caches											
LRB [68]	2020	E	Bytes		✓	GBM	✓		TB (ATS)	CDN	Randomly sample items; ML chooses one to evict
Pannier [38]	2017	E	Hits, latency		✓	✗	-		Sim+SSD	Block I/O	Middleware
RIPQ [72]	2015	E	Hits		✓	✗	-		Prod (Meta)	Blob storage	Segmented-LRU, GDSF
Policies for DRAM caches											
GL-Cache [91]	2023	E	Hits, Bytes			GBM	✓	✓	TB (SC)	CDN, Block I/O	Groups similar objects together to aid ML
HALP [69]	2023	E	Bytes	✓		MLP	✓		Prod (YouTube)	CDN	Heuristic shortlists candidates for ML to evict
MAT [88]	2023	E	Bytes			GBM	✓		TB (CL)	CDN, Key-value, Block I/O, Object Store	Heuristic shortlists candidates for ML, which predicts TTL. Focuses on reducing ML overhead.
LRU-BaSE [82]	2022	E	Hits, Bytes			DQN	✓	✓	Prod (Tencent)	CDN	RL with CDN- and LRU-specific improvements
Zhou *et al* [106]	2021	E	Hits			✓	✗		✗	Bulk storage	Mine text tags for features

continued on next page

[1] AP = Admission Policy, PF = Prefetching, E = Eviction Policy.
[2] ✓= Evaluates at peak.
[3] ✓= Evaluates flash writes, ✓✓= Explicitly co-optimizes for flash writes.
[4] TB = Testbed, Prod = Production, ✗= Simulator, CL = CacheLib, Caf = Caffeine, ATS = Apache Traffic Server, SC = SegCache, mc = memcached, V = Varnish.

System	Year	AP/ PF/E[1]	Metric	Peak[2]	Flash[3]	ML	On-line	Size-aware	Real evaluation[4]	Application	Main contributions
CACHEUS [63]	2021	E	Hits			RL	✓		✗	Block I/O	Improves LeCaR for scan and churn workloads
Parrot [44]	2020	E	Hits			Transformers	✗		✗	CPU	Imitates approximated Belady
LeCaR [77]	2018	E	Hits			RL	✓		✗	Block I/O	Regret minimization
LHD [3]	2018	E	Hits			✗	✓	✓	TB (mc)	Key-value, Block I/O	Optimize for hit density

[1] AP = Admission Policy, PF = Prefetching, E = Eviction Policy.

[2] ✓= Evaluates at peak.

[3] ✓= Evaluates flash writes, ✓✓= Explicitly co-optimizes for flash writes.

[4] TB = Testbed, Prod = Production, ✗= Simulator, CL = CacheLib, Caf = Caffeine, ATS = Apache Traffic Server, SC = SegCache, mc = memcached, V = Varnish.

Table 2.2: **Caching simulators and production systems**

Name (Author)	Target	Status	Lang	Papers	Eviction	Admission
Simulators						
BCacheSim (Daniel Wong)	Bulk storage	Active Dev	Python	[85]	LRU, FIFO, LIRS, TTL	Baleen, RejectX, CoinFlip, OPT
libCacheSim (Juncheng Yang)	Key-value, CDN	Active Dev	C++	[89, 91]	*See below*	TinyLFU
Eviction policies: FIFO, LRU, Clock, LFU, LFU with dynamic aging, ARC, SLRU, GDSF, LeCaR, Cacheus, Hyperbolic, LHD, LRB, GLCache, Belady, BeladySize						
webcachesim2 (Zhenyu Song)	CDN	Maint mode	C++	[68]	*See below*	AdaptSize, Adaptive-TinyLFU
Eviction policies: LRB, LR, Belady, Relaxed Belady, Inf, LRU, B-LRU, ThLRU, LRUK, LFUDA, S4LRU, ThS4LRU, FIFO, Hyperbolic, GDSF, GDWheel, LeCaR, UCB, LHD, Random						
Production systems						
CacheLib (Meta)	Key-value, bulk storage, CDN	Active Dev	C++	[5, 53, 85]	LRU, Segmented LRU, LRU-2Q, TTL, FIFO	TinyLFU, RejectX, Random
Caffeine (Benjamin Manes)	In-memory	Maint mode	Java	[18, 19, 20]		
Pelikan/SegCache (Twitter)	In-memory key-value	Stable	Rust	[91]		

Table 2.3: Cache workloads used in literature. We include all bulk storage system traces in addition to selected CDN, block I/O and key-value traces that have been used in multiple papers.

Source	Year	Meta-data features	Length	Avg Req Size	Avg Obj Size	#Num	Total trace size	Public	Years in use	#Papers	Used by
Bulk storage											
Meta (Tectonic)	2024	✓	30-day					Not yet	2024		**Thesis (drift)**
Meta (Tectonic)	2021, ‘23	✓	7-day	3 MB	6.3 MB	4	170 GB	✓	2023	1	**Thesis (Baleen)** [85]
Google (Colossus)	2022	✓	2-day					✗	2023	1	CacheSack [96]
Meta (Tectonic)	2019	✓	7-day	3 MB	5.8 MB	3	11 GB	✓	2020–23	2	CacheLib [5], **Thesis (Baleen)** [85]
Block I/O[1]											
Alibaba	2020		31-day				751 GB	✓			SepBIT [83], [39]
TencentCloud	2020		9-day					✓			OSCA [101]
Fujitsu (SYSTOR)	2016		28-day	9 KB	31 KB		52 GB	✓	2016–23	Many	[37]
CloudPhysics (VM)	2015		7-day					✓	2015–23	2+	GL-Cache [91], Cacheus [63]
MSR Cambridge	2007	✗	7-day		40 KB		5 GB	✓	2009–23	Many	Pannier [38], LHD [3], CloudPhysics, GL-Cache [91], MAT [88]
FIU	2008		3-mth	8 KB			29 GB	✓	2018	1+	LeCaR [77]
CDN											
Meta	2023		7-day			3	40 GB	✓			
Google (YouTube)	2021		3-day					✓[2]	2023	1	HALP [69]
Tencent (QQPhoto)	2016		9-day					✓	2018–22	1+	LRU-BaSE [82]

continued on next page

[1] Block I/O is defined by SNIA as including "block level (e.g., at the logical volume manager, disk driver, etc. level) and block protocol (e.g., SCSI, ATA, Fibre Channel) traces." We distinguish this from bulk storage (distributed exascale cluster storage systems used in the cloud that aggregate storage needs of many systems).

[2] Google has offered to release two traces (a developed market region and an emerging market region) upon signing of a data sharing agreement.

[3] MemCachier is a commercial memcached service.

Source	Year	Meta-data features	Length	Avg Req Size	Avg Obj Size	#Num	Total trace size	Public	Years in use	#Papers	Used by
Wikipedia	2018		14-day					✓	2020–23	3+	LRB [68], GL-Cache [91], MAT [88]
Key-value (e.g., Memcached, Redis)											
Meta	2022		5-day			1	24 GB	✓			
Twitter	2020		7-day			54	14 TB	✓	2020–21	2+	[89], Segcache [90]
IBM Cloud Object Storage	2019	✗	7-day	0.2 MB	1 MB	98	88 GB	✓	2020–23	3+	[23], MAT [88]
MemCachier[3]	≤2017		7-day					✗	2017–23	4+	LHD [3], Flashield [21], Hyperbolic [11], MAT [88]

[1] Block I/O is defined by SNIA as including "block level (e.g., at the logical volume manager, disk driver, etc. level) and block protocol (e.g., SCSI, ATA, Fibre Channel) traces." We distinguish this from bulk storage (distributed exascale cluster storage systems used in the cloud that aggregate storage needs of many systems).

[2] Google has offered to release two traces (a developed market region and an emerging market region) upon signing of a data sharing agreement.

[3] MemCachier is a commercial memcached service.

Chapter 3

Experimental setup

In this chapter, we introduce the bulk storage workloads used throughout the rest of this dissertation. We also describe the hybrid cache simulator we designed around the episodes model, and detail our efforts to ensure the fidelity of its results, including an academic testbed we set up to run our policies in CacheLib.

3.1 Datasets: real traces from production caches for bulk storage systems

A factor that differentiated our work was our extensive collection of caching traces from high-performance production caches for bulk storage systems. We are grateful to our collaborators at Meta who helped us collect traces over the years from 2019 to 2024.

Note that the workload drift section employs additional traces beyond the ones presented here. They are described in §8.2.

Comparable traces The closest traces we know of are the Google CacheSack [97] traces, which are not publicly available. The Google Thesios [60] traces were promising and we evaluated them, but we were unable to use them directly as they were sampled post-cache. Many prior caching papers, even as recent as 2023, were still using the MSR traces from 2007, as can be seen in Table 2.6.

3.1.1 Trace collection and preprocessing

Traffic to the bulk storage system was sampled on the storage nodes themselves, which was where the flash also was. Traffic was sampled before it hit the flash cache.

Trace collection in 2019 The process through which the 2019 traces were collected is unknown, but we strongly suspect that each trace was collected by sampling the traffic at a single node.

Trace collection from 2021 onwards Starting from 2021, the trace collection process was standardized to take a sample from every single storage node in the cluster at a fixed sampling rate (e.g., $\frac{1}{4000}$). These thousands of machine-specific samples were then aggregated and buffered for 30 days in a separate system. which could be dumped to collect a trace for the last 30 days.

The sampling rate and number of nodes are recorded at the time the trace was dumped, and we use this to determine the fraction of a machine's traffic that the received trace represents: $\frac{NumberOfNodes}{SamplingRate}$, so that we can scale the cache size correctly.

Trace anonymization Before we receive the traces, they would be anonymized by replacing string values with integers in a process similar to one-hot encoding. From 2024, the trace anonymization process was augmented to ensure that keys were consistently anonymized across dumps, enabling traces dumped at different timepoints to be stitched together to form a longer trace.

Preprocessing To speed up evaluations and standardize our evaluation process, we further downsample the trace from Meta by sampling it on the block key space. **Learning point:** We weigh each block by the number of accesses, and found that this reduced variance when using different samples of the trace and also allowed us to use smaller samples (as low as 0.05%) of the trace while still getting meaningful results.

Time of collection The Region1 and Region2 traces were recorded from different clusters over the same 7 days in Oct 2019, while the Region3 trace was recorded from another cluster over 3 days in Sep 2019. Region4 was recorded over 7 days in Oct 2021, and the remaining traces (Region5, Region6, Region7) were collected in Mar 2023.

3.1.2 Workload characteristics

Description of clusters and their workloads Early clusters (also known in this text as a Region) each supported only one tenant (such as data warhouse or blob store), but eventually clusters were made multi-tenant. Clusters had thousands of nodes, although this number would flunctuate over time, including during the course of the trace, as individual machines failed or were rotated out.

This is a description of the workloads for each cluster:

1. Regions 1-3 (2019): each a data warehouse
2. Region4 (2021): data warehouse
3. Region5 (2023): 10 “tenants”, largest being data warehouse and blob store
4. Region6 (2023): 10 “tenants”, largest being data warehouse and blob store
5. Region7 (2023): 10 “tenants”, largest being data warehouse and blob store
6. Regions 4-7 are from different geographical regions.

Each tenant supports 100s of applications. Data warehouse is storage for data analytics (e.g., Presto, Spark, AI training), with larger reads than blob storage. Blobs are immutable and opaque, and include media (photos, videos) and internal application data (e.g., core dumps). See the Tectonic[58] paper for further details.

Trace statistics We make a few observations from the data in Table 3.1:

1. Admit-All Write Rate, the flash write rate required to admit everything, varies but exceeds 300 MB/s for most of the traces, a rate that would wear out an SSD within a few months instead of 5 years, underlining the importance of flash admission policies.
2. The average block size is 5–6 MB and the average access size is 2–3 MB, much larger than the 10s of KBs in the Block I/O storage traces shown in Table 2.6.
3. One-hit wonder rate is high, meaning that the cumulative penalty for flash admission false positives is significant.
4. The high percentage of PUT-only blocks with no subsequent GETs makes predicting whether to prefetch on PUT very, very challenging, since there are minimal features and the odds are stacked against having any reuse.

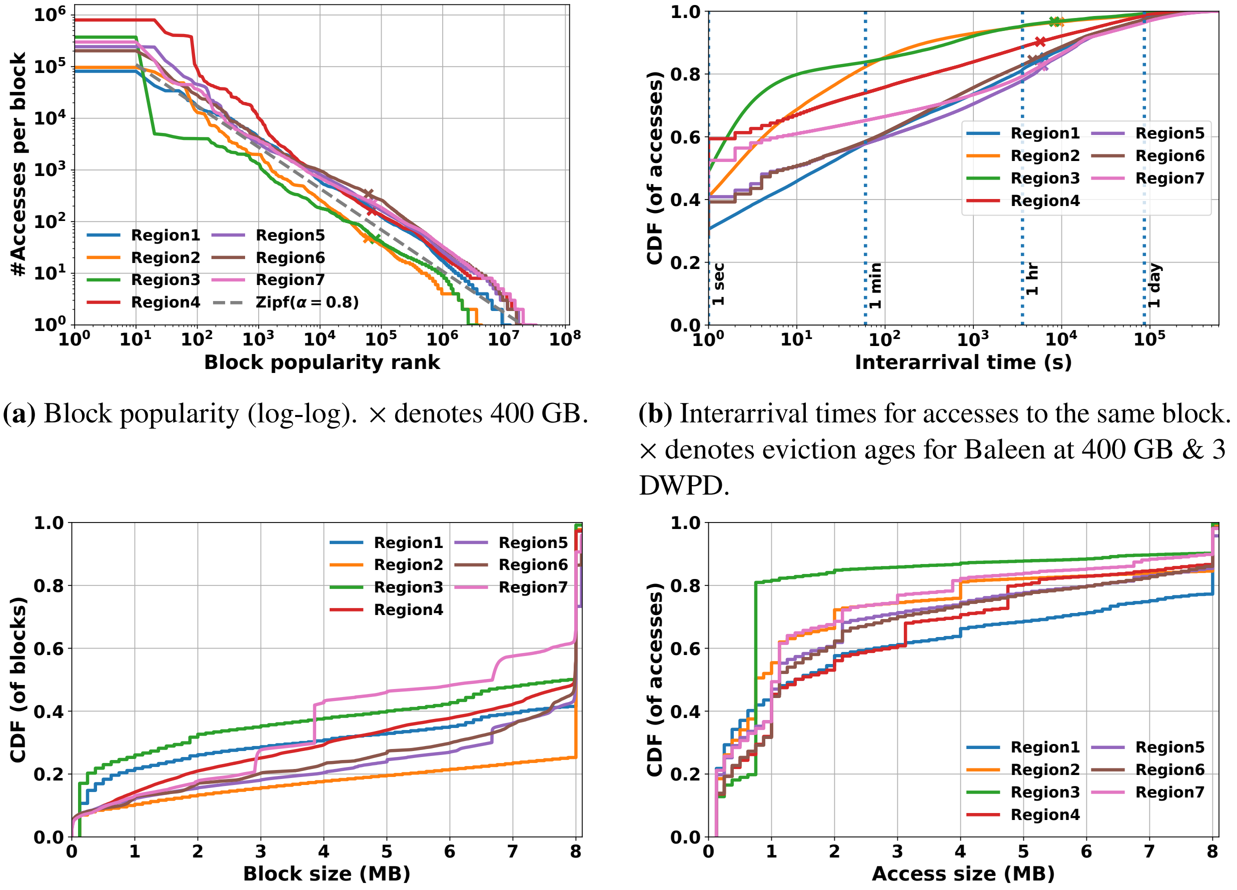
Table 3.1: Full statistics of traces.

Dataset	**Year**	**Request Rate** (s^{-1})	**Avg Block Size (MB)**	**Access size (MB)**	**Compulsory miss rate**[1]	**One-hit-wonder rate**[2]	**PUT-Only Blocks**	**#PUT / #Acc**	**Admit-All Write Rate**
Region1	2019	244	5.70	3.41	18%	54%	46%	13%	316 MB/s
Region2	2019	106	5.07	2.85	39%	83%	81%	14%	121 MB/s
Region3	2019	139	6.71	2.42	19%	48%	46%	16%	45 MB/s
Region4	2021	406	5.87	2.87	14%	53%	40%	10%	280 MB/s
Region5	2023	364	6.84	2.62	18%	59%	33%	9%	480 MB/s
Region6	2023	404	6.77	2.74	14%	55%	38%	10%	478 MB/s
Region7	2023	426	5.71	2.23	17%	62%	38%	12%	492 MB/s

[1] Compulsory miss rate refers to the ratio of blocks to accesses;

[2] One-hit-wonder rate is the fraction of blocks with no reuse.

The popularity distribution of blocks (Fig 3.1a) fit a Zipf($\alpha = 0.8$) distribution, where the i-th most popular block has a relative frequency of $1/i^{\alpha}$. Fig 3.1b shows the interarrival time distribution, with the converged eviction age for Baleen marked with crosses. For all traces, less than 20% of interarrival times exceed the converged eviction age. Fig 3.1c and Fig 3.1d shows the size distribution for blocks and accesses respectively. The majority of blocks are the maximum size (8 MB) with averages of 5.1-6.8 MB across traces, but most accesses are only a fraction of the block with the median access less than 2 MB.



(a) Block popularity (log-log). × denotes 400 GB.

(b) Interarrival times for accesses to the same block. × denotes eviction ages for Baleen at 400 GB & 3 DWPD.

(c) Block size distributions.

(d) Access size distributions.

Figure 3.1: Distributions of block popularity, access interarrival times, block sizes, and access sizes. In a, lower values of α indicate it is harder to cache.

3.2 BCacheSim: our online hybrid cache simulator

We developed a Python simulator to accurately estimate CacheLib performance without doing the actual heavy lifting. This is an approach taken by other ML for Systems projects [73]. This lightweight simulator is easier to include in a ML training pipeline, and takes as input a Tectonic trace and measures many end-to-end metrics (e.g., average eviction age, Peak DT) that cannot be obtained from offline episode analysis. Having the training setup be Python-centric aids in faster prototyping, ease of use by data scientists, and ease of integration with existing ML training pipelines.

Modeling of HHD and SSD differences Baleen accounts for differences in hardware (HDDs, SSDs) via the target flash write rate and constants in the TCO & disk-head time formulas. For flash, the pertinent characteristics are those affecting endurance (and thus write rate). Fig 5.6 and 5.7 show Baleen performing at different flash write rates and cache sizes. For HDDs, our simulations assume a constant average seek time and bandwidth in the DT formula (Eq 4.2). These parameters vary minimally across disks, as illustrated in Fig 4.1 (simple formula closely matches actual disk utilization in production). Baleen includes a small benchmark to measure these constants for a given disk.

Simulation time 624 machine-days were used for the final runs to generate the results used in the paper. Each simulation of a ML policy takes at least 30 minutes, multiplied by 7 traces and 10 samples each.

3.3 Our academic CacheLib testbed

In the absence of production access, we took the additional step of creating an academic testbed environment that resembles the production environment more closely. Like BCacheSim, our Python simulator, it replays bulk storage workloads that were recorded from production systems. Compared to BCacheSim, however, it has the following advantages: 1) it performs actual cache insertions, evictions and disk accesses on real hardware, and 2) it does so using the same CacheLib library used in production (along with its accompanying characteristics and quirks), increasing the likelihood our reported gains will be more easily realized in production.

Hardware The Tectonic production setup used to record traces and counter values has a 400 GB flash cache, 10 GB DRAM cache and 36 HDDs. Our academic testbed uses enterprise-grade hardware, but with less HDDs per node and thus a proportionally smaller cache size. It is a 24-node cluster, where each node has a 16-core Intel Xeon E5-2698 CPU, 64 GB of DRAM, Intel P3600 400 GB NVMe SSD, Seagate ST4000NM 4 TB HDDs, and runs Ubuntu 18.04. The SSDs and HDDs used are enterprise-grade. The size of the cluster does not affect the veracity of the testbed as each individual experiment run only involves one node; multiple nodes are used to speed up the completion of the experiments, given the large number of runs necessary (number of policy configurations times 7 traces times 10 samples from each trace).

Modifications to CacheLib We implemented support for ML admission and prefetching policies. Our prefetcher is implemented as a header-only library that other CacheLib applications may include. We mock calls to Tectonic so that every miss issues a real IO of the right size against HDDs, and measure the wall-clock time as Disk-head Time consumed. Static features are stored in the CacheLib payload, while history counts are tracked by CacheLib. We added functionality to CacheBench (CacheLib's benchmark suite) to replay Tectonic traces. Our mock application follows Tectonic's behavior of breaking down each IO into one or more CacheLib acccesses. CacheLib employs a region-based LRU, with different regions for different sizes. Since segments are uniformly 128 KB, we set region size to 142 KB to contain one segment each plus overhead.

Prefetching implementation in CacheLib applications Every request to the bulk storage system references a block in the backing store and a byte range within that 8 MB block. Each request is translated by the application into (potentially multiple) CacheLib segment-level requests. CacheLib is not aware that segments may belong to the same block.

Thus, prefetching must be implemented by the application issuing requests against CacheLib, which is the bulk storage system in our case (Fig 5.2). For our studies, we implemented it in CacheBench. On each client request, Baleen's prefetcher will be triggered after the application has queried CacheLib and found out whether segments are hits or misses. Thus, the prefetcher has access to the client request metadata and knows how many requested segments were present in cache. On a miss, the application makes a request to the backing store, giving the prefetcher a chance to fetch extra segments and insert those into cache.

3.4 Validation of BCacheSim simulator and CacheLib testbed

In this section, we validate both the CacheLib testbed and our Python simulator (BCacheSim) against the Meta production environment.

Fig 3.2 shows that testbed and simulator are faithful to production counters for disk utilization. We compare production counters for one day (collected on a per-minute basis and aggregated to 10-min intervals) to simulator and testbed results for a trace collected on the same day. One observes that the lines for the testbed and the production counters are remarkably similar even though the trace used for the testbed line is only 1% of 1/10000 (10^{-6}) of the full traffic pattern that is responsible for the Production Counters line (in green).

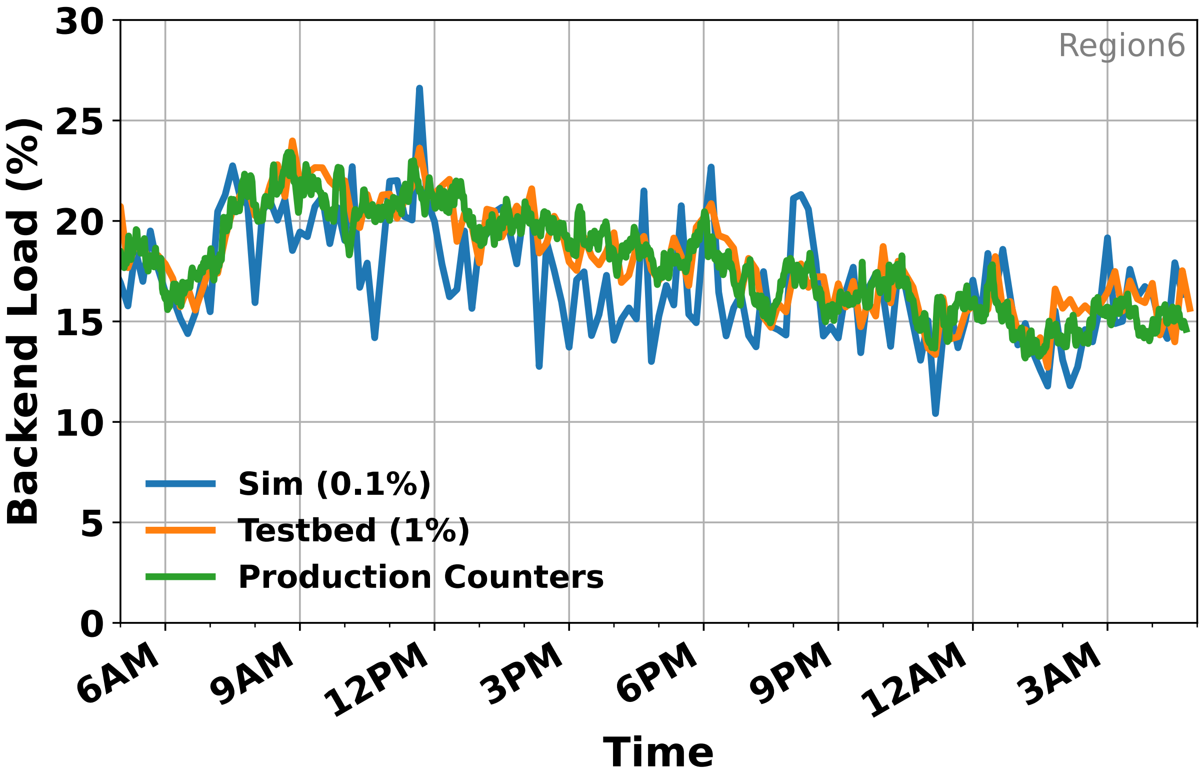


Figure 3.2: Sim-Testbed-Production comparison, RejectX, 1 day

3.5 Miscellaneous experimental details

Sample rates used Testbed results (used to validate our simulator at a fixed flash write rate) used 1-5% samples (maximum sample rate is 5%, limited by the ratio of HDDs (2:36)). For each trace, we used the smallest trace percentage that gave us consistent results with higher trace percentages. For the newer traces, the lower value was used, while the higher value was required for the older traces collected in 2019 which were smaller with less requests.

Simulator results used sampling rates from 0.1% to 5%. A higher sample percentage was used for smaller workloads. We scale to a 400 GB-equivalent flash cache and our target flash write rate.

ML training setup I wrote a Python module that generates episodes and trains the ML models. This plugs into BCacheSim. The episodes module takes in a trace and returns the ML models. I then run simulation loops to converge on an assumed eviction age and admission policy threshold. LightGBM [32] was used for training and inference, with 500 rounds of boosting and 63 leaves.

A train-test split is performed on the time dimension, i.e., the first day of each workload is used as training data, with the remaining days used for testing.

Metrics The savings from using Baleen are dominated by the degree by which it reduces the number of HDDs required to handle peak load. Therefore, our evaluation focuses on Peak DT (see § 4.3). To aid comparison across traces, we normalize each policy's Peak DT by the Peak DT required with no cache.

Admission policy baselines We compared Baleen to 4 baselines: RejectX, CoinFlip, and two state-of-the-art ML baselines, Flashield [21] and CacheLib [5]). We focus on RejectX as it is publicly available and has been chosen over state-of-the-art ML models in industry. The CacheLib ML policy addresses Flashield's limitations (see §5.3.1) and uses non-episode-related features.

Chapter 4

Episodes & OPT: modeling flash caching and exploring savings in Disk-head Time

This chapter describes elements of the principled approach we developed for ML in caching. We wanted to be able to try and prototype different ideas quickly, and get an upper bound on their possible benefits. Also important was to establish how far the gap to optimal was for different scenarios and workloads in order to quantify the possible benefits of improving caching algorithms.

We identified Peak Disk-head Time of the backend hard disks as the key metric that should be optimized in bulk storage systems and their caches, given a fixed target flash write rate. We devised the episodes model to facilitate reasoning about flash caching, making it easier to describe and think about caching policies. In a nutshell, using episodes decouples formerly dependent decisions by summarizing the state of the cache in a single statistic: the assumed eviction age. Our OPT policy (which approximates an optimal online admission policy) followed naturally from the episodes model.

Using our analytical model allowed us to approximate cache performance, get upper bounds, and see cache behavior at the extremes, but greater fidelity to production systems was necessary. Thus, we developed BCacheSim, our online flash caching simulator, and validated those results against a CacheLib academic testbed and CacheLib in production, as discussed earlier (§3).

4.1 Measure Disk-head Time, not hits or bandwidth

We quantify backing store load via *disk-head time* (DT), which is a metric that balances IOPS and bandwidth.

Definition **Disk-head Time** (DT) is the cost of serving requests to the backend. For a single IO that fetches n bytes, with t_{seek} the time for one disk seek and t_{read} the time to read one additional byte:

$$DT_i = t_{seek} + n \cdot t_{read} \tag{4.1}$$

Definition **Backend load (Utilization)** of a time window is the total DT needed to serve misses, normalized by provisioned DT (1 disk-sec per disk per sec): $Util_{DT} = \frac{\sum_i DT_i}{DT_{Provisioned}}$, where

$$\sum_i DT_i = Fetches_{IOs} \cdot t_{seek} + Fetches_{Bytes} \cdot t_{read} \tag{4.2}$$

DT accurately models throughput constraints of bulk storage systems. DT models both the IOPS and bandwidth limitations of the backing HDDs. (This concept can be extended to other systems with IO setup and transfer costs, such as CDNs.) In our caching setup, we fetch the smallest range covering all cache misses, and normalize DT by HDDs per node to get backend load.

In Fig 4.1, we validate DT that can be calculated using only two production counters, IO misses and bytes fetched, against system-reported disk utilization on a Meta production cluster in Feb 2023. The peaks line up within 1%, which was surprisingly accurate given the simplicity of this formula (t_{seek} and t_{read} are constants) and the vagaries of production systems (included in the system disk utilization measurements). Industry experts were initially surprised by this finding given that this simple formula does not account for the variation in seek time between different locations on the disk platter (and related optimizations like skewing and disk head scheduling), queuing delays, file system fragmentation, and numerous other factors. On our academic testbed, we recorded and measured the variability of Disk-head Time and found that while the time consumed to perform individual disk I/Os did vary from the formula-predicted time (on average, 11%), this was not sufficient to perturb the trends sufficiently in production as shown in Fig 4.1.

DT correctly balances IO misses and byte misses. In practice, $Fetches_{Bytes} \approx Misses_{Bytes}$ (there is a very small difference due to non-consecutive misses). Hence, $\sum DT$ can be interpreted as a weighted sum of IO misses and byte misses, and reducing DT consumed reduces the familiar caching metrics of IO miss rate and byte miss rate.

Conversely, optimizing only the IO miss rate or byte miss rate may result in mistakes made. For example, IO hit rate cannot distinguish these two scenarios though one is better than the other. Consider two blocks, both with 64 accesses. For the first block, each of the 64 segments is requested, one at a time. For the second block, every access requests all 64 segments. While both require the same cache space and save the same IOs, caching the second block saves more DT.

Definition **Peak DT** is the P100 backend utilization ($Util_{DT}$), measured every 10 minutes. The peak refers to the 10-min interval with the highest DT:

$$PeakDT = Util_{DT}^{P100} \tag{4.3}$$

Peak DT is proportional to the number of backend servers required. System capacity, such as the number of backend servers, is provisioned to handle peak load in systems that need to meet real-time demand. Therefore, to reduce the backend size required, Peak DT should be minimized. This introduces the need for scheduling (i.e., when to spend the flash write rate budget) to prioritize the admission of items that contribute to the Peak DT. As explicitly optimizing admission for the

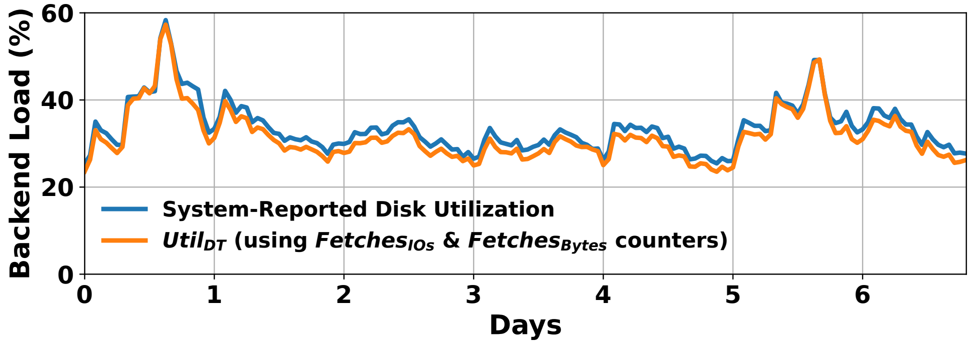


Figure 4.1: DT validated in production. Our DT formula (plugging counters into Eq 4.2) matches measured disk utilization (blue) closely, with the peaks lining up within 1%. This was surprisingly close. The peak of 58% occurs on Day 0.

peak introduces significant complexity, we leave that for future work. For this paper, we design our admission and prefetching policies to minimize average DT (and show that they are successful in reducing Peak DT), and optimize for Peak DT in other aspects of the system. To improve admission, we must first know what "better" looks like. We use *Disk-head Time* as an end-to-end throughput metric to evaluate this. This section describes our decomposition of the flash caching problem, and our attempt at approximating an optimal admission policy (OPT) and a framework (episodes) to evaluate the cost-benefit trade-offs of not just admission policies, but orthogonal improvements such as prefetching.

4.2 Episodes: an offline model for flash caching

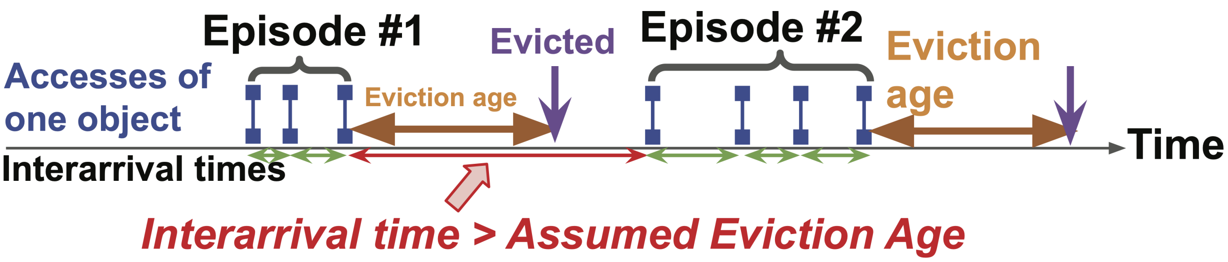


Figure 4.2: An episode is a group of accesses during a block's residency. Accesses (in blue) are grouped into two episodes as the interarrival time (in red) exceeds the assumed eviction age.

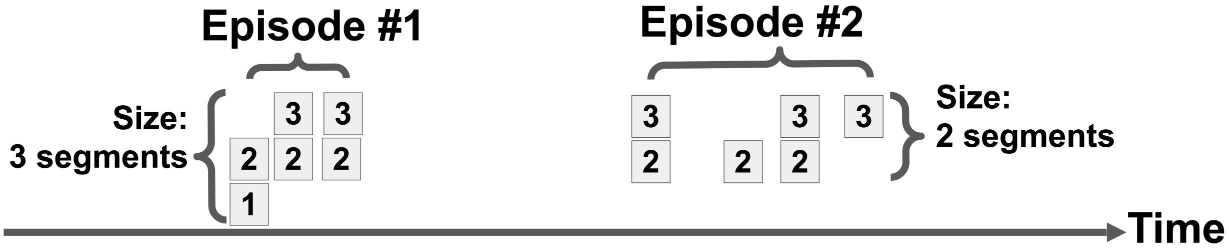


Figure 4.3: Episodes span space (measured in segments) in addition to time. An episode's size is the smallest number of segments required to be admitted to get all possible hits within an episode. OPT-Range (§ 4.4) is (1,3) and (2,3) respectively.

We devised an offline model for flash caching for efficient evaluation of flash caching improvements, and to facilitate the training of ML-based policies. This model revolves around *episodes*, which are defined as:

Definition An **episode** is a sequence of accesses that would be hits (apart from the first access) if the corresponding item was admitted. It is defined on a block (the rationale being that a cache hit only occurs if all segments are present in cache).

An episode may span multiple segments, and as shown in Fig 4.3, an episode's size is the number of segments needed to cache it. This leads naturally to a formulation for prefetching. (An important distinction between episodes and block-level LRU analysis is that different episodes for the same block can have different sizes.) An episode's timespan is the length of time between the first access of any segment and the last eviction of a segment.

We generate episodes to aid ML training by exploiting the model of an LRU cache as evicting items at a constant logical time (*eviction age*) after the last access [7, 15, 24, 52]. In an LRU cache, the eviction age is the logical time between an item's last access & eviction. As shown in Fig 4.2, we group accesses into episodes such that all inter-arrival times within episodes are no larger than the assumed eviction age.

Episodes provide a direct mapping to the costs and benefits associated with an admission, and which corresponds directly to the decisions being made by admission policies. These benefits and costs are associated with an item's entire lifespan in cache, and are not obvious from looking at a stream of individual accesses. Moreover, with flash caching, it is optimal to admit as early as possible in the episode, given that the flash writes required are a fixed cost. By shifting the mental model from interdependent accesses to independent episodes, we can reason about decisions more easily.

Decisions on episodes can be made independently by assuming a constant eviction age. This also allows decisions to be made in parallel. The added pressure on cache space via an admission is accounted for via downward pressure on the eviction age. We determine an appropriate eviction age using simulations that measure the average eviction age. In reality, the eviction age is not constant and varies with cache usage over time. One approach deals with this by calculating policies for a wide range of possible eviction ages [96]. However, we find that in terms of end-to-end performance, Baleen is not sensitive to the assumed eviction age (typically 2+ hours) as long as it is not extremely low (e.g., seconds to minutes).

The episode model also allows for an efficient offline analytical analysis of policies via Little's Law. Given the arrival rate and assumed eviction age, we can estimate the cache size required, and set the eviction age such that the analytical cache size is equal to the cache size constraint. While this is much more efficient than an online simulation and is useful to explore a greater range of parameters than is possible with online simulation, the numbers will differ from simulated ones as the cache size constraint is not enforced all the time, only as a long-term average.

Admission policies can be viewed as partitioning these episodes into those admitted and discarded. This can be done via scoring episodes and ranking them by score, and we elaborate on this in the next section.

4.3 OPT approximates optimal online admission policy

Using episodes, we can devise an admission policy (AP) for online simulation that approximates the optimal AP using offline information from the entire trace.

1. Each block's accesses are grouped into episodes using an assumed eviction age.
2. All episodes are scored and sorted.
3. The maximum no. of episodes are admitted such that the total flash writes required do not exceed the write rate budget.

During online simulation, accesses will be admitted if they belong to episodes marked as admitted during the offline process. OPT scores each episode to maximize on the DT saved if admitted and to minimize its size (flash writes required to admit):

$$Score(Episode) = \frac{DTSaved(Episode)}{Size(Episode)} \tag{4.4}$$

In the development of OPT, we also evaluated other scoring functions.

$$\begin{align} Score(Episode) &= Hits(Episode) \tag{4.5} \\ Score(Episode) &= \frac{Hits(Episode)}{Size(Episode)} \tag{4.6} \\ Score(Episode) &= \frac{Hits(Episode)}{Size(Episode) \cdot Timespan(Episode)} \tag{4.7} \\ Score(Episode) &= \frac{DTSaved(Episode)}{Size(Episode) \cdot Timespan(Episode)} \tag{4.8} \\ & \tag{4.9} \end{align}$$

Eq 4.5 is straightforward and intuitive. Adding size-awareness to caching algorithms, as we did by incorporating the episode size in Eq 4.6, was an important improvement which has also been observed by others [8]. Hit density, as used in LHD [3], is in theory the best objective function for a standard cache eviction policy, but we found that (as shown in Eq 4.7, and its DT variant, Eq 4.8) to be suboptimal in our flash caching case. This lends credence to our belief that for flash caching, the flash writes incurred are typically the limiting factor rather than the time spent in cache, and thus it makes to prioritize flash writes and also to make admission decisions as early as possible, since the cost is paid at time of admission.

4.3.1 Comparison to LRB's Relaxed Belady

At first glance, the use of the assumed eviction age to generate episodes may resemble LRB's Belady boundary, but we explain how episodes are different and the advantages of using episodes for flash caching.

For context, LRB [68] introduces Relaxed Bélády for eviction, which only considers objects for eviction beyond a time it calls the Belady boundary. Like our OPT's use of the assumed eviction age, it prunes the decision space making it more efficient; our OPT is able to make stronger assumptions (due to the flash admission context), and train ML at a higher granularity of disjoint episodes, whereas LRB still operates at the finer granularity of accesses and is choosing which object is *more likely* to be good (has higher Good Decision Ratio) whereas OPT can determine which object is better to admit).

In addition, we evaluated weighing some terms more, e.g., $\frac{Hits(Episode)}{Size(Episode)^2}$, but found that it only decreased performance.

4.4 Extending OPT for prefetching

Episodes are also used to design our prefetchers and generate OPT labels for prefetching. By default, on a miss, the smallest IO that covers all missed segments is made, i.e., no prefetching occurs. It is possible to extend this IO and preemptively admit more segments. If done correctly, this reduces the total no of IOs needed and thus reduces DT.

Prefetching the correct segments is important to achieve a reduction in DT given a write bound. With imperfect admission policies, predicting a confidence value is necessary to balance the risk of real prefetching costs against possible benefits. Otherwise, prefetched segments compete with segments admitted from misses and drive up write rate while not reducing DT, meaning an overall reduction in DT for the same write bound. Note that the costs and benefits of prefetching must be evaluated against the opportunity cost of using writes for admission of missed blocks instead.

Deciding *when* to prefetch Fetching insufficient segments results in minimal or no DT reduction. On the other hand, fetching excess segments results in a high write rate. To balance these trade-offs, we need to know our confidence in our range prediction.

For instance, prefetching the entire block *on every miss* will result in an overall IOPS reduction given write rate constraints. A blunt method to increase precision is to prefetch *on every 2nd miss* or *on every partial IOPS hit* (when some but not all segments in an access return a hit). This indicates that part of the block was admitted to cache. For OPT prefetching, we prefetch on *OPT-Ep-Start*, the start of the episode as determined by the episode model.

Deciding *what* to prefetch: Whole-Block, OPT-Range The straightforward choice is to prefetch the entire 8 MB block (*Whole-Block*). However, the resultant write rate is too high, making it infeasible unless combined with prefetching on every partial IOPS hit. To evaluate how well we could perform given offline information from the whole trace, we introduce **OPT-Range**, which uses the generated episodes to determine an optimal range of segments to prefetch. OPT-Range is the minimal range of segments that covers all accesses in an episode. For the episodes in Fig 4.3, OPT-Range is (1,3) for Ep 1 and (2,3) for Ep 2. Whole-Block always fetches (1,64).

4.5 Efficiently exploring the space of possible improvements

Little's Law [27, 43] states the expected number of jobs in the system ($E[N]$) is equal to the product of the arrival rate (λ) and the mean time that jobs spent in the system ($E[T]$).

$$E[N] = \lambda \cdot E[T] \tag{4.10}$$

We apply this to flash caching, as follows:

- λ, the arrival rate is the flash write rate
- N, the number of jobs in the system, is the number of cacheable items, i.e., the flash cache size divided by the size of each item
- T, the time that jobs spend in the system, is the sum of the eviction age (the time between an item's last hit and eviction) and the item's useful time or timespan (the time between an item's admission/first access and the last access before being evicted)

This means that we have 3 constraints: the target flash write rate (which fixes λ), the cache size (which determines N) and the assumed eviction age (which determines T). In truth, there are only two free parameters here as setting two will lead to only one valid value for the last parameter.

When we run the analytical model, we fix the flash write rate and cache size and converge on the eviction age through a loop that terminates when the expected cache size is equal (or very close) to the target cache size.

We can think of cache policies as a pareto frontier on a Disk-head Time saved rate against cache size (or flash write rate) graph. Better policies move the frontier outwards. On these graphs, we can plot Max-WriteRate (if everything is admitted) or Max-CacheSize (if eviction age is infinite) respectively, which establishes an upper bound on how much savings could be gained from the optimal policy.

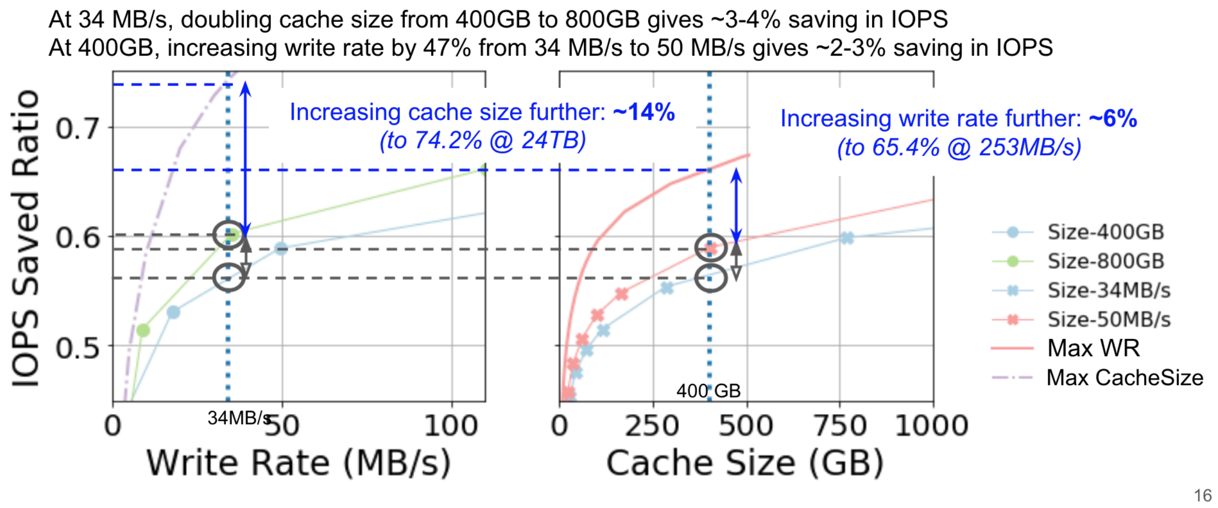


Figure 4.4: Analytical bound models (an early iteration).

This analytical model also allows us to easily determine, say, the flash write rate needed to admit all items, or the maximum cache size for a certain flash write rate, flash in a far faster and elegant fashion than running multiple simulations in a loop.

4.6 From analytical model to simulation

Converging on Eviction Age, Policy Threshold Parameters We repeatedly run the offline episode model and online simulation in a loop to converge on values for assumed eviction age (EA) and admission policy threshold. Recall that episodes are generated with an assumed EA. These episodes are used to train models, which are used in an online simulation where the average EA can be measured. We initialize assumed EA to an arbitrary value of 2 hours and repeat episode generation, model training, and online simulation until the assumed EA converges on the average EA from an online simulation. Within each loop iteration, there is another nested loop to find the correct admission policy threshold that results in the simulation achieving the target flash write rate. This inner loop aims to offset the small differences between offline analysis and a higher-fidelity online simulation.

4.7 Summary

In this chapter, we set up the metric (Peak Disk-head Time) that we will use to measure policies, in addition to introducing the episodes model which we use in the design of Baleen. In addition, we established the veracity of our metric and other design assumptions (in particular, the Disk-head Time formula and episodes using an assumed eviction age) through comparisons with production counters.

In the next chapter, we describe how Baleen builds on these foundations to train ML admission and ML prefetching policies.

Chapter 5

Baleen: Training ML policies for flash caching

We describe how Baleen provides episode-based solutions to two problems: how to train an **ML-based admission policy**, and using **prefetching** to improve beyond admission policies.

5.1 ML for flash admission

All caches, including flash caches, need to weigh the value of an item being inserted against the value of evicted and future items. However, flash caches are at a disadvantage relative to RAM cache, because:

- DRAM caches do not need admission policies as they can defer decisions to the eviction policy, which has the advantage of knowing the item's usage while in cache.
- Flash caches incur write costs at insertion time, forcing admission policies to decide a priori to optimize the limited write budget. A longer residency better amortizes this upfront write cost. In contrast, the space-time cost of an item is incurred at a steady rate over time in DRAM caches.

We describe 4 challenges for ML admission:

Correct optimization metric is not obvious. The right metric is important not only because optimizing it gives better performance, but because it makes the system more robust. Systems practitioners know the importance of using end-to-end metrics such as IO hit rate, rather than cache hit rate (problem: an IO hit can require multiple cache hits) or ML model accuracy (problem: asymmetrical misprediction cost and class imbalance). Yet even optimizing for IO hit rate is still an (easy) misstep, as a policy that increases the IO hit rate but consumes much more bandwidth may result in overall higher DT, and require more HDDs to serve that load.

The cost of a misprediction is asymmetrical. Mispredictions consist of false positives (FPs) and false negatives (FNs). A FP incurs a full write cost (reducing writes left for true positives), and time in cache. FPs have a large performance impact since given the limit on flash writes. With an FN, a hit is lost but the policy may have further chances to admit the item. These lost hits are insignificant for popular items, but have an outsized impact on items with only a few potential hits. There is a long but heavy tail of such items; our traces show many admitted items with 5–8 hits (Fig 5.1). Policies trading off too many FNs for FPs suffer a performance hit [96].

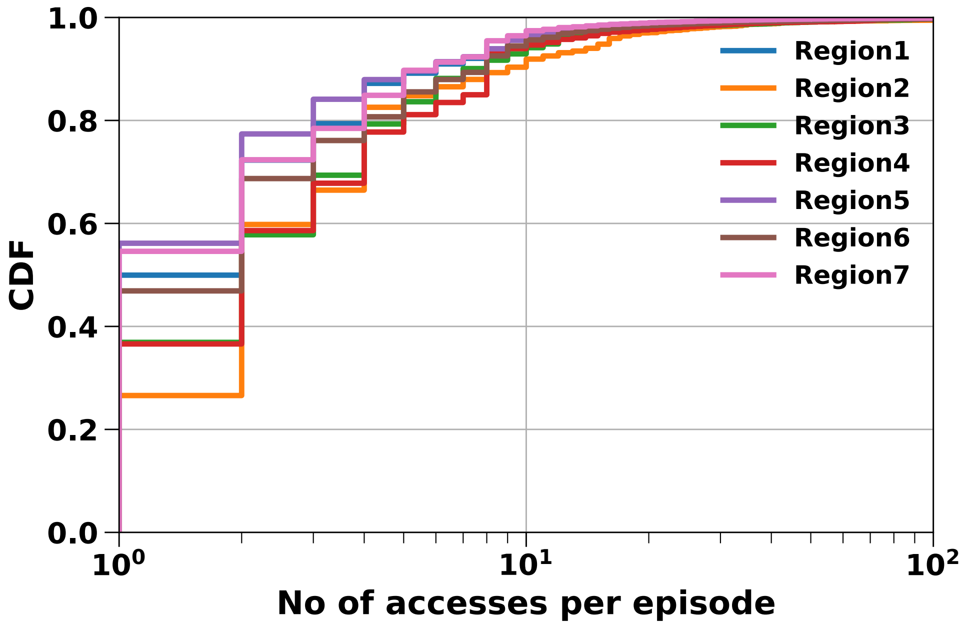


Figure 5.1: Distribution of hits per episode. This reflects the possible hits accrued from admitting an item. The majority of episodes have only 1–2 hits, with many admitted episodes having only 5–8 hits.

Classes are highly imbalanced. Since most items will not be admitted (94% in our experiments), true negatives (accesses that should not be admitted) far exceeds the number of true positives (accesses that should be admitted). Indeed, we observe that while ML admission policies may achieve a high ML accuracy, this does not always translate into a high cache hit rate. We found typical solutions (oversampling, undersampling, and sample weights) ineffective at countering the extreme imbalance.

Admission policies operate only on misses. For an ML policy, it makes sense to train only on accesses in a trace that result in misses, rather than all accesses in the trace. However, this requires an online simulation to determine which accesses are misses, adding additional complexity to training.

5.1.1 Design and implementation

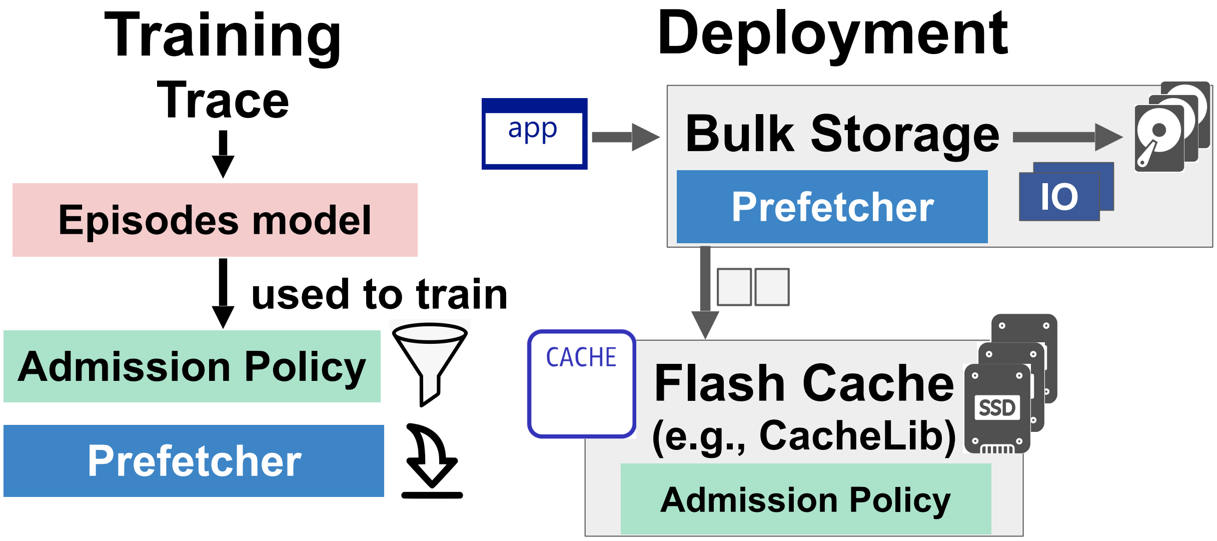


Figure 5.2: Architecture. An admission policy in CacheLib decides whether to admit items into flash. Prefetching (preloading of data beyond current request) takes place in Tectonic.

Episodes generated from the trace are used to train an admission policy, as shown in Fig 5.2. The policy is a binary classification model. We describe:

1. how we generate training data and labels from episodes,
2. what features and architecture we use for the ML admission model,
3. how we determine appropriate values for training parameters (assumed eviction age, admission policy threshold) through an iterative loop, and
4. how we implement ML admission in CacheLib.

Features Baleen's admission policy utilizes a total of 9 features, grouped into offline metadata and online usage counts.

Metadata features are provided by the bulk storage system and supplied in the trace. These metadata features identify the provenance of the request (namespace, user) and indicate whether the block is tagged as temporary (e.g., as a result of a JOIN) or permanent. Feature cardinality is less than 100 for namespaces and less than 200 for users. Both features are associated with the system user (internal service) executing the request rather than an end user. These features are often the same for accesses to the same object and almost always the same for accesses belonging to the same episode. These features are provided per IO and thus the same for all segments.

Online dynamic features (times the item is accessed in the last 1,2, . . . 6 hours) change with every access. This can be measured at the block or segment level. For Baleen, we record both the number of IOs for each block and the cumulative segment accesses for each block to use as features. For each workload, a simple simulation is done on the training set (the first day) to collect these dynamic features. We do not use individual segment counts as features, as this would add 64 features without an appreciable increase in performance.

Modeling admission as binary classification We admit misses if the classifier's output probability exceeds the policy threshold. We also considered regression models (e.g., predicting no. of expected hits). Such models eliminate the threshold parameter, but we found they perform worse end-to-end [21], perhaps because their loss functions incentivize performance at all thresholds and not just those at the boundary.

Training data and label generation The goal is to differentiate episodes at the decision boundary, which tend to have few accesses. Learning to identify these episodes is hard but important as they are significant in aggregate. To avoid a training bias towards popular but easy-to-differentiate episodes, only the first 6 accesses from each episode are incorporated into training data. Baleen learns to imitate OPT, and the binary labels are determined by whether that episode, based on its score, would have been admitted under OPT.

Gradient boosting machines (GBM) We chose to use GBMs as they are fast and have some inherent tolerance to overfitting and imbalanced classes. Compared to deep neural networks, they are far more efficient and are well-proven to run within the latency requirements of a production caching system [5]. Practitioners also find them easier to understand, given that they are based on widely-understood decision trees.

Adding a ML admission policy to CacheLib The open-source version of CacheLib supports flash admission policies, but does not include a mechanism for storing and supplying features to ML admission policies. We describe how this may be done. For the static metadata features, they can be embedded as part of the item payload. Since payloads are a few MB on average, storing the features (less than 1 kB) in this way does not impose any significant overhead. To provide the dynamic features, counts of accesses are tracked in CacheLib using a count-min-sketch data structure (similar to bloom filters, but with counts). Each data-structure holds the count for approximately one hour, with a queue of 6, such that we have counts at hour-level granularity for the last 6 hours.

5.2 ML for prefetching

On a miss, a backend IO must be made to retrieve all missed segments. This IO can be extended and more segments admitted. Done correctly, compulsory misses (when a segment is first observed) are eliminated, reducing disk-head time. However, prefetching mistakes are costly as they consume both writes and extra DT.

Next, we describe the design of our ML prefetching policies. We train models to solve two subproblems: what to prefetch, and when to prefetch.

5.2.1 Learning what to prefetch: ML-Range

We need a ML model that predicts a range of segments for prefetching. We do this by training the model to imitate *OPT-Range*, the smallest range of segments needed for all accesses in an episode to be hits (defined in §4.4). We use the same metadata features as the ML admission model (namespace, user, temporary/permanent flag), but add size-related features (access start index, access end index, access size). We train two regression models to predict the episode range start and end. Each episode is represented once in the training data, with only episodes that meet the score cutoff for the target write rate included. As training data, we use the episodes that would be admitted according to the analytical model for the target write rate.

5.2.2 Learning when to prefetch: ML-When

Mispredictions by the ML admission policy and in ML-Range can easily cause prefetching to hurt instead of help. In reality, the expected benefit will be lower than OPT prefetching and the cost can only be higher. DT saved from prefetching ML-Range may not be realized (which we call underfetch, see Eq 5.1a). Further, prefetching mispredictions are costly in terms of DT consumed to fetch unused segments (which we call overfetch, see Eq 5.1b) and the opportunity cost of flash writes used to store them.

ML-When aims to address this by **excluding episodes that do not have a high probability of benefiting from prefetching**. In particular, it hedges against the broader effect of prefetching on eviction age by requiring that the marginal DT gained from ML prefetching ($PFBenefit_{eps}^{ML}$, Eq 5.1c) be larger than ϵ (ML-When label, Eq 5.1e). ϵ is a proxy for the unknown broader opportunity costs of flash writes and cache space, and set to 5 ms (for comparison, an IO seek is 12 ms).

$$UF : underfetch = true \text{ if ML-Range} \subset \text{OPT-Range} \tag{5.1a}$$

$$OF : overfetch = DTUsed(\text{extra segments}) \tag{5.1b}$$

$$PFBenefit_{eps}^{OPT} = DT_{eps}^{NoPrefetch} - DT_{eps}^{OPT-Range} \tag{5.1c}$$

$$PFBenefit_{eps}^{ML} = \begin{cases} 0 & \text{if underfetch} \\ PFBenefit_{eps}^{OPT} - OF & \text{otherwise} \end{cases} \tag{5.1d}$$

$$\text{ML-When}(eps) = PFBenefit_{eps}^{ML-Range} > \epsilon \tag{5.1e}$$

For general, non-ML-specific details of prefetching in CacheLib, please see §3.3.

5.3 Evaluation

This section evaluates and explains Baleen's effectiveness in reducing backend peak load for 7 real workload traces.

5.3.1 Baleen reduces Peak DT over baselines

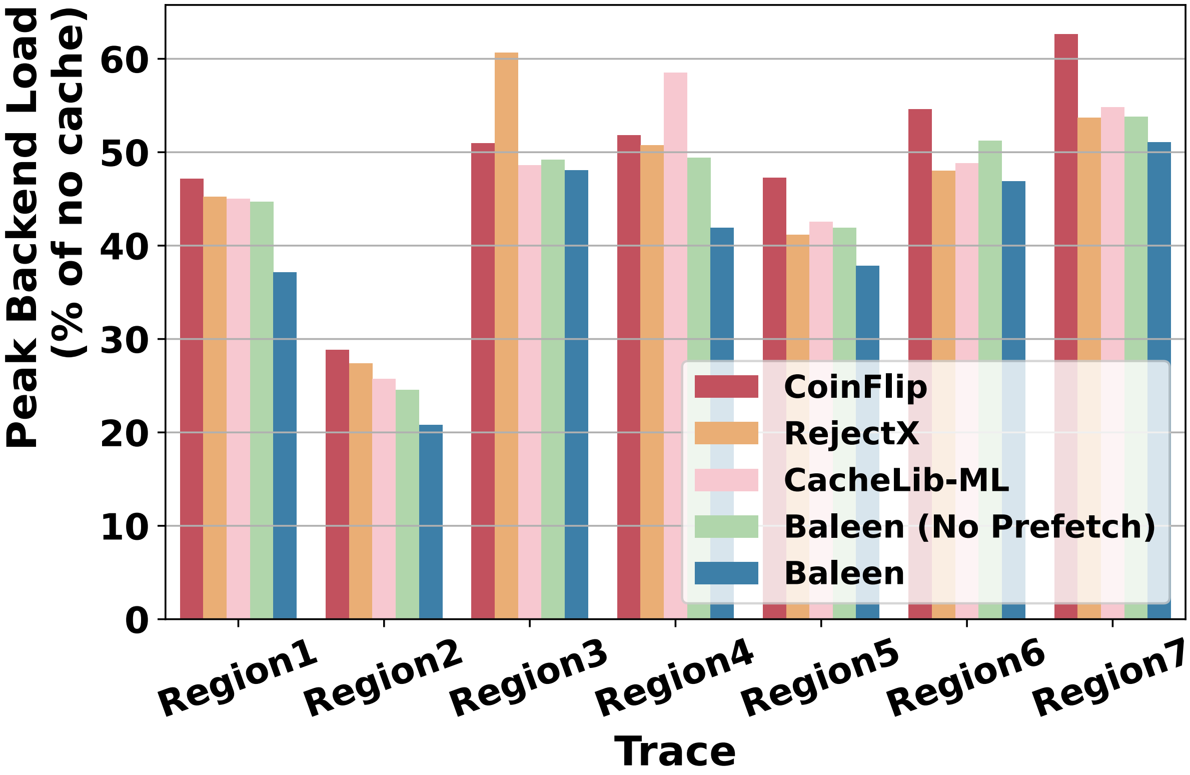


Figure 5.3: Baleen reduces Peak DT.

Fig 1.1b shows Baleen reduces Peak DT over RejectX by an average of 12% across all traces for a fixed target flash write rate. Fig 5.3 shows this ranges from 5% to 29% across the traces. Region1, Region3 and Region4 derive most of their gains from prefetching.

Flashield is not shown in the graphs as it failed on half the trace samples due to insufficient training data (more details in § 5.4.2). If we consider only workloads Flashield could train a model on, Baleen outperformed Flashield by 18%.

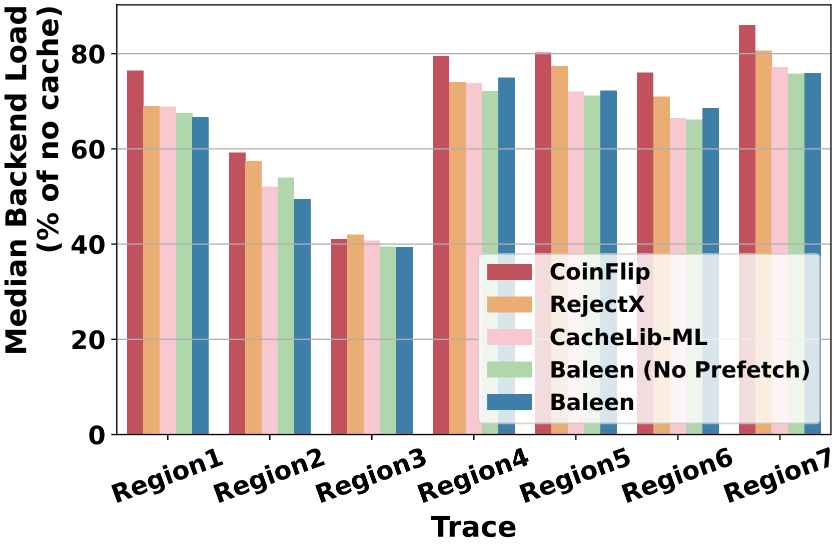


Figure 5.4: Median DT

In Fig 5.4, we show that Baleen also reduces Median DT over RejectX by X% acrsos all traces for a fixed target flash write rate. In our experience, we find that doing better at the peak usually means that the median also improves and generally does not result in a regression at the median. However, the converse is not true, which we will talk about in §5.4.

Reducing the peak Fig 5.5 shows the load over the 7-day Region1 trace in a testbed experiment for Baleen and baselines. We can observe that there is a large reduction in the peak on Day 5 when using Baleen.

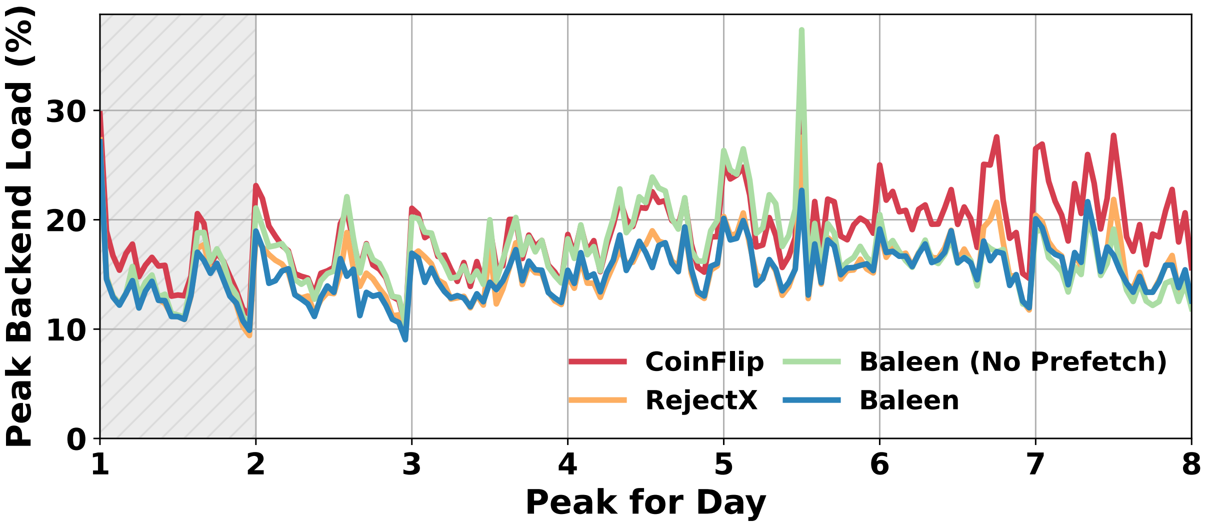


Figure 5.5: Testbed backend load on Region1. Day 1 (shaded) is used as training data. Peak is on Day 5 and is lowest for Baleen.

Training on episodes (instead of accesses) is essential to ML prefetching Episodes make it easier to reason about flash caching and was key to designing both OPT and ML prefetching. We also found that in the absence of episodes, others in the literature devised ad-hoc sampling heuristics that would achieve the same goal of avoiding ML training bias towards popular objects [68]. In addition, we quantify the benefit of episodes by comparing Baleen to an earlier ML admission policy that did not use episodes (CacheLib-ML). Adding prefetching to RejectX or CacheLib-ML would cause it to perform worse than without prefetching.

Benefits consistent at higher write rates and larger cache sizes Fig 5.7 shows that Baleen allows for a reduction in cache size by 55% while keeping the same Peak DT as RejectX, or alternatively a reduction in Peak DT equivalent to a 4X increase in cache size. As expected, increasing write rate or cache size has diminishing returns in reducing Peak DT. Also, the different admission policies (without prefetching) start to converge, indicating that admission by itself is insufficient to drive further reductions in Peak DT.

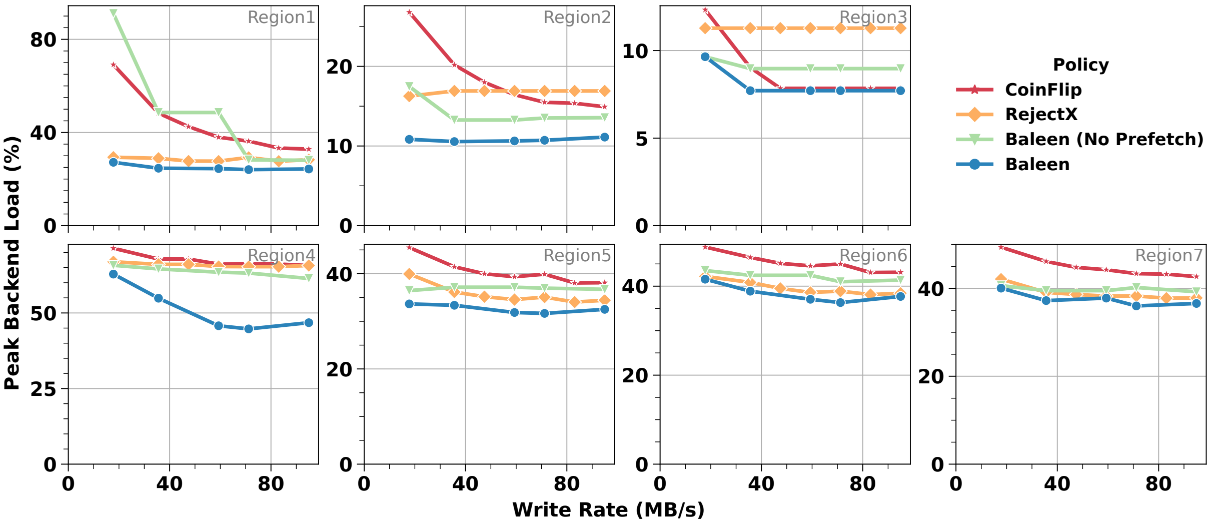


Figure 5.6: Benefits consistent as write rate increases.

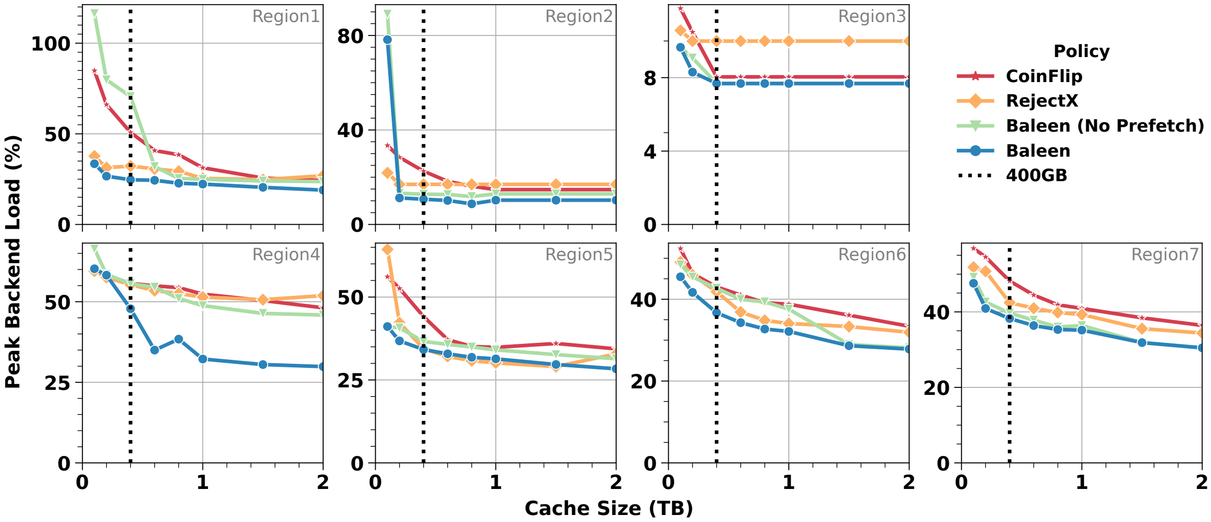
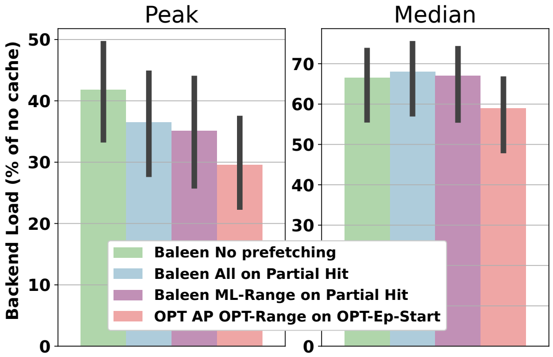
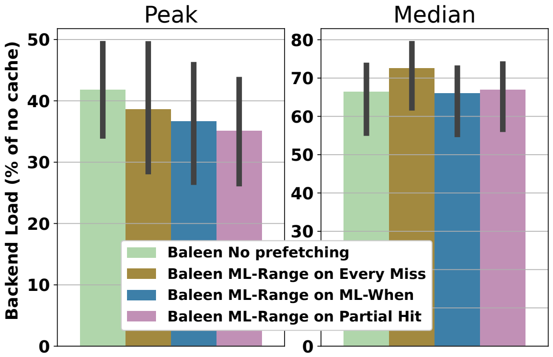


Figure 5.7: Benefits consistent as cache size increases.

5.3.2 Prefetch selectively, in tandem with admission





]

Figure 5.8: ML-Range saves Peak DT. ML-Range outperforms the baseline (whole block) and No Prefetching at the expense of Median DT.

Figure 5.9: Choose *when* to prefetch. Indiscriminate prefetching (on Every Miss) can hurt. Using ML-When or Partial Hit reduces Peak DT without compromising Median DT.

We show both ML-Range and ML-When are effective in reducing Peak DT over static baselines, and contribute to Baleen's robustness across the multiple traces. We also show that prefetching must be paired with a good admission policy; if not, the same prefetching policy can hurt rather than help.

We defined the static prefetching baselines (such as fetching the Whole-Object always, or on Partial Hit) in §4.4, in addition to an approximate optimal for prefetching (*OPT-Range* and *OPT-Ep-Start*).

ML-Range outperforms no prefetching and fixed range prefetching. Using ML to decide what to prefetch (ML-Range) saves 16% of Peak DT over no prefetching, and 4% over a simple baseline (All on Partial Hit) (Fig 5.8). Baleen admission is used in all cases, with only the prefetching policy varied. We note this comes with a small increase in Median DT. When we use Baleen admission with OPT prefetching (not shown in Fig 5.8 for brevity), it betters it (possible as OPT-Range does not account for mistakes by Baleen's ML admission policy), suggesting that the admission policy is the limiting factor rather than ML-Range.

ML-When helps Baleen discriminate between beneficial and bad prefetching. ML-When expresses Baleen's confidence in the quality of its ML-Range prediction. A general challenge with prefetching is that one is predicting without a direct signal (such as a miss in the case of admission). If used indiscriminately, prefetching can hurt rather than help. This is best illustrated by how prefetching ML-Range on Every Miss is worse than no prefetching in Fig 5.9. Prefetching only on ML-When or on Partial-Hit consistently does better than both no prefetching and prefetching on every miss across all traces.ML-When performs better on 2 traces (Region2, Region7) and Partial Hit on the remaining 5.

Poor admission decisions lead to poor prefetching ML prefetching reduces Peak DT most when paired with a good admission policy like Baleen. With RejectX, prefetching is less helpful or even hurts (in Region7). Thus, the Baleen admission policy is important to the performance of prefetching despite not always reducing Peak DT by itself. Adding prefetching to CoinFlip yielded results similar to RejectX.

5.4 Importance of optimizing the right metric: Disk-head Time

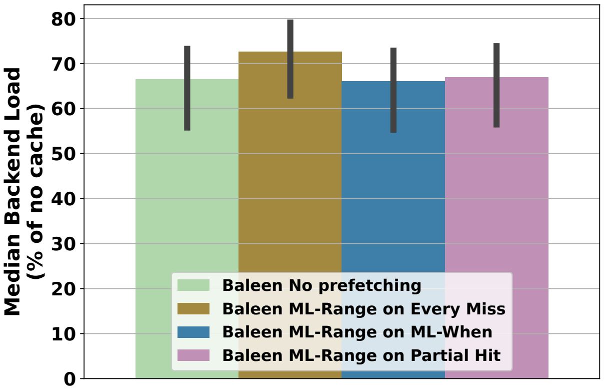


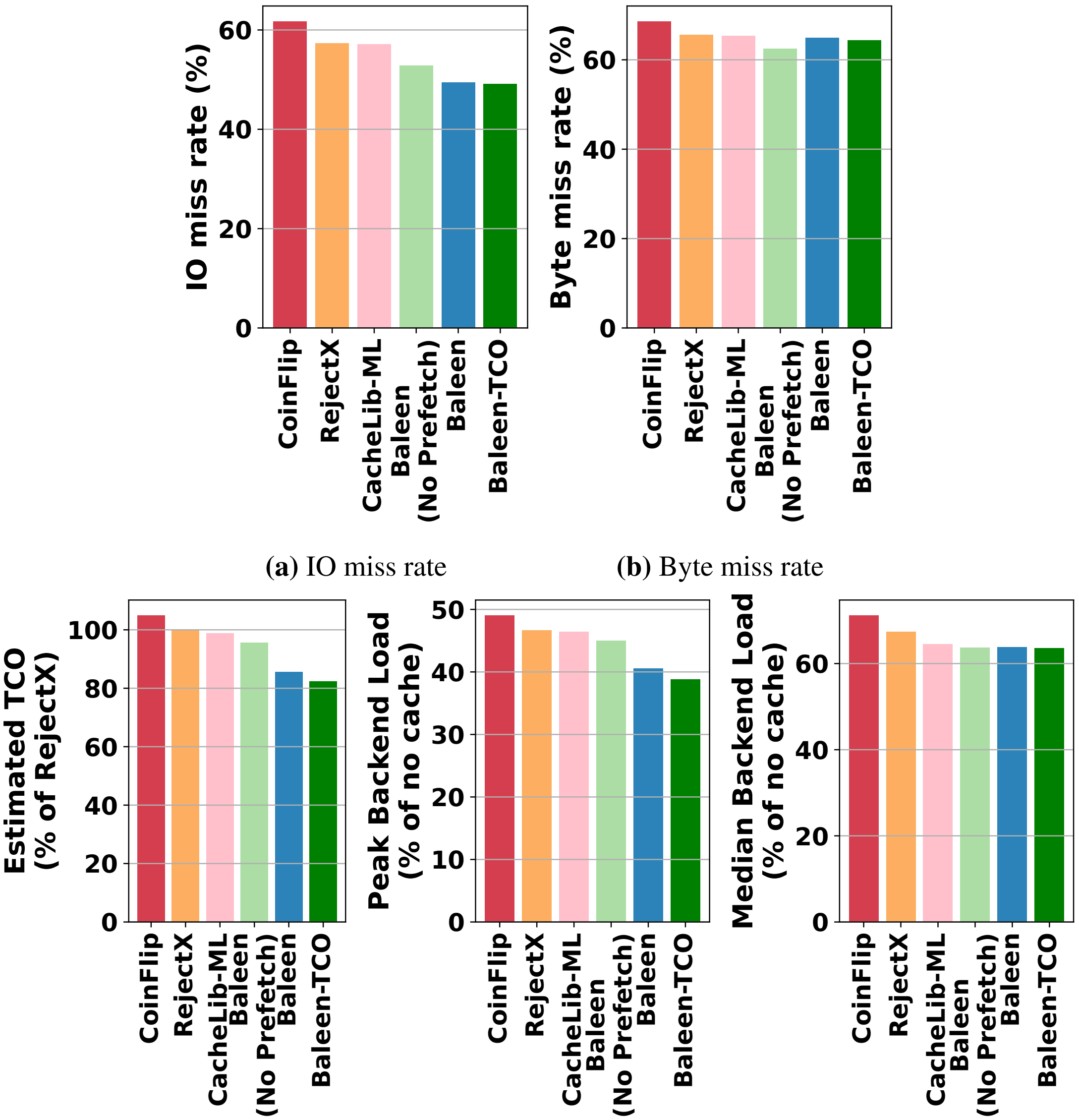
Figure 5.10: Importance of optimizing Disk-head Time instead of hit ratio. Using the "ML-Range on Every Miss" prefetching option resulted in an improvement in IO hit rate at the expense of Disk-head Time.

Optimizing for IO hit ratio can be misleading as doing so is optimal for reducing seeks, not total disk-head time. Policies that do so may reduce IOs at the expense of increased bandwidth, which can be a net loss in bandwidth constrained systems. For the prefetching option "ML-Range on Every Miss" from Fig 5.10, relative to no prefetching, the mean Disk-head Time used ratio worsened from 67% to 73% despite the IO hit ratio increasing from 46% to 47%.

5.4.1 Reductions in IO miss rate, bandwidth miss rate

We show that Baleen, in reducing Peak Disk-head Time (and TCO; more to come in §6), also improves the commonly used metrics of IO miss rate and byte miss rate. For ease of comparison, we also show median load, TCO and peak load.

We can infer from Fig 5.11a and Fig 5.11b that Baleen's prefetching improves IO miss rate at the expense of byte miss rate (the difference between the light green and blue bars), with an overall reduction in peak load and thus TCO.



(a) IO miss rate

(b) Byte miss rate

(c) Estimated TCO

(d) Peak load

(e) Median load

Figure 5.11: Baleen also reduces IO miss rate and byte miss rate, two commonly used caching metrics.

5.4.2 Comparison to ML baselines: Flashield and CacheLib-ML

Flashield We compared Baleen to Flashield, a state-of-the-art ML baseline. We adapted the implementation of Flashield used in the S3-FIFO paper in SOSP 2023 [94]. Flashield was worse than our RejectX baseline.

In practice, we found that a disadvantage of this approach is that DRAM lifetimes are too short to yield useful features. (Flashield assumes a 1:7 DRAM:Flash ratio, whereas Tectonic has a 1:40 ratio.)

Flashield failed on half the trace samples due to insufficient training data, because it relies on items' hits in DRAM for its features and labels. With DRAM lifetimes of seconds-to-minutes, most items never receive DRAM hits. Considering only workloads favorable to Flashield (that it could train a model on), Baleen outperformed Flashield by 18%.

Comparison to CacheLib-ML CacheLib-ML is a ML model that Meta used in production for 3 years, which was first described by Berg *et al* [5]. Baleen uses the same ML architecture (GBT) and serving (inference) setup, but a different training setup (episodes and optimizing DT instead of hit rate). Based on this, we assert that Baleen's architecture is feasible for production with acceptable inference overhead. Meta's implementation is proprietary but general lessons learnt from it were described in §9.7.

5.4.3 Overhead

Baleen's runtime overheads are low in the context of caching for bulk storage systems.

CPU inference overhead Baleen adds 4 inferences per IO miss (admission, start & end of ML-Range, ML-When). The system is limited by the latency of disk IOs upon misses (10–70ms per IO) rather than ML inferences (30 microseconds per inference). Even when replaying a trace at full speed, CacheLib only contributes a small fraction of overall system CPU utilization (5% of the 16-core CPUs in our testbed) because it is waiting for disk IO, and thus using ML policies only translate to an additional 1% increase in overall CPU usage.

Metadata overhead Baleen also stores static metadata features in the payload (<1kB), but as payloads are at least 128KB, this overhead is not significant (<1%).

Training overhead Baleen uses gradient boosting machines and is able to get good results even with highly sampled training data. Training is done on the CPU and takes under a minute, with training duration never coming close to being a limiting factor.

5.4.4 Validation of Baleen on testbed

As we did not have direct access to production hardware, we ran simulations (on our Python simulator, BCacheSim) and testbed evaluations (using our modified version of CacheLib) on our academic testbed. We described this testbed and validated it against production using baseline policies and production counters earlier in §3.4.

Fig 5.12 shows us validating Baleen on our simulator against Baleen on our testbed.

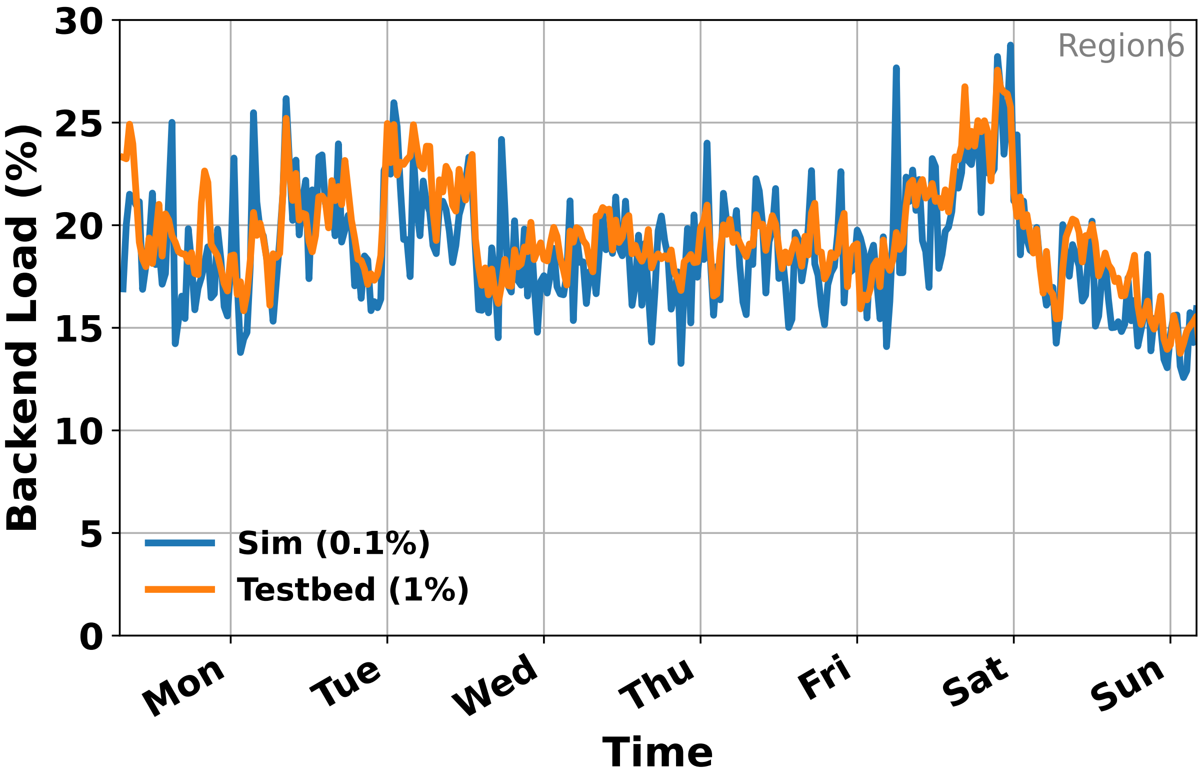


Figure 5.12: Sim vs Testbed, Baleen

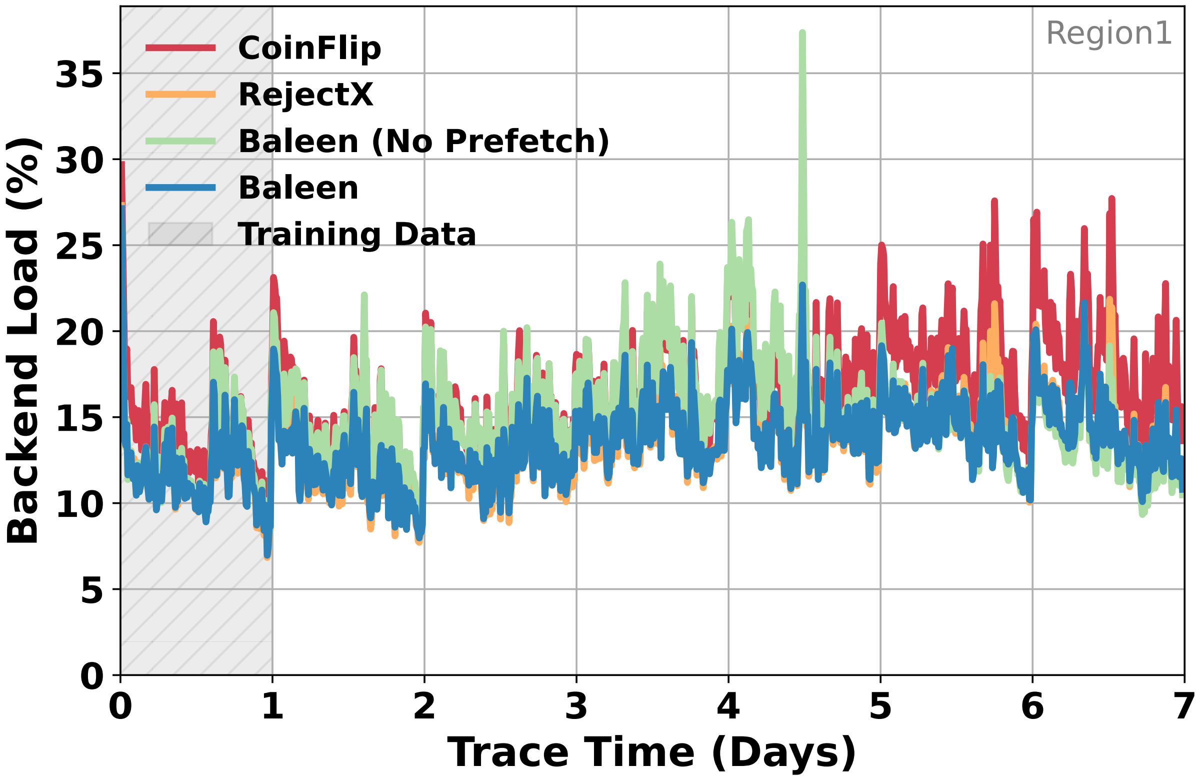


Figure 5.13: Testbed backend load over time, on the Region1 trace. Peak-to-mean ratio is 2. Granularity is 10 mins.

In Fig 5.13, we show Baleen and the other baselines on Region1 evaluated on our testbed, with each point on the graph representing the backend load over a 10-minute window.

5.5 Summary

Baleen uses episode-guided ML to guide both prefetching and cache admission, reducing peak disk time by 16% and TCO by 17% on real workload traces, compared to state-of-the-art ML policies. Although applying ML to caching policies is an expected advancement, Baleen's design arose from false-step lessons and a cache residency formulation (episodes) that improves training effectiveness, provides a target (OPT), and exposes the value of ML-guided prefetching. As such, Baleen is an important step forward in flash caching for disk storage.

Chapter 6

Baleen-TCO: choosing the best parameters to minimize cost

In this chapter, we describe the TCO function we devised and Baleen-TCO, an extension of Baleen which minimizes our TCO function.

A TCO function allows us to quantify the cost impact of changes in caching policies, which is important for multiple reasons:

1. **Evaluating the development cost of caching improvements** Designing new caching policies requires the investment of valuable software engineering man-hours.
2. **Comparing caching improvements against other non-caching opportunities** Being able to consider the opportunity cost of investing in caching is important. For instance, improving another part of a storage pipeline may not be directly measurable in cache metrics and thus difficult to compare without a TCO function. Moreover, even within the realm of caching, not everyone is mentally calibrated and can quickly translate the impact of, say, a 20% reduction in Peak Disk-head Time, into a ballpark cost reduction figure.
3. **Picking a point on the cache's pareto frontier** Each policy represents a set of possible points. It is desirable to automatically determine the optimal flash write rate, and additionally capture benefit from being able to vary it by workload instead of using a single static flash write rate target.

However, there are difficulties that come with using the true TCO (total cost of operation) as a metric:

1. **It may reveal commercial secrets.** Exact costs (e.g., of procuring new equipment or power prices) are often under non-disclosure clauses in contracts. It could also be used to infer the volume of a business and thus reveal material financial information. It may also introduce legal risk by providing ammunition for opponents in lawsuits.
2. **It is not a 'pure' reflection of system performance.** Costs can change over time for reasons unrelated to system performance, making TCO numbers challenging to compare and interpret over time (whereas the meaning of a 60% hit-rate, for instance, does not change).
3. **Devising a model for TCO can itself be costly.** Coming up with an all-encompassing cost model for a system is itself a research endeavor and can require significant effort to collate

and keep up-to-date. Yet some sort of model is required for an algorithm to optimize it.

We strike a balance between these trade-offs in designing our TCO formula, which allows us to measure (and thus optimize) the percentage of cost reduction, without needing to work with (or reveal) exact dollar costs. We focus on media costs (cost of SSDs and HDDs), which are known to dominate the true TCO function [96, 97]. Moreover, for a fixed hardware generation where we can only vary the amount of storage capacity by varying the number of servers, we assert that many of the variable non-media costs (e.g., power, CPU, networking) will also grow proportionately to the media costs.

6.1 Background: TCO dominated by backend HDDs required

Related work on caching's impact on TCO in bulk storage systems CacheSack [97] uses a TCO function (described only as the cost of disk reads and the cost of written flash bytes) and finds the optimal policy per category (a feature) that minimizes TCO. They reported a 7.7% TCO reduction, with a 9.5% reduction in disk reads and a 17.8% reduction in flash writes. Note this is not directly comparable to our results since we are using different workloads.

Designing a TCO function In the absence of actual cost numbers, we approximate TCO (total cost of ownership) based on public information. We design a function that is focused on the same cost components as [97] (SSD writes and HDD reads).

Weassuming that the cost of HDD reads is proportional to the HDDs required (and Peak DT), and the cost of written flash bytes is proportional to the SSDs purchased in the long run:

$$TCO \propto Cost_{HDD} \cdot \#HDDs + Cost_{SSD} \cdot \#SSDs \tag{6.1}$$

We calculate relative TCO savings using the Peak DT saved with our baseline AP RejectX ($PeakDT_0$), and relative to the default target flash write rate ($FlashWR_0$).

$$\mathbf{TCO_1} \propto \frac{\mathbf{PeakDT_1}}{PeakDT_0} \cdot \#HDDs_0 + \frac{Cost_{SSD}}{Cost_{HDD}} \cdot \frac{\mathbf{FlashWR_1}}{FlashWR_0} \cdot \#SSDs_0 \tag{6.2}$$

This gives us a TCO function based on a policy's Peak DT ($PeakDT_1$) and the flash write rate chosen ($FlashWR_1$).

The skewed ratio of HDD to SSD capacity in bulk storage systems like Tectonic (945:1 [58]) means that SSD cost is a fraction of TCO relative to HDD cost (3% on our workloads), even though SSDs are more expensive on a per-GB basis. Hence, reducing Peak DT (and HDDs needed) is key to reducing TCO.

Comparison with CacheSack Like CacheSack, our function focuses on the two major elements of SSD writes and HDD reads. However, we have a number of differences: 1) We provide a detailed derivation of the function. 2) They assume that that other costs (CPU, RAM, power, network) are negligible; we assert that they are not (and have confirmed this with industry experts). However, we assert that for a given hardware generation, we make scaling changes by changing the number of servers and not how much storage is inside each server, and we assert thus most of these costs (CPU, RAM, power, network) will scale proportionally with the the media costs and thus can be dropped to simplify the TCO function. 3) We focus on the cost of Disk-head Time at the **peak**, instead of the mean number of disk reads (or equivalently, hit ratio).

6.2 Deriving a TCO function based on public data

We provide a line-by-line derivation of Eq 6.2 below.

$$
\begin{aligned}
\mathbf{TCO_1} &\propto \frac{\mathbf{PeakDT_1}}{PeakDT_0} \cdot \#HDDs_0 + \frac{Cost_{SSD}}{Cost_{HDD}} \cdot \frac{\mathbf{FlashWR_1}}{FlashWR_0} \cdot \#SSDs_0 && \text{(6.3a)}\\
TCO_1 &\propto PeakDT_1 \cdot R_1 + FlashWR_1 \cdot R_2 && \text{(6.3b)}\\
R_1 &= \frac{1}{PeakDT_0} && \text{(6.3c)}\\
R_2 &= \frac{1}{FlashWR_0} \cdot \frac{\#SSDs_0}{\#HDDs_0} \cdot \frac{Cost_{SSD}}{Cost_{HDD}} && \text{(6.3d)}\\
&= \frac{1}{FlashWR_0} \cdot \frac{1}{36} \cdot \frac{170}{281} && \text{(6.3e)}
\end{aligned}
$$

We calculate relative TCO savings using the Peak DT saved with our baseline admission policy RejectX ($PeakDT_0$), and relative to the default target flash write rate ($FlashWR_0$). From [58], we know that each node has 1x 1-TB SSD and 36x 10-TB HDDs ($\frac{\#HDDs_0}{\#SSDs_0} = 36$).

The price of storage procured by hyperscalars is commercially sensitive and would be lower than prices available to the public. However, we assume that the percentage savings the hyperscalars can extract in their contract negotiations is similar for both SSDs and HDDs, and thus would cancel out in our formula since we use only the ratio. This allows us to use public prices of SSD and HDD storage, which we take from Newegg. We substitute the 2023 price of a 10TB HDD ($281) and a 1 TB SSD ($170) on Newegg [56, 57] ($\frac{Cost_{SSD}}{Cost_{HDD}} = \frac{170}{281}$), i.e., the HDD is 6x cheaper per TB than the SSD. As a point of comparison, a 2020 industry report stated a 10x difference [54], which is in the same order of magnitude as the 6x used in our calculations.

6.3 Baleen-TCO

Vanilla Baleen allows us to determine the Peak Disk-head Time savings from a cache policy at a fixed flash write rate and cache size. Baleen-TCO optimizes our TCO function by simulating Baleen over a range of flash write rates (as illustrated in Fig 6.1) to get the respective Peak DT. Baleen-TCO then chooses the optimal flash write rate to minimize the TCO function. This is done on a per-workload basis, since the optimal flash write rate can vary between workloads. We also considered allowing it to vary over time, but this did not seem necessary, at least in the 7-day workloads we evaluated Baleen-TCO on, and was also more realistic given the relative inelasticity (from a planner's point of view) of capacities in the bulk storage systems we know.

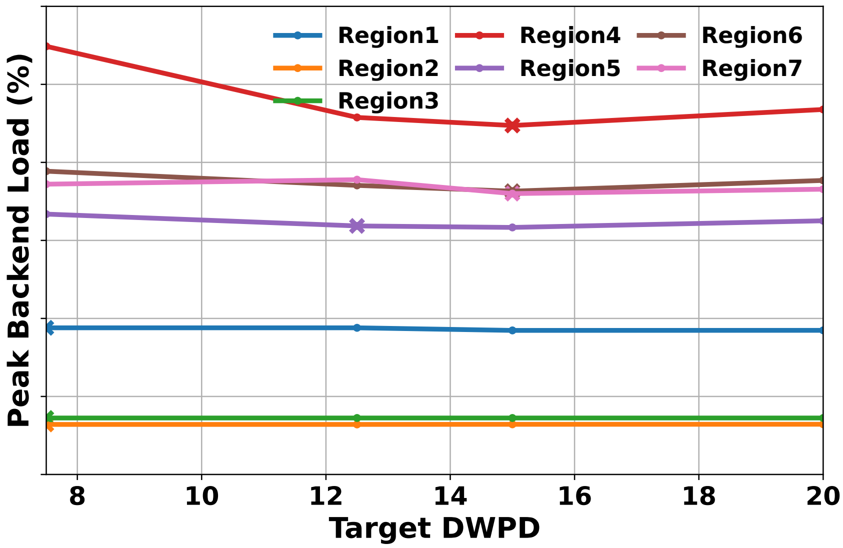


Figure 6.1: Baleen-TCO reduces TCO by choosing a higher flash write rate when justified to lower peak backend load. × denotes the optimal flash write rate for that workload.

Baleen-TCO can be easily adapted for other companies or deployments (or even used to predict what may change with changes in media cost ratios caused by advancements in technology), since it allows for a different flash-to-HDD cost ratio to be substituted in.

6.4 Evaluation: Baleen-TCO chooses optimal flash write rate

Workloads have different optimal flash write rates. In Fig 6.1, we observe that some workloads, like Region4, benefit from an increased flash write rate budget, whereas others do not, in which case one should reduce costs from flash. These higher optimal flash write rates were significantly higher than the static flash write rate budget previously set by Meta, thus unlocking even more savings from Baleen. One should also note that there is no single flash write rate that is optimal for all workloads, justifying the additional complexity from this adaptive strategy.

Baleen-TCO saves additional 4% of TCO over vanilla Baleen. Fig 6.2 shows Baleen-TCO reducing TCO by 17% over CacheLib-ML and 18% over RejectX. If a constant flash write rate target is used, Baleen is able to reduce TCO by 14% over RejectX. Thus, Baleen-TCO saves an additional 4% over Baleen with a fixed write rate. We note that there is an improvement in all regions except Region1 and Region3, and there are no regressions in TCO.

Flash writes account for 2% to 5% of TCO (3% on average). While 5% may seem small, it is still significant as it means that adopting an inferior flash admission policy can rapidly increase costs (say, if the bad policy requires double the flash write rate, that means TCO could jump by 5%).

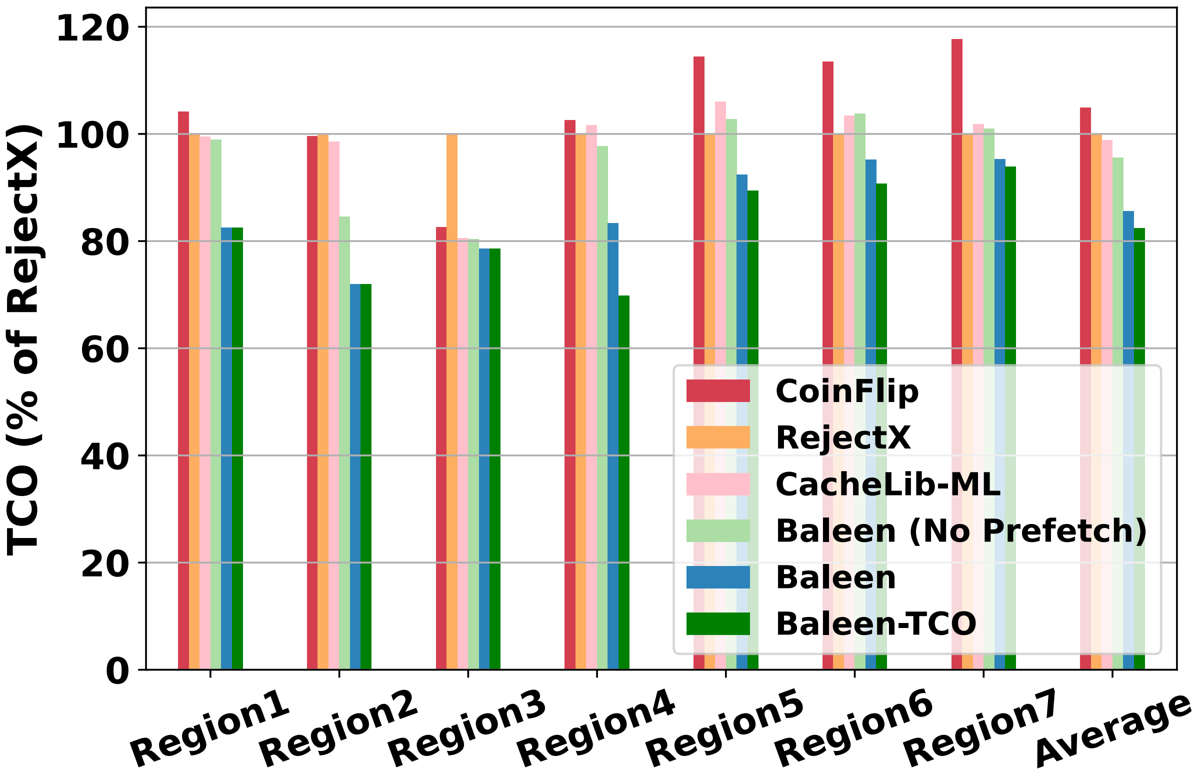


Figure 6.2: Baleen-TCO reduces TCO across all traces. On average, Baleen-TCO saves an additional 17% over CacheLib-ML, 14% over RejectX, and 4% over Baleen with a fixed flash write rate.

Chapter 7

Optimizing for peak load

Ideally, backend capacity should be provisioned to handle peak load in order to fulfill assurances and SLOs (service-level objectives) to customers during periods of high load. In practice, existing caching systems optimize for average load, rather than peak load. In this chapter, we show why it is important to optimize for peak and not not average load (as we did initially), our early attempts to optimize indirectly for peak load, and our subsequent attempts to optimize for the peak explicitly.

7.1 Background

Caches are crucial to reducing peak backend load in modern clouds [62, 69]. Although this is widely accepted, most large-scale caching works evaluate average performance instead of at peak load (Table 2.6), with only one work other than Baleen doing so [69], with none explicitly optimizing for the peak. This presents missed opportunities, as 1) it may be possible to decrease the peak further if explicitly optimizing for it, and 2) further resource savings are possible at off-peak, e.g., by admitting less during off-peak periods to save on flash writes. Previous works on caching at the block I/O level have reduced peak load by offloading IOs to underutilized hosts [55] decomposing and rescheduling data flows [46], traffic shaping [81], or resizing cache according to load [109]. These solutions are not suitable for the modern data center cache as they either require modifications to the storage system itself (some of which is already in place, such as traffic shaping [58]), or assume the problem can be pushed further down the stack to cloud providers. Bulk storage systems consolidate storage needs of many subsystems; while this may smooth out some peaks in demand, the presence of co-mingled workloads also make optimizing the peak a more challenging task and less easy to optimize by hand, given that a policy that is optimal for one cluster may not be optimal for the next [97].

7.2 Indirect optimization for peak load

This section details early attempts at optimizing for Peak DT. In these cases, parameters are chosen to maximize Peak DT, while the underlying admission and prefetching policies still optimize for episodes' contribution to mean DT. This strikes a balance between introducing the additional complexity which would come by making it a scheduling problem, while still optimizing for the correct metric. This could be useful in guiding practitioners who want to modify an existing production cache to optimize Peak DT.

7.2.1 Choosing parameters to optimize for Peak DT.

Choosing prefetching method to optimize for Peak DT. Prefetching is key to Baleen's performance on most workloads, but on some workloads, ML-When is not aggressive enough as it optimizes for the mean, not Peak DT. To correct for this, we allow Baleen to choose another prefetching option per workload (e..g, *ML-Range on Partial-Hit*) if it is better at reducing Peak DT in training. This allows us to pick prefetching methods that optimize the peak but are suboptimal in terms of average Disk-head Time. Without this extra optimization, savings over RejectX would have been reduced from 12% to 6.6%. In Fig 7.1, we show how this extra optimization enables a reduction in Peak DT for Region4.

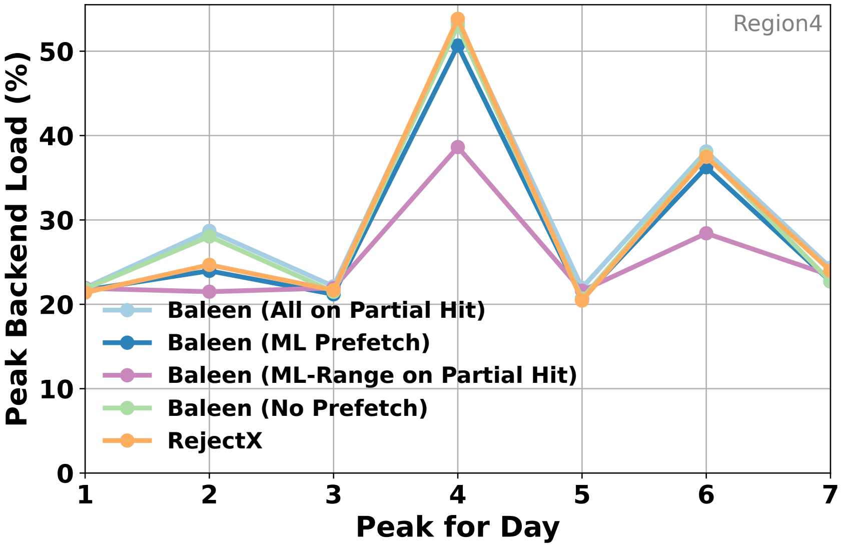


Figure 7.1: Choosing best prefetching method based on Peak DT in Region4. Picking the best prefetching method using Peak DT instead of Median DT enables a significant reduction in Peak DT. Here, the highest peak for each day (i.e., the maximum of the hourly peaks) is shown.

Baleen-TCO optimizes for Peak Disk-head Time. In the design of Baleen-TCO, we have it evaluate each flash policy write rate by the Peak Disk-head Time of policies performing on that level. Please refer to §6 for more details.

7.3 Analyzing trends in Peak DT over time

We first sought to gain intuition into trends in Peak DT over time in order to inform our attempts to explicitly optimize Peak Disk-head Time.

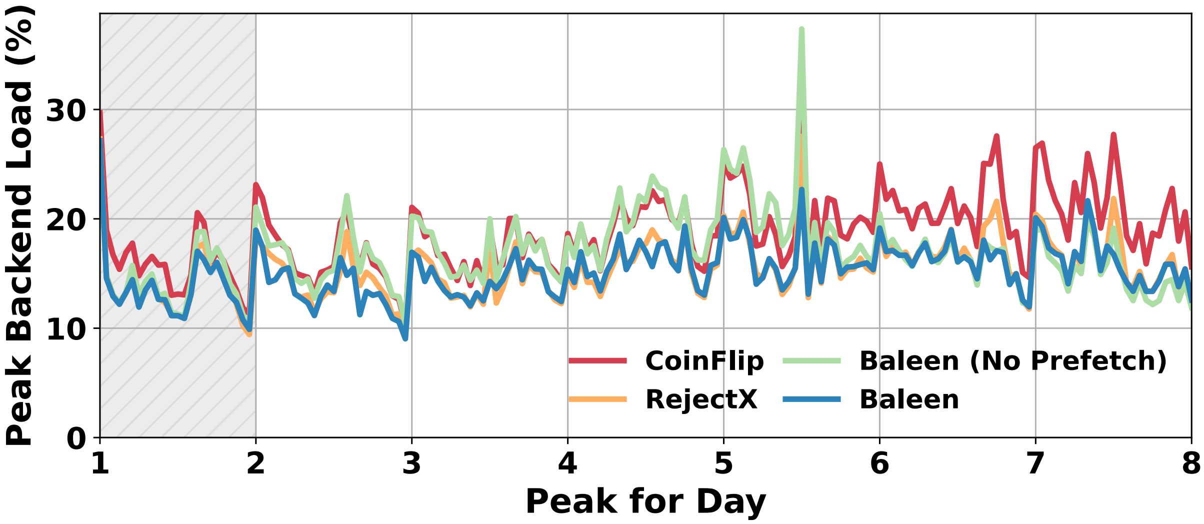
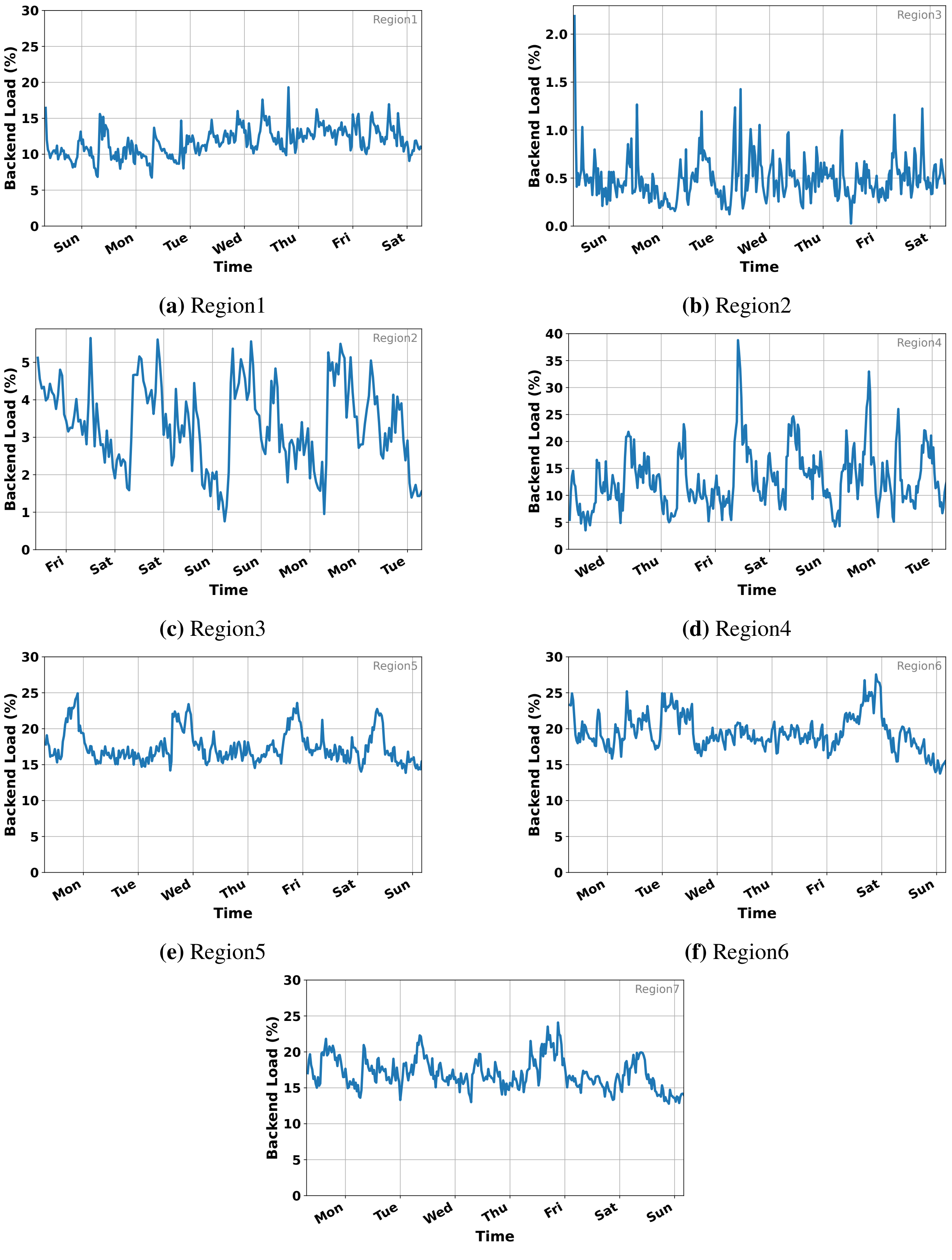


Figure 7.2: Load variation over a week in Region1. The peak is on Day 5. Smoothing (hourly averaging) has been applied, while the shaded area indicates the period used as training data.

Fig 7.2 shows the load variation over one week on a testbed (smoothed to hourly intervals). While the peak for all policies is in the middle of Day 5, the second-highest peak is different for different policies, if you look at Day 5 (CoinFlip and Baleen without prefetching) and Day 7.5 (Baleen and RejectX). While reducing the peak generally means reducing the average as well, what is useful for reducing one policy's peaks may not be as useful for a different policy. Noting that each point is one hour, we can see that while there are periods of sustained activity lasting days, individual bursts of load may only be 1-3 hours.

Peak periods are often sustained for less than a day, as shown in Fig 7.3, and often just an hour or two, which is close to the average eviction age of 2 hours. This means admission policies can in theory effectively reduce the peak, as the peak period is long enough to reap the delayed rewards from changes in the admission policy's decisions. (This delay refers to the time for a ML policy to detect increased load, adjust its admission decisions and get hits on its admitted items.) As these traces are only a week long, they do not show longer peak periods spanning days (e.g., Black Friday, major sports events) or weeks (e.g., company performance review).



(a) Region1

(b) Region2

(c) Region3

(d) Region4

(e) Region5

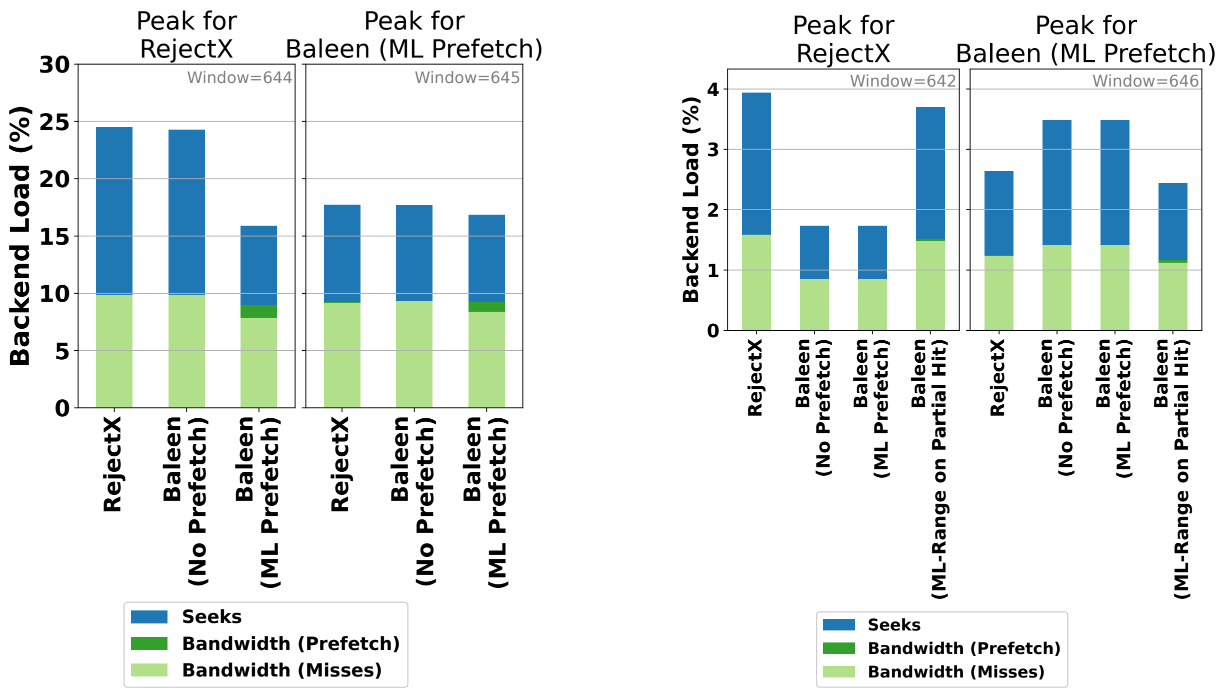
(f) Region6

(g) Region7

Figure 7.3: Workloads over a week with Baleen on our testbed. We observe that peak periods are typically sustained for 1-2 hours.

7.3.1 Breaking down DT at peak periods

Fig 7.4a breaks down DT at the peak hours. We observe that most of the reduction in Peak DT comes from eliminating seeks rather than read time, often through prefetching. Certain traffic patterns affect some policies more, which is why the DT peaks for different policies can differ. In particular, Baleen's peaks occur when prefetching is not beneficial.



(a) Region1

(b) Region7

Figure 7.4: Breakdowns of Disk-head Time at Peaks. Each graph shows the peak 10-min window for that setup. Baleen's Disk-head Time reduction is mostly due to reduced seeks.

Fig 7.4 shows policies' performance at the respective peak windows for Baleen and RejectX. The peak window can differ from policy to policy, as one policy may be good at dealing with a traffic pattern that causes peaks for other policies, but be foiled by a pattern that is handled well by others. This makes optimizing the peak a whack-the-mole game. Baleen's worst time intervals are those in which prefetching is not beneficial. This suggests that a policy wanting to optimize Peak DT could do by being aware of the current load level and able to adapt to it.

7.4 Explicitly optimizing for Peak Disk-head Time

We explored how we could explicitly optimize Peak Disk-head Time. This introduces the additional complexity of scheduling (i.e., when to spend the flash write rate budget) to prioritize admission of items that contribute to the Peak Disk-head Time.

7.4.1 Varying policy selectivity by system load level

Design An online and more flexible method that is not dependent on retraining would be to vary the behavior of the ML policy dynamically in response to load. One approach is to make the admission policy threshold a function of the load factor. This would in effect transfer the write rate budget from off-peak to period periods. Since we seek to optimize Peak Disk-head Time, incurring more misses during off-peak periods to save on flash writes for use during peak periods is a viable strategy for a policy. As eviction ages are around 2 hours, this implies that hits from admitted episodes must happen in less than that time. Thus, we considered it likely that admission decisions will have a positive effect within the peak period, as they are often sustained over multiple hours.

Evaluation We performed an experiment where we modified OPT and only allowed it to admit to the cache during periods of high load (defined as being than 30% of the last peak). We show the results in Fig 7.5, with the modified OPT labeled as "Peak Reduce".

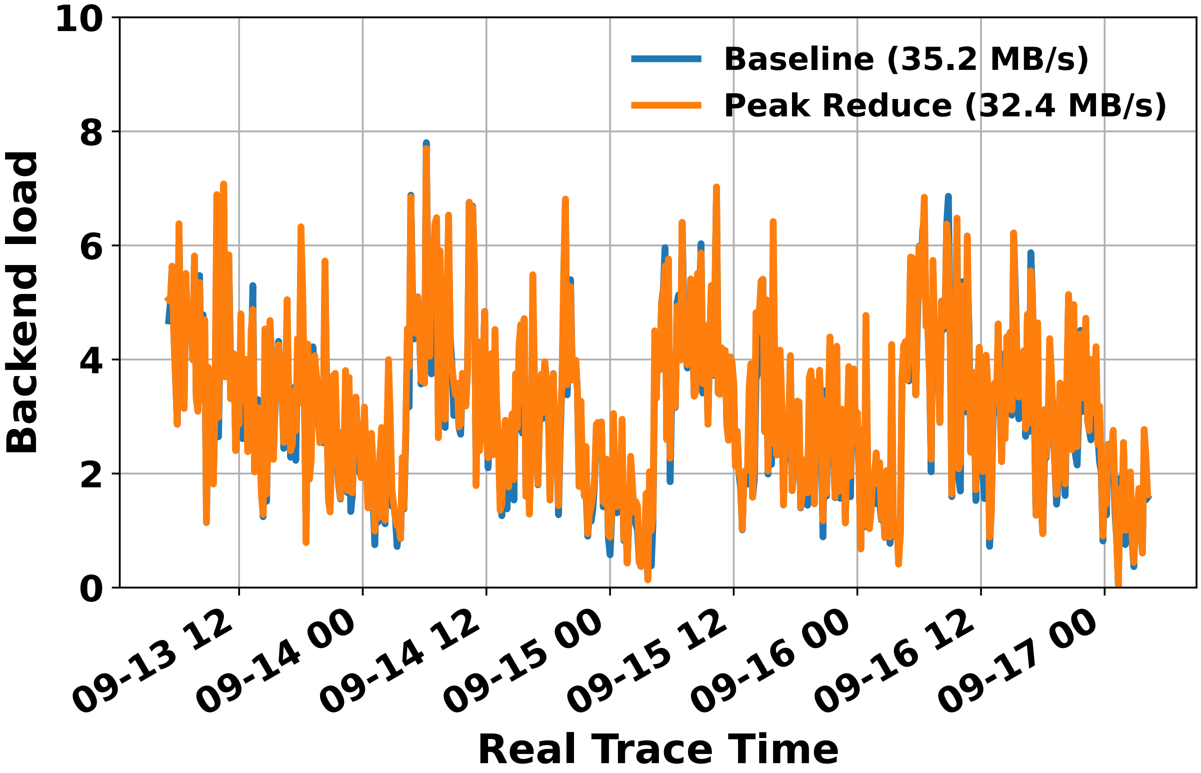


Figure 7.5: Peak reduction by varying policy selectivity in response to load. A dynamic admission policy threshold based on the load level was able to reduce flash writes by 8.0% and Peak DT by 1.4% on Region3. The long-term average flash write rate for the respective policies is included in brackets in the legend.

This reduced Peak Disk-head Time by only 1.4%, but was able to reduce flash writes by 8.0%.

From this, we draw the inference that more fundamental changes (e.g., scoring episodes by their usefulness in reducing Peak DT) are required to optimize explicitly for peak load.

7.4.2 Prioritizing episodes by their contribution to peak load

In this set of experiments, we sought to explicitly prioritize the admission of episodes that would help to bring down peak load.

Design Our design sought to accomplish this by tweaking the training data. In both cases, we identified a fixed ‘peak period’ from a previous simulation run on that trace with Baleen. We experimented with two approaches:

1. **Weigh episodes that overlap with the peak period higher than episodes that do not overlap with the peak period.** The intuition was that the ML might identify common characteristics of items that are cached during periods of peak load and prioritize these items.
2. **Tweak the episode scoring function to account for how much DT was saved during the peak period.**

Evaluation Initial experiments using the analytical model suggested that significant savings were possible, with a modified ‘PeakOPT’ policy reducing Peak Disk-head Time by 29% over vanilla ‘OPT’. However, we could not translate these savings over to simulation.

We modified both OPT and Baleen policies and evaluated both in simulation. The modified OPT policy (which we called OPT-PeakDT) had a regression with a new, much higher peak appearing in a different part of the trace. The modified Baleen policy did not perform better than the vanilla Baleen policy.

7.4.3 Future work

Extend analytical model to peak The current model calculates the average Disk-head Time saved rather than peak. It would be useful to extend it to allow easier calculation of Peak Disk-head Time without requiring a full simulator run. One possible way to approximate this is to divide trace time into smaller intervals (e.g., 10 minutes) and calculating the Disk-head Time saved per interval from each admitted episode. A further improvement would be to permit different assumed eviction ages for each of those smaller sections of the trace.

7.5 Summary

We were able to reduce Peak Disk-head Time by choosing the right parameters (e.g., prefetching method and flash write rate) in conjunction with an underlying admission policy that optimize mean Disk-head Time. However, we were unsuccessful in directly optimizing the peak by modifying the scoring function in the admission policies.

Our efforts suggested that it was harder to avoid regressions than we initially anticipated and we believe that a more complex solution would be required to directly optimize Peak Disk-head Time.

Chapter 8

Workload drift in caching

Caching, at its core, is the problem of predicting what items will be used (i.e., popular) in the future. *Workload drift*, which refers to changes over time in access patterns and the popularity of items, is therefore a threat to the success of a caching policy. Mitigating drift is important to ensure ML-based caches perform well in the long run and not solely at time of deployment or publication.

Our study of caching drift was motivated by the experience of our collaborators at Meta, who found that ML caching policies performed the best at initial deployment and regressed over the years despite periodic retraining (details in §10.1). Moreover, the impact of drift has been corroborated in other applications of ML for large-scale systems, with Microsoft reporting a ML accuracy drop of up to 40% in its network incident routing (NIR) and VM CPU utilization models, with its ML NIR outperformed by a non-ML solution 28% of the time [50].

Our explorations quickly garnered the interest of industry, but a significant limitation was the availability of long-term traces for evaluating drift in caching. Thus, collecting the necessary traces became an important part of this work. The longest public traces prior to this work were 7 days; over a 6-month period, we were able to collect usable data amounting to two 30-day traces.

We found that ML for caching solutions are indeed vulnerable to workload drift. Common drift mitigations adopted by ML-based caches include periodic retraining of models on new data or the adoption of online policies (like reinforcement learning) that continuously update their assumptions with incoming data. We evaluated multiple parameters of (re)training, such as the frequency of retraining and the training window length.

8.1 Background

Unlike computer vision or natural language processing where data has a *stable* sample space, storage access patterns are inherently subject to change over time, whether from configuration changes, data migrations, changing user demand, or system changes [50, 64]. This problem is exacerbated in bulk storage systems which consolidate a multitude of storage workloads and applications and need to meet the needs of the entire spectrum of cloud computing customers.

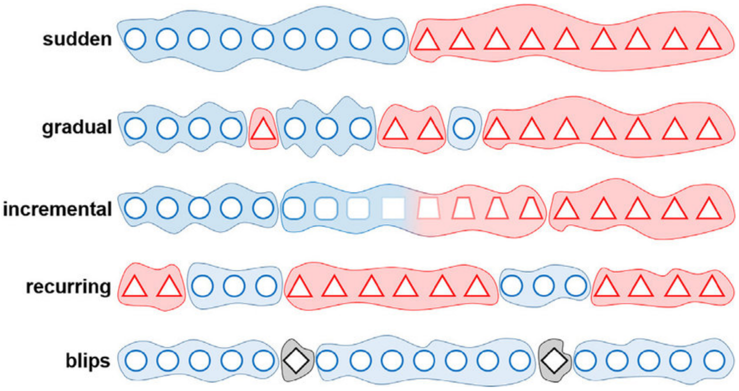


Figure 8.1: Types of drift. Different severity and speeds of drift are shown, including noisy blips, in this figure from [35].

Categorizing drift Workload drift is classified by the ML literature into 1) *covariate drift*, where the data distribution changes, and 2) *concept drift*, where the target concept changes and breaks the assumptions underpinning a model.

Another way to categorize drift is by its cause. We describe three types of drift highlighted in the literature [25, 74, 110] noting these are neither exhaustive nor mutually exclusive [110]. We also illustrate them in Fig 8.1.

- **gradual and incremental drift**, caused by changes in item popularity over time e.g., a slow shift towards video content over photos and text
- **sudden drift**, caused by abrupt (and permanent) changes, e.g., a sharp rise in deployments of large language models
- **recurring drift**, a temporary shift caused by cyclic or irregular phenomena, e.g., Black Friday sales or an election campaign launch

Drift mitigation in the ML literature Many complex solutions exist in the ML literature for dealing with drift [10, 13, 22, 34, 59, 71, 84] and tend to revolve around window-based solutions, change detection, and ensembles. However, they were not designed for real-time use in production systems and have issues such as a high computational cost, the need to wait for ground truth (labels), and not being designed to deal with covariate and concept drift simultaneously [50]. While Matchmaker [50] addresses a number of these concerns in its design (which finds the most similar data batch and uses the matching model for inference), it is hardly a panacea given that it delivered significant gains on only 1 of their 2 case studies, leaving much room for improvement. Matchmaker [50], DriftSurf [71] and AUE [13] are examples of state-of-the-art solutions for drift that would be ideal benchmarks.

Drift in caching ML caching policies are often designed to have models that are trained offline on a trace and then deployed periodically to production [5, 21, 44]. Online methods collect data locally for retraining [96] or do incremental updates with reinforcement learning [8, 18, 87].

While workload drift has been studied in the larger context [25, 31, 71, 74] and in ML for systems [50], its effect on caching is relatively unknown. We know of only one other large-scale caching work that evaluates drift [82], with none explicitly tackling it. In that work, LRU-BaSE, an eviction policy that uses deep reinforcement learning to improve byte miss ratio without worsening object miss ratio, is evaluated on a synthetic drift scenario where two traces are spliced together (corresponding to the sudden drift scenario described earlier in this chapter). Robustness to different workloads is a closely related problem, which has been examined in workload analyses [89] with adaptive solutions proposed for setting cache parameters [19]. Such parameters are often manually set for the workload (e.g., ghost cache size is set to 6 hrs in [5] and 20 hrs in [97]), and may need to be dynamically set to properly cope with workload drift.

Drift mitigation in caching A common mitigation for workload drift is periodic retraining, at the cost of training overhead and complexity for model deployment. If retraining is done on the node itself, this incurs additional CPU and memory overhead to collect training data and train the model. While it is possible to reduce ML overhead with heuristics [69, 88], it would be better to know how often retraining even needs to be done and to reduce the need for retraining in the first place. Further, retraining intervals used in production ML-based caches vary from seconds [88] to 5 minutes [97] to days to months, with no principled way of determining the right interval.

Another simple mitigation for workload drift is to allow a ML policy to fall back on a well-understood heuristic policy, alleviating systems practitioners' fear of performance regressions (e.g., [17] runs a classical policy in parallel with ML and uses the better performing policy). However, this merely limits the downside of ML policies instead of tackling the actual problem and potentially unlocking better ML policy performance.

8.2 Collecting longer traces for analyzing drift

As we are focused on drift as it pertains to caching, we decided to evaluate on datasets derived from cache traces (as opposed to generic datasets as used in [50]). Acquiring the appropriate cache traces for drift analysis was an important and challenging part of this process. We needed traces that were:

1. **Long enough.** The duration of previous public caching traces only went up to 7 days, which made it difficult to distinguish between short-term day-to-day variations and longer term gradual drift. We needed traces that were long enough for gradual drift to be present.
2. **Contained features for ML.** There were other storage traces of appropriate length but which did not have the metadata features needed for a ML policy.
3. **Collected before the flash cache.** Google released two months of storage I/O traces along with the Thesios publication [60], for which we were very grateful. They did contain ML features, but as they were collected post flash cache, many requests would have already been absorbed by the flash cache and were not available for our analysis.

Consequently, we collaborated with Meta to collect additional traces from Jan to June 2024, for which we were most grateful for. We show part of this trace in Fig 8.2. Unfortunately, due to a bug in the key anonymization process that we only discovered later, only two continguous one-month periods of the trace was usable for caching analysis. Notwithstanding this, a 30-day long trace is still four times longer than the longest publicly available trace of 7 days (as per Table 2.6).

Longer traces were collected by using a fixed key anonymization method. Trace dumps were manually collected every month and subsequently stitched into a longer trace.

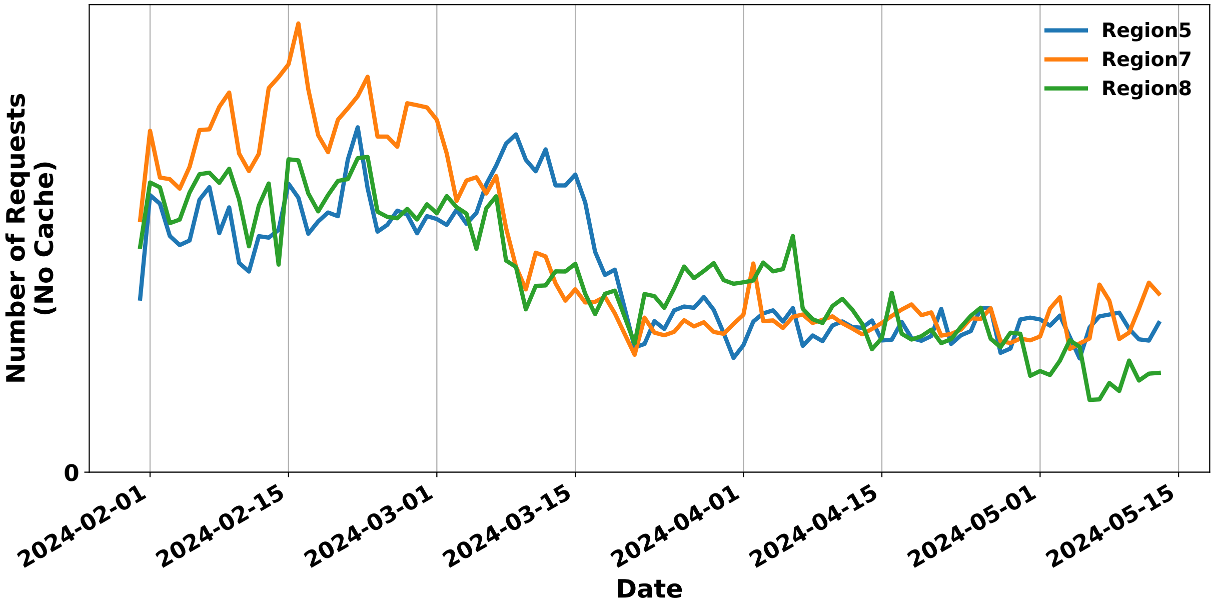


Figure 8.2: Request load level without cache over 3 months. This figure shows the request load level over a period of 3 months without caching.

In the end, we ended up using 3 trace periods for comparison:

1. 7 days ending 2023-March-25 (20230325)
2. 1 month ending 2024-March-02 (20240302)
3. 1 month ending 2024-June-11 (20240611)

We had data for 3 regions (Region5, Region7 and Region8). Region5 and Region7 were present in our 2023 traces, whereas Region8 was new. During our preliminary analysis, we found that Region5 experienced sudden drift in May due to a sharp decrease in incoming load level, which our collaborators confirmed. In our evaluation later in this chapter which is focused on gradual drift, we thus focused on Region7 and Region8.

Block reuse over time We analyzed the longest time that each block is seen in the trace (i.e., the time between the first and last access to the block). We found that a majority of blocks were not seen after 2-4 days, as shown in Fig 8.3.

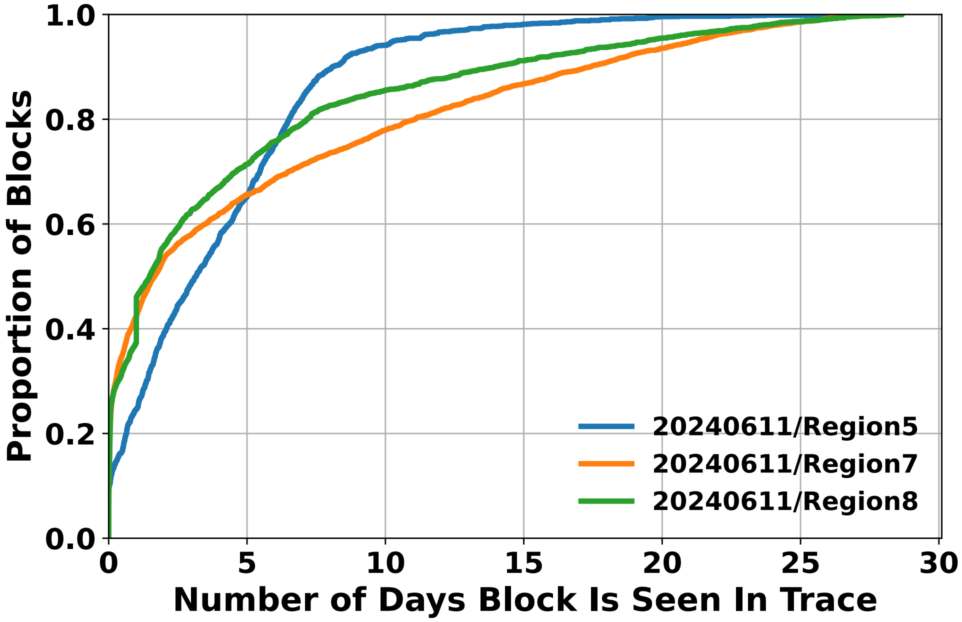


Figure 8.3: Block lifetime. The majority of blocks were seen in the trace for no longer than 2-3 days, indicating a high degree of turnover in the key space.

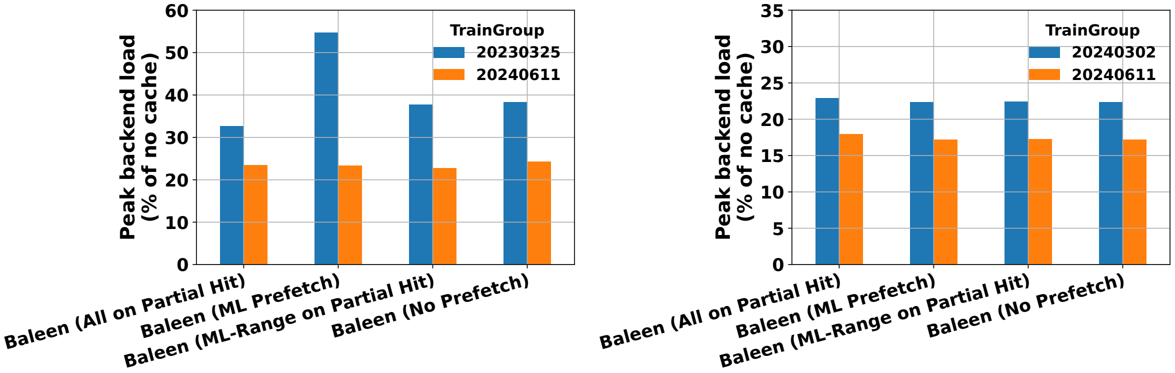
Table 8.1: Statistics of drift datasets

Dataset	Time	Request Rate (s^{-1})	Obj Size, Avg (MB)	Access Size, Avg (MB)	Compulsory miss rate	One-hit-wonder rate	PUT-Only Blocks	#PUT / #Acc
Region5	20230325	364	6.84	2.62	18%	59%	33%	9%
Region5	20240302	326	5.12	3.16	72%	93%	23%	18%
Region5	20240611	21.2	5.81	5.01	57%	86%	52%	43%
Region7	20230325	426	5.71	2.23	17%	62%	38%	12%
Region7	20240302	218	4.52	2.75	83%	95%	20%	18%
Region7	20240611	120	4.61	2.32	22%	53%	29%	19%
Region8	20240302	242	4.82	2.96	69%	92%	29%	22%
Region8	20240611	48.4	5.64	3.55	37%	74%	40%	30%

Trace statistics Table 8.1 shows statistics for the different traces and time periods. We note that the compulsory miss rate appears to have the most variability across time periods, and postulate that it might be useful in designing a measure of drift.

8.3 Evaluating drift across time and clusters

We performed two sets of experiments, where we used 1) a different time period for training or 2) a different cluster for training. In both cases, we evaluated on the respective cluster for the `20240611` period.



(a) Region7, 14.5 months' gap

(b) Region8, 3 months' gap

Figure 8.4: Drift over time decreases ML performance. The increase in peak load is worse with ML prefetching methods (the middle two). The y-axis shows performance on the *TestGroup*. *TrainGroup* refers to the time period used to train the model; the *TestGroup*, which refers to the time period used to test the model, was in both cases `20240611`.

Figure 8.4 shows that training the ML model using an older dataset (in blue) causes a decrease in performance. A larger performance degradation was observed with Region7, where the gap between the training and test period was longer. With Region7, a larger gap is observed when Baleen is used with ML prefetching, with the smallest gap when Baleen is used with static prefetching (All on Partial Hit).

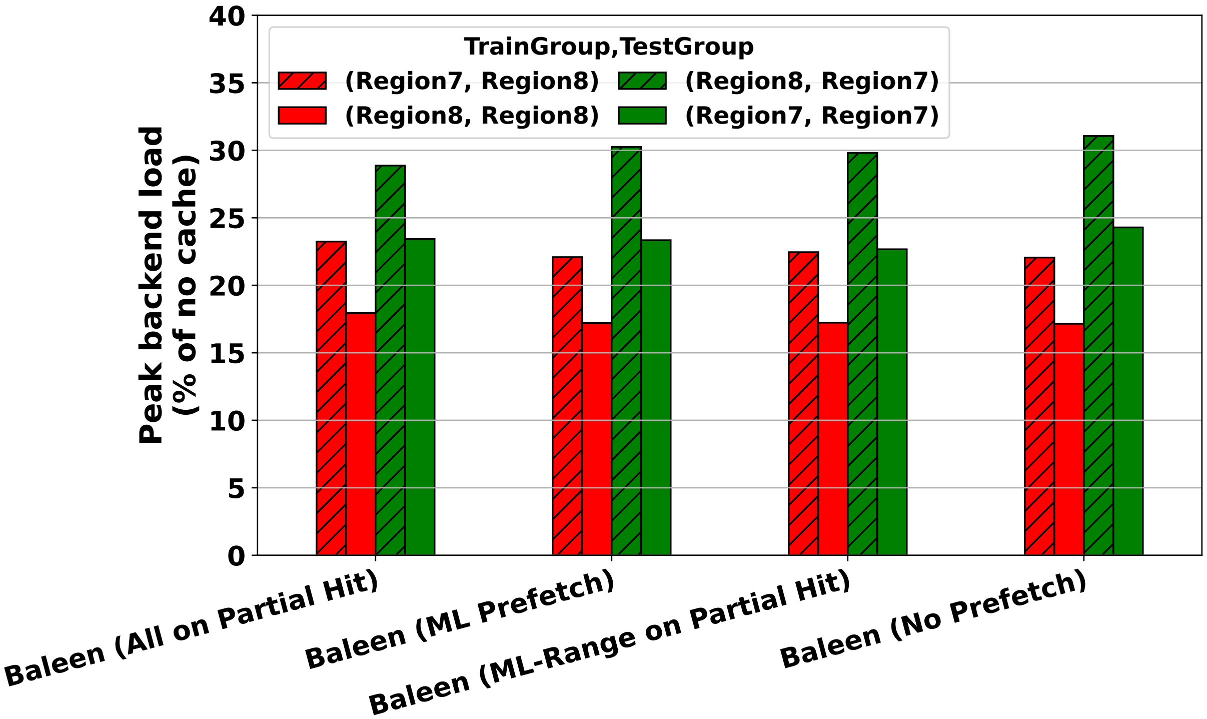


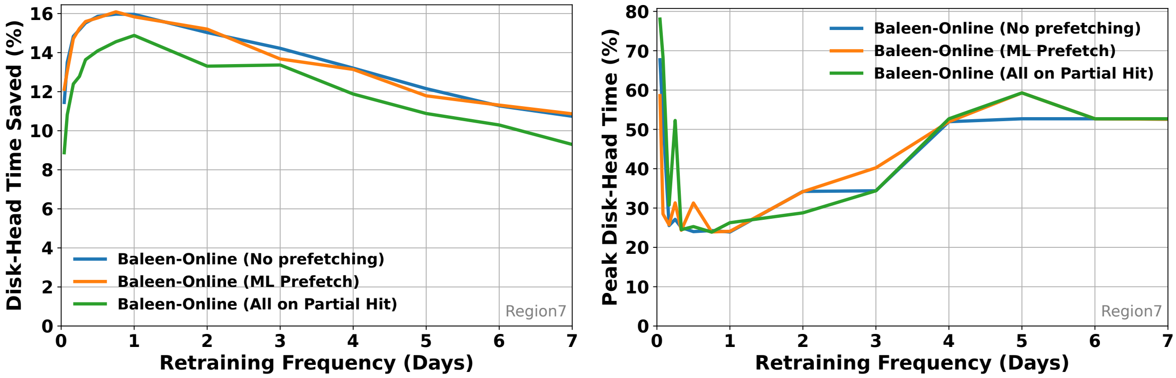
Figure 8.5: Drift: training using a different region decreases performance. The colors denote the trace used at test time, with a shaded pattern if the training trace used is a different cluster than that used for testing.

Figure 8.5 evaluated how performance would be affected if different regions are used for training and testing, which could also be considered a form of synthetic drift (sudden drift) since the change is not across time. Training the ML model using training data from a different cluster (albeit from the same time period) shows a decrease in performance. The drop in performance is worse for methods that use ML prefetching (the middle two) and the least when static prefetching is used (All On Partial Hit).

8.4 Drift mitigation via retraining

To the best of our knowledge, researchers have not explored how long a window should be used to gather training data or how often the model should be retrained.

Frequency of retraining We created an online variant of Baleen (Baleen-Online) which accumulates training data on the fly (and uses RejectX as a fallback option initially until it had accumulated enough training data). We then tested it with different retraining frequencies, with the results shown in Fig 8.6. We found that the sweet spot for retraining interval on our datasets was between 12 to 24 hours, given a fixed training history length of 2 hours. Too frequent or too infrequent retraining caused worse performance. The trend in average Disk-head Time saved in Fig 8.6(a) can also be seen in Peak Disk Head Time in Fig 8.6(b).

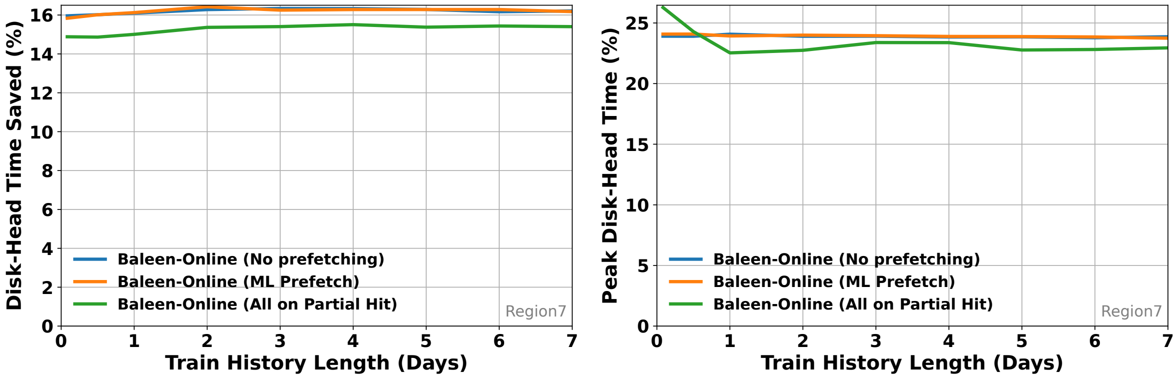


(a) Region7, Average DT saved

(b) Region7, Peak DT

Figure 8.6: Retraining frequency. Baleen-Online performs best with a retraining period of 12 to 24 hours. The results shown use a constant training history length of 2 hours and are evaluated on a 1-month trace from June 2024.

Length of training data We also conducted experiments to vary the length of training data retained to understand how much history is needed for an effective ML policy, and to understand the impact of the length of training data separately from retraining frequency. In Fig 8.7, we show that a longer training data history does not increase the performance of Baleen-Online, and indeed it is relatively insensitive to the training data length when given a fixed retraining frequency of 24 hours.



(a) Region7, Average DT saved

(b) Region7, Peak DT

Figure 8.7: Training history length. Baleen-Online does not require very long training data retention. The results shown use a constant retraining frequency of 24 hours and are evaluated on a 1-month trace from June 2024.

8.5 Future work

A system that optimally mitigates drift should adapt quickly to drift, distinguish noise from drift, and recognize recurring contexts, according to [74]. For the context of flash caching in data centers, we also desire that the system be computationally efficient as it must run in real-time on a data stream with many queries per second and be able to recognize recurring contexts without storing an excessive amount of past data.

Identifying recurring contexts and mitigating recurring drift Instead of using a raw access trace, we will need to devise a compact representation of training data (likely episode-based), and to come up with a similarity metric to aid in identifying recurring contexts. Clustering, such as that used in GL-Cache [91], will be useful to reduce the number of past contexts we need to store.

A simple, first-cut solution would be to implement a drift detector and build in a heuristic policy as a fallback. A more complex solution would involve recognizing recurring drift. This means comparing the current workload to past patterns, which can be done in whole (comparing the current hour with a past hour) or in part after detanglement (comparing the current hour for items with this feature value with a past hour for items with the same feature value). Exploiting recurring contexts has the most potential for improving performance, since unseen data drift (that has not occurred in the past) is unavoidable for all drift mitigation systems [50]. Being able to detect and handle different speeds of change is important. In hyperscalars where flash caches are common, some causes of sudden change (e.g., configuration changes) are caught by site reliability engineers and performance testing, and addressed in a timely manner. Slow drift can be harder to deal with as it creeps up over time, and if the original engineers who implemented a ML caching policy are no longer available to deal with it, the easiest solution for a systems practitioner is to turn the ML policy off and revert to a heuristic baseline. It would be useful for our solution to quantify the different types of covariate drift (e.g., the number of new feature values) and concept drift (e.g., how different the decision tree models are) over time and optionally trigger an alarm to flag the need for human intervention.

Federated learning Being able to operate at the level of the individual node (perhaps via federated learning [31]) instead of requiring central training and coordination would be a bonus.

Feature selection We should also consider adding seasonality features such as day-of-week and hour-of-day. Automated feature selection may also be part of a drift mitigation solution, as features may become less or more useful over time (for instance, the *pipeline* feature was part of an initial ML model but later dropped).

8.6 Summary

In this section, I presented the first substantive study of drift in ML for caching which I evaluated on month-long traces (the second-longest public trace is 7 days long) collected by my Meta collaborators. I developed a online variant of Baleen (Baleen-Online) and offer some insights into retraining, a common drift mitigation method. I show that retraining is useful in preventing large accuracy drops over the timescale of months and that a 12 to 24 hours retraining frequency is optimal for our workloads. While my experiments may seem to differ from the observations of practitioners that motivated this work (drift that hurt ML performance despite retraining), our data was from a much shorter period (months as opposed to years) and could only be collected *after* the period in which they experienced drift that caused them to stop using ML for caching. To further explore the issue of gradual drift across years would require historical caching traces spanning years into the past, which do not exist for the Meta Tectonic system.

Chapter 9

Lessons learned from other ML-guided caching explorations

Baleen is the end result of substantial exploration and experimentation with ML for caching, including negative outcomes from which we drew lessons and see unrealized potential. This section shares and quantifies these lessons.

9.1 ML for flash eviction

We examined how we could use ML for eviction, in tandem with our ML admission and our ML prefetching policies. While having a good admission policy may be instrumental to flash caching performance, using a standard eviction policy (LRU, FIFO) in tandem with it leaves performance on the table. ML for eviction is a well-studied problem compared to admission and prefetching, and we provide more detail on the challenges we faced and the opportunities for future work.

9.1.1 Background

ML has been applied to eviction policies in DRAM caches, while the conventional policy in flash caches has been to use a simple eviction policy (such as LRU or FIFO) and have a complex admission policy. However, flash caches like RIPQ [72] have eviction policies (Segmented-LRU, GDSF) designed to reduce write amplification, indicating that being flash-aware can matter in eviction policies too.

For a simple DRAM cache, Bélády's algorithm [4] (evicting the item with the longest time to next access) is the optimal eviction policy for maximizing hit ratio. This assumption breaks down quickly in real systems: others have shown it is not optimal for flash caching as it does not consider write endurance [16], nor is it optimal for variable size objects [9, 82], optimizing byte miss ratios [82], or variable size caches [41]. Despite these limitations, Bélády's remains the go-to benchmark for eviction policies [99] even though other benchmarks have been proposed to address some of its limitations [9, 16, 41].

A common way to apply ML to eviction is to learn a relaxed version of Bélády. With any ML variant of Bélády, the efficacy of the method hinges upon how well the time to an item's next access can be predicted, which varies from workload to workload. Different ML architectures have been tried including GBMs [68, 88], SVMs [66], LSTMs [40], and RL [86, 87]. Offline methods are trace-based, which often requires separate training infrastructure and may be susceptible to workload drift, while online methods such as RL suffer from a long delay in rewards [6]. Others have proposed different ways to apply ML, such as learning from distribution (e.g., LHD [3]), learning from simple experts (e.g., LeCaR [77], CACHEUS [63]), and group-level learning [91].

While surveying ML eviction policies, we found others encountered problems that Baleen addresses, e.g., the need to prioritize sampling less popular objects around the decision boundary [68] and insufficiency of optimizing for hit ratio alone [82]. They mitigated these problems by applying one-off heuristics that were not adopted by successive works. Applying the episodes model could improve the training process for predicting time to the next access. We can also exploit characteristics of flash caching workloads (e.g., focus on throughput over latency, block-segment structure) and apply ML to eviction in other ways, such as classifying items into different eviction queues.

9.1.2 Analytical model showed potential benefits of early eviction in reducing cache space needed

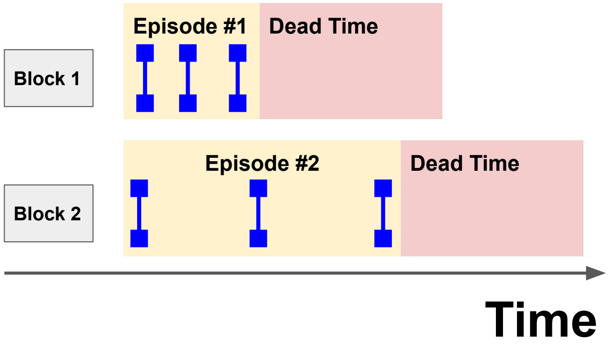


Figure 9.1: Dead Time in episodes. The dead time of an episode is the logical time between the last access in an episode and the time it is evicted. This is equal to the cache's eviction age. For LRU, the expected dead time for each item is constant. A longer eviction age is necessary to get all hits in Episode 2, but not Episode 1. Episode 1 would have less dead time while still getting all hits with a much lower eviction age.

Dead time accounts for 50% of time in cache. If items could be evicted immediately after they are last accessed in an episode, instead of waiting to leave the cache, this would decrease dead time to zero and result in a greater effective cache size. Episode-based analysis showed mean Disk-head Time could be reduced by up to 11%, showing that there is still significant room for improvement in eviction policies.

9.1.3 Improving eviction by using ML to predict episode properties

Using ML to predict episode timespan. The most useful episode property to predict would be episode timespan, the time between the first and last access in an episode. An eviction policy could evict the item after the predicted time passes. If perfectly accurate, this would reduce dead time to zero. Moreover, this could also be used to group together things that will be evicted at the same time to lower flash write amplification. We trained ML regression models to predict episode timespan, but found ML model accuracy to be low, as shown by the low R^2 score in Table 9.1. Thus, we sought to evaluate other episode properties.

Using ML to predict maximum interarrival time for a TTL-based (time-to-live) eviction policy. The **maximum interarrival time** of an episode is the largest of the interarrival times between accesses. We found that this could be predicted much better than episode timespan, as shown in Table 9.1. In a typical LRU eviction policy, the dead time is the cache eviction age. The eviction age of a cache is necessarily larger than the maximum interarrival time of all episodes. If we knew the maximum episode interarrival time, we could evict items before the full eviction age, thus reducing dead time and cache space needed.

Fig 9.2 shows the cumulative write rate of episodes for increasing maximum interarrival times, indicating that the majority of episodes have a maximum interarrival time of 10 minutes or less, meaning that the many items could be evicted early 10 minutes after the last access, instead of having to wait for the full eviction age of the cache (e.g., 2 hours). Otherwise, for many of these items that have a few accesses in quick succession, the time spent in cache is dominated by the 2-hour eviction age (and thus dead time) rather than the useful timespan which may only be a few minutes.

Metric	R^2 **score**
Timespan (logical)	0.34
Timespan (wall-clock)	0.36
Max Interarrival Time (wall-clock)	0.52

Table 9.1: Predicting different targets for ML eviction We evaluated different episode properties for use as intermediate targets in ML eviction policies. Timespan refers to the time between the first and last access in an episode, while the maximum interarrival time refers to maximum time between any two consecutive accesses in the episode. Logical time refers to time in logical timestamps (the order of requests seen by the cache). R^2 (coefficient of determination) is a regression metric measuring the goodness of fit, with random noise having a value of 0 and a perfect fit having a value of 1.

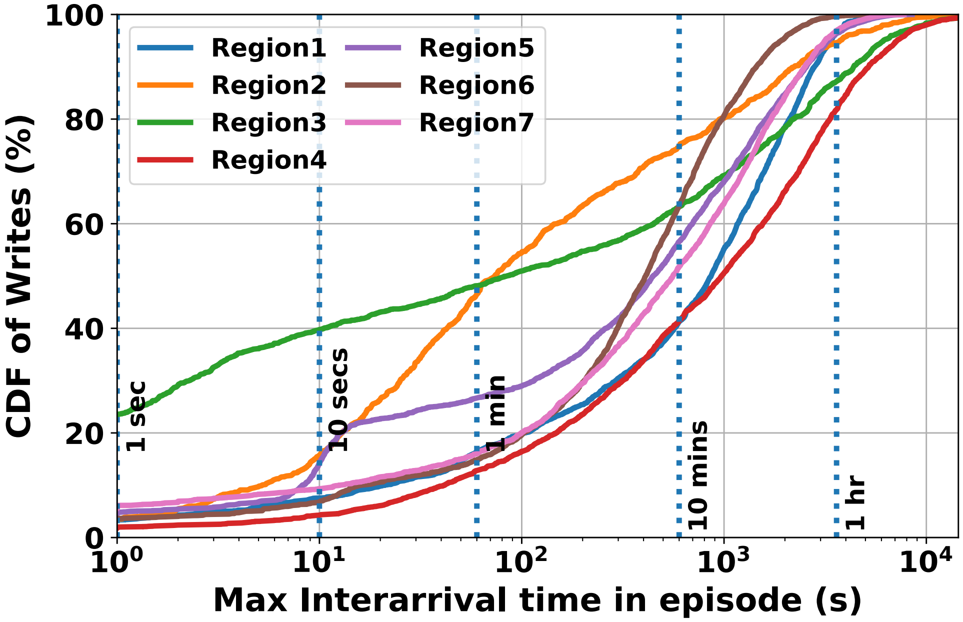


Figure 9.2: Maximum interarrival times for episodes. For example, episodes with interarrival times no larger than 10 seconds account for 40% of writes in OPT-admitted episodes for Region3.

We implemented a TTL (time-to-live) cache where each cache item has an associated expiry time, which is extended with every hit. If the time since the last access exceeds the TTL, the item is considered as available for eviction. We then evaluated an extension of OPT called *OPT-TTL*, where TTL is set to the maximum interarrival time within each episode. This saved 2% of Disk-head Time for Region7, as shown in Fig 9.3.

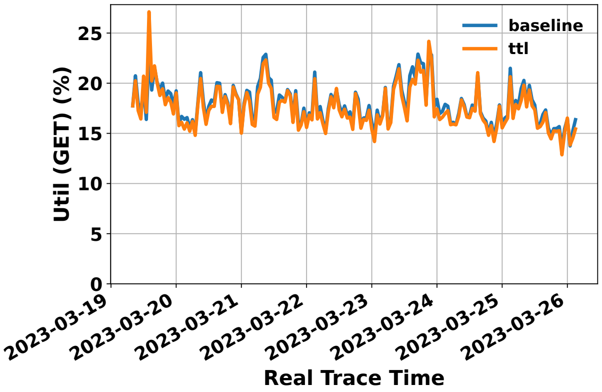


Figure 9.3: OPT-TTL: An optimal TTL-based eviction policy with early eviction. This saved only 2% of Disk-head Time for Region7, indicating that there remains work to be done.

Additional work remains to be done to explore how the amount of Disk-head Time savings can be increased, and to translate the policy from OPT to ML (and deal with the accompanying drop as we go from potential to actual savings).

In terms of feasibility of implementation, CacheLib supports efficient TTL eviction in DRAM with Meta employing it successfully in key-value stores where the TTL is set manually for each use case. However, additional work is required to adapt it for flash (without excessive write amplification) and to predict an appropriate TTL.

9.1.4 Future work

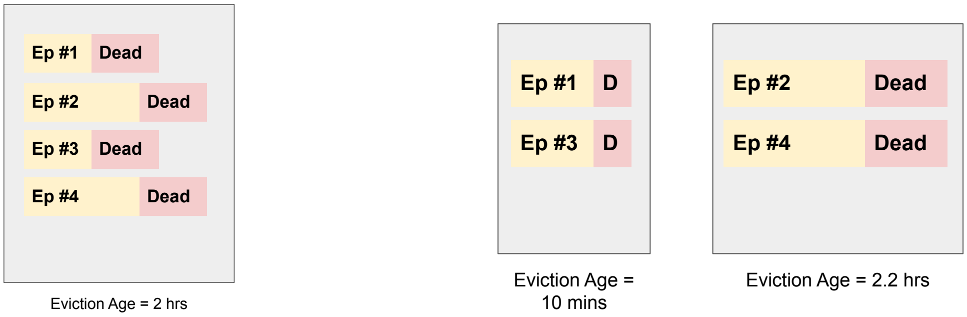
Improving on TTL-based eviction Here, we list additional improvements that could be implemented to improve on the current TTL-based eviction policy.

1. **Adjust the TTL prediction on successive hits.** A further extension would be to continue adjusting the TTL value as hits (and information) arrive, and not just set it once at insertion. For instance, a possible strategy could be to adjust the TTL prediction based on observed interarrival times after the item is inserted. (In a queue-based implementation, this could be realized by having allowing evictions from one queue to be inserted into another queue, not unlike what is described in the Multi-Queue policy for second level buffer caches [108], although care needs to be taken to avoid additional flash writes in the process.)
2. **Predict variance of interarrival times for episodes** Episodes could be grouped by their predicted maximum interarrival time plus a constant ($MaxIA + c$), where c corresponds to the variance of interarrival times. The access patterns of blocks vary in predictability, ranging from those with short, deterministic timespans/interarrival times to access patterns that are essentially memory-less and Poisson-like. Determining where an episode falls on this spectrum can be helpful in adjusting the eviction strategy (and potentially admission as well).

Future work could also include a full evaluation with smaller cache sizes (to see if the same Peak Disk-head Time can be achieved with smaller caches), as well as comparisons with state-of-the-art eviction policies such as LRB [68] and TinyLFU [18].

Using multiple FIFO or LRU queues Another avenue of work would be to apply ML to a setup with multiple LRU or FIFO queues, and to determine the optimal queue sizes for those setups. Recently, cache policies centered on FIFO queues have been shown to work well in production DRAM caches [92, 94, 100].

Having multiple LRU queues would allow for differentiated eviction while being simpler to implement than a TTL cache, with less overhead. If we knew the maximum interarrival time of episodes, we could group insertions into different queues by their maximum interarrival time, reducing dead time and making better use of the cache space. Episodes could be grouped by their (predicted) maximum interarrival times (e.g., 0-10 seconds, 10-60 seconds, 1-10 minutes, 10-60 mins, above 60 minutes).



(a) Single queues

(b) Multiple queues

Figure 9.4: Opportunity for differentiated eviction strategies (multiple queues).

For example, in Fig 9.1, Episode 1 has a much lower maximum interarrival time than Episode 2. With a single LRU queue (Fig 9.4a, the eviction age is the same for both episodes and there is more dead time. Each episode remains intact as long as the eviction age is more than its maximum interarrival time. Thus, Episode 1 can be placed in a queue with a much lower eviction age without loss of hits compared to for Episode 2. By placing these episodes in separate queues (Fig 9.4b), the dead time for Episode 1 will be reduced, making the cache more efficient in its use of space. This efficiency increase is reflected in the slightly higher eviction age for the second queue in Fig 9.4b.

The amount of cache space to allocate to each queue could determined offline using our episodes model based on the expected size footprint of admitted items from each group (i.e., by reading off the y-axis for the intersections between the dotted vertical lines and each workload's CDF in Fig 9.2.)

9.1.5 Summary

In summary, our ML for eviction efforts showed us the difficulty of predicting episode timespan, while showing that there was potential for a TTL-based eviction policy which predicts the maximum interarrival time of episodes.

9.2 ML for DRAM placement to reduce writes

The design of hybrid caches and how to best use a mix of DRAM and flash is a recurring topic of interest. Conventional wisdom is to promote an object to the higher level in a memory hierarchy (i.e., DRAM) when there is a hit on an object in a lower level (i.e., flash). This was designed with latency in mind, but does not help in reducing flash writes (see §9.7). If an item is present simultaneously in DRAM and flash, it means wasted space.

We found that a small DRAM cache used in this conventional manner does not contribute significantly in improving end-to-end caching metrics. Thus, instead of retaining popular items in DRAM, we sought to explore if DRAM could be used to play a very different function within our hybrid cache setup: reduce flash writes.

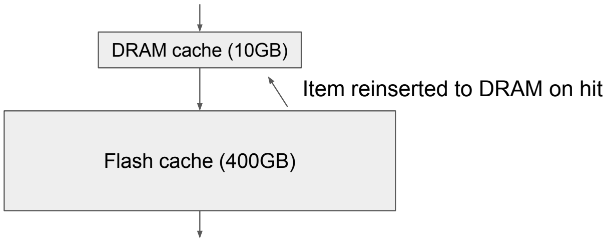


Figure 9.5: Use of DRAM in hybrid caches. DRAM is typically used as a small cache in front [5] or behind [97] a larger flash cache. All items must pass through DRAM, meaning that popular items (that will be admitted to flash anyway) waste space in the DRAM cache.

9.2.1 Background

In hybrid caches, heterogeneous storage mediums are used to meet design goals. HDDs are most cost-effective for capacity and flash drives are more cost-effective for IOs than HDDs. DRAM has higher throughput per byte than flash, but Tectonic-Shift [103] found that the bandwidth of DRAM-only storage nodes is in practice limited by NIC throughput (at 100 Gbps) and that flash drives would have the same bandwidth per watt as DRAM.

Flashield [21] admits every item to DRAM first and uses the number of reads and writes to determine which items are more suited for flash, while treating DRAM and flash as a single memory pool for eviction (CLOCK, a LRU approximation, is used). In production systems, DRAM has been used as an additional buffer in front of [5] or behind [97] a larger flash cache. The high flash-to-DRAM ratio of production caches (1:40 in Meta's Tectonic [26]) means that the effect of DRAM on end-to-end systems metrics are limited and that DRAM lifetimes are too short for Flashield (which was designed for a 1:7 ratio) to be effective [5]. (More details are provided in §5.4.2.)

Thus, we assert that a better way to utilize a small amount of DRAM would be to use it selectively and judiciously to reduce flash writes instead of letting *every item pass through DRAM*. This is a fundamental difference with other systems that also use DRAM as a filter [5, 21]. In Tectonic, the DRAM available for caching is 1/40 that of flash [26], meaning that using DRAM to extend the cache size or even to collect features for later admission has minimal impact. When we simulated a DRAM cache in front of the flash cache, a DRAM eviction age of 10 seconds to 1 minute was typical.

9.2.2 Evaluation

We proposed and evaluated multiple approaches, all of which revolved around admitting only a fraction of items to DRAM and ensuring that they are retained in DRAM for only a short time before being evicted.

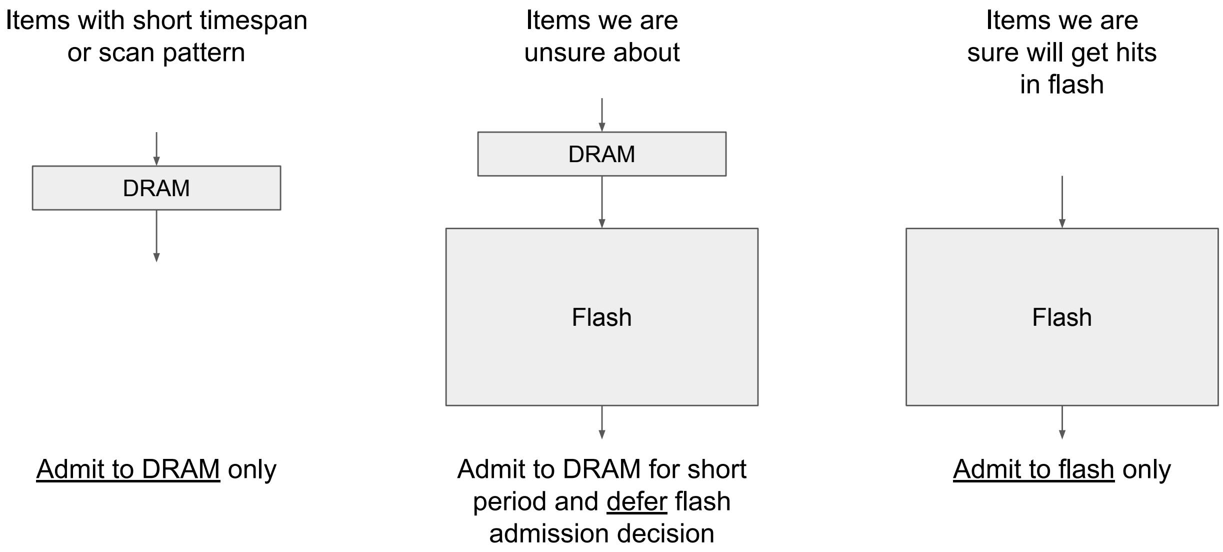

Items with short timespan or scan pattern
DRAM
Admit to DRAM only
Items we are unsure about
DRAM
Flash
Admit to DRAM for short period and defer flash admission decision
Items we are sure will get hits in flash
Flash
Admit to flash only

Figure 9.6: Proposed use of DRAM in hybrid caches. We proposed that items be classified into three groups: those that should be admitted to DRAM only, those that we expect to gain more knowledge on within their DRAM lifetime, and those that should be admitted to flash only, bypassing DRAM.

9.2.3 Using DRAM to gain more information on episodes before deciding

Design Here, we use DRAM to gain more information and figure out which items are worth admitting into flash. For items in this category, upon a hit (or more, if necessary to make the decision to admit), they would be evicted from DRAM and inserted into DRAM.

With a perfect admission policy, items would be admitted into flash directly. Real policies make mistakes: false positives (causing wasted writes) and false negatives (lost opportunities to reduce backend load). 45% of flash writes are wasted for Baleen (i.e., do not receive a hit after being admitted) which is double that of OPT's 20%, showing that flash writes could be reduced by up to 25% with better ML admission. Moreover, items that share the same metadata features cannot be distinguished the first time they are seen by a ML admission policy. Admitting such items into DRAM provisionally for observation gives the cache additional information to distinguish whether they will have further reuse.

While systems such as Flashield [21] may also use DRAM as a filter, the key difference is that they let *every* item pass through DRAM. This results in DRAM lifetimes that are too short to collect the information required. To mitigate this, we propose to reduce both the number of unique items admitted to DRAM and to cap the DRAM lifetimes of items.

First, apply a filter to select items for DRAM, such as by selecting items that are within the 10% of items closest to the admission threshold cut-off, or by training a ML model to identify those items that would benefit (e.g., items that are indistinguishable at first access but have subsequent follow-up accesses that enable them to be separated from the duds). Second, use a FIFO-like policy instead of LRU to avoid popular items hogging DRAM. Third, add a heuristic to quickly evict items to flash items that have been shown to be worth admitting to flash.

We are not the first to suggest using a FIFO policy for modern caches, with others having compared the trade-offs of FIFO and LRU [23, 93].

Results We assessed the feasibility of this approach by looking at the information gained about an item during its DRAM lifetime. We found that 37.2% of OPT-admitted episodes (equal to 45.7% of write budget) have a second access within 10s of the first, while 26.7% of OPT-rejected episodes have a second access within 10s of the first. If there is a second access within 10s of the first, the probability of the episode being an OPT-admitted episode is 23.5%. This suggests that the threshold for admission to flash may need to be higher than 1 hit within 10 seconds in order for precision to be acceptable and avoid excessive false positives.

However, we realized that the DRAM to flash ratio is too skewed in our setup for this to work well. Most items never receive enough DRAM hits during the time needed and the features are not discriminating enough to be able to filter enough episodes out to the degree required to raise eviction age to the length required to get the amount of data required.

In future work, we would want to evaluate this approach with much larger RAM caches to see if that would make this approach feasible.

9.2.4 Admit episodes with a very short timespan directly to DRAM

Design These episodes are short enough to be stored in DRAM for their entire lifetime and bypass flash entirely. Since their useful time in cache is so short, their eviction age under a typical policy like LRU would dwarf the time between the first and last hit and thus these episodes would have a high dead time ratio.

In choosing admitted episodes to place in DRAM instead of flash, we prioritize episodes with the short time in cache regardless of their size to maximize the benefit-cost ratio (as flash writes and space taken up in DRAM are both proportional to the episode size and thus cancel out), we aim to admit episodes with the shortest time in cache regardless of their size in order to maximize the benefit-cost ratio:

$$\frac{Benefit}{Cost} = \frac{WritesSaved}{SpaceInDRAM \times TimeInDRAM} = \frac{NoOfSegments}{NoOfSegments \times TimeInCache} = \frac{1}{TimeInCache}$$

A policy like FIFO rather than LRU makes sense here, given that we expect these episodes to have a short timespan. We can train a ML model to identify episodes with a short timespan, with this model being run on the first miss.

Results Using the analytical model, we assessed the potential benefit from this approach. Consider a DRAM eviction age of 10 seconds. 11.3% of OPT-admitted episodes have interarrivals no longer than 10 seconds (from Fig 9.2). If we can identify them and admit them only into DRAM, we could save 7.8% of flash writes. If we are able to increase DRAM eviction age to 1-min, those numbers increase to 15.6% of OPT-admitted episodes and savings of up to 11.0% in writes.

However, further progress on our workloads was stymied by the lack of success in developing an accurate model to predict episode timespan, as explained in §9.1.3. However, this could be worth revisiting on other workloads or feature sets that enable accurate prediction of episode timespan.

9.2.5 Future work

Avoid flash entirely for blocks with a scan and churn pattern. These episodes may have long timespans at the block level, but at the segment-level have no reuse during their cache lifetime and thus the useful timespan of each segment is zero. Scan and churn patterns are present in most workloads and known to be challenging for LRU and LFU caches [63]. Scan patterns are those in which items are accessed exactly once (often in sequence), while churn workloads have working sets larger than the cache size potentially causing the cache to lose older valuable items. Examples for both are shown in Fig 9.7.

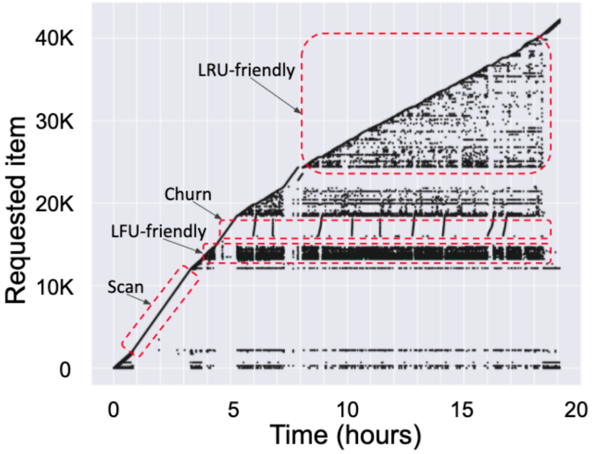


Figure 9.7: Scan and churn workloads. Scan, churn, LRU-friendly and LFU-friendly workloads from the FIU block I/O traces are shown in this figure from [63].

An example of a scan pattern is a block being accessed segment-by-segment in sequence, with no actual reuse of any segment. In churn workloads, items are equally likely to be accessed and not necessarily in sequence.

Scan episodes in particular are good candidates for prefetching and admission, but should be prioritized for placement in DRAM and fast eviction after a single hit as there is no reuse. If churn workloads have too large a working set and too poor spatial locality to benefit from prefetching, they should be rejected from both flash and DRAM.

A ML model could be trained to identify blocks with a scan pattern and then admit them directly to DRAM, bypassing flash.

Adding features to coordinate eviction with prefetching Adding the prefetcher's decision as a decision would be a promising extension, since we expect prefetching may be positively correlated with both. We also expect this to increase the net benefits from prefetching, since, for instance, coordinating with the prefetcher to default to faster eviction or DRAM placement for prefetched segments would help to reduce the cost of false positives in prefetching.

9.3 More advanced models: Cache Transformer

GBMs are relatively simple and thus we also implemented more complex ML models for learning cache access patterns. Specifically, we add two deep models used to learn sequences in natural language processing:

Baseline: MLP feedforward A basic multilayer perceptron (MLP) feedforward model that takes the same features as our GBM model, i.e., only features from the current access, with a single hidden layer of size 80.

Cache Transformer architecture A Transformer [75] encoder that uses features from the prior h ($h = 16$) accesses in addition to the current access. Instead of sequences of words, it uses sequences of accesses. We describe the details in the section below.

9.3.1 Neural Architecture of Cache Transformer

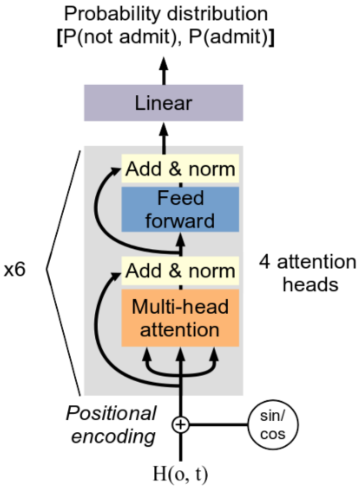


Figure 9.8: Cache Transformer architecture.

As shown in Fig 9.8, the Cache Transformer architecture consists of a series of Transformer encoders stacked together, with a linear classifier at the end. Before being passed to the first encoder, the windows are normalized and a sinusoidal positional encoding is applied. The encoders serve the purpose of learning and evaluating the self-attention between different accesses in the window. After the windows are passed through all the encoders, a final linear layer maps the last encoder's output to the model's prediction, which is represented as a probability distribution.

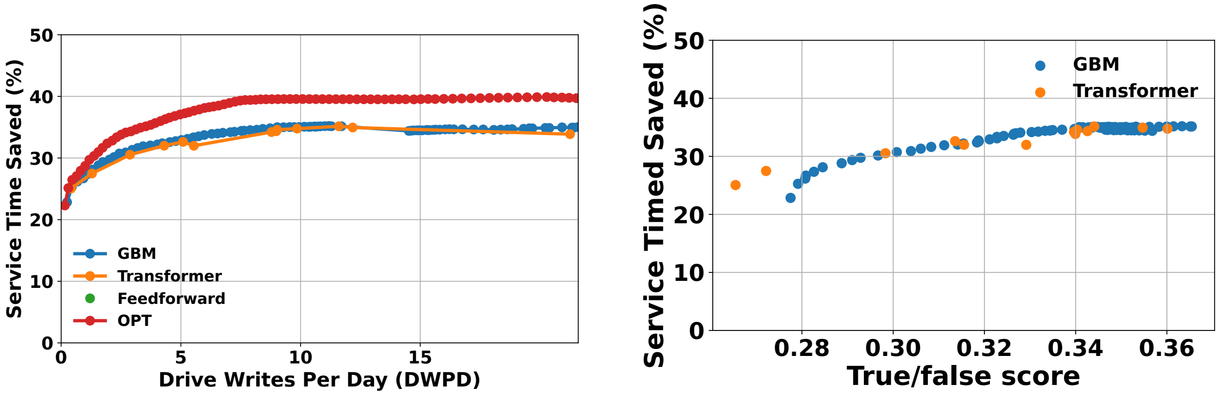
First, the model passes the sequence through a sinusoidal positional encoding to inject relative position information. Then, the encoded sequence is passed through 6 encoders with 4 attention heads each, followed by a linear layer that maps to a similar binary probability distribution to the MLP feedforward model.

9.3.2 Training setup

Neural network models such as the Transformer used PyTorch for training and prediction. When training the Transformer neural network models, positive training examples are upsampled to balance out the classes and reduce the tendency to overfit. The MLP used for comparison had one 80-size hidden layer. Neural network training was done using RaySGD on a cluster with 8 Nvidia GeForce Titan X GPUs.

9.3.3 Evaluation

We found that GBM performs best (0.2% better than Cache Transformer), despite only having features for the current access. This was contrary to our hypothesis that more historical information and access to the pattern of accesses would help model performance.



(a) Disk-head Time Saved against Write Rate.

(b) Disk-head Time Saved against true/false linear regression score (R^2). R^2 goes from 0 (random noise) to 1 (perfect fit).

Figure 9.9: Different architectures for ML admission. GBM is the best non-OPT policy. A 10%-trace was used. Mean DT is reported here, relative to no cache. Service Time is the old name for Disk-head Time.

Table 9.2: Performance of different models, online and offline. h denotes the number of past accesses used as input into the model. Write rate and IO hit rate are from online simulations.

Model (h, history)	**Loss**	**Offline accuracy**	**Online accuracy**	**Write Rate**	**IO hit rate**
MLP feedforward (h = 1)	0.41	90.2%	88.5%	28.1 MB/s	48.1% (-8.6%)
Transformer (h = 16)	0.18	92.6%	89.5%	42.9 MB/s	49.3% (-6.5%)
GBM (h = 1)	-	93.8%	91.1%	37.9 MB/s	49.4% (-6.3%)
OPT	-	100%	100%	30.4 MB/s	52.7%

Delving deeper, we show the precision and recall for the different methods in Table 9.3. We observe that GBM produces the highest F1-score, i.e., it balances recall and precision the best.

The MLP has the highest precision at the expense of recall. We observe that amongst the ML models, the trend in write rates corresponds with the trend in precision.

Table 9.3: Recall, precision, and F_1 score of models. GBM has the best F_1 score amongst non-OPT policies.

Model	Precision (%)	Recall (%)	F_1 score
MLP feedforward ($h = 1$)	85.6	35.5	0.502
Transformer ($h = 16$)	66.7	50.7	0.576
GBM ($h = 1$)	76.8	51.9	0.619
OPT	100	100	1

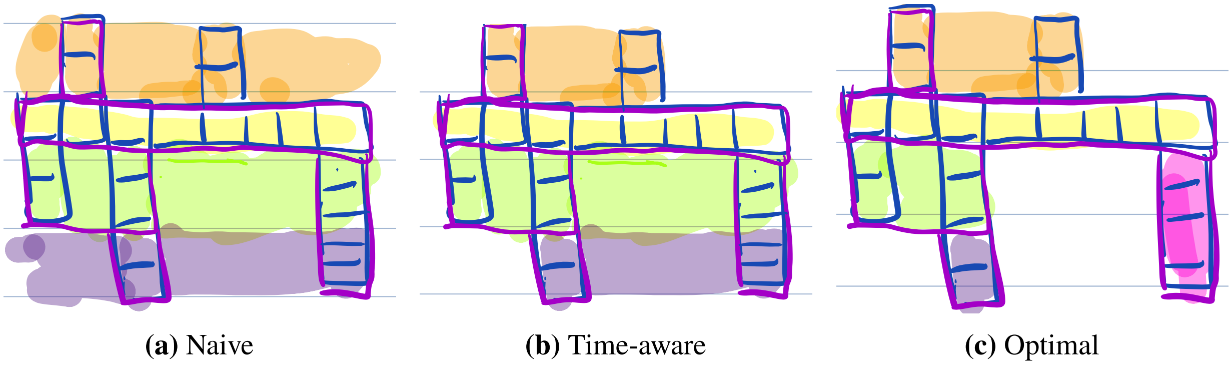
9.3.4 Summary

Although we cannot dismiss the possibility that the Cache Transformer model is held back by our training process, a challenge we struggled with was the highly imbalanced classes. GBMs are known to be robust and work out of the box on many datasets. Baleen hence uses GBM given that it performs best and is the most efficient of the options explored.

9.4 Segment-aware admission

Segment-aware admission's potential Baleen operates at the block level and can only choose to admit or reject the entire access range, rather than individual segments (unlike RejectX). This approximation results in performance being left on the table, versus admitting only what is actually needed. We speculate that some of RejectX's strong performance as a baseline is due to its ability to admit only part of a block.

Extending episode model for segment-awareness Segment-aware admission allows the policy to admit only part of an episode, i.e., a sub-episode. A **sub-episode** is a consecutive range of segments from an episode, that are always accessed at the same time. There are four sub-episodes in Fig 9.10a and 9.10b, represented by the four colors. The cost of these sub-episodes are the number of segments written (y-axis), whereas the benefit (in hits) depends on the exact combination of sub-episodes admitted (see Table 9.4).



(a) Naive

(b) Time-aware

(c) Optimal

Figure 9.10: Segmentaware admission modeled using sub-episodes 9.10a represents a naive segment-aware model, 9.10b represents a time-aware model that is more accurate in modeling cache space and (c) represents a model that allows admissions for the same segment twice in one episode. 9.10c is complex to implement as it must model re-admissions and segments falling out during the episode. Sub-episodes present are represented by the different colors. The y-axis is the segment offset, and the x-axis is time. Each column represents one access or I/O. The cost of these sub-episodes are the number of segments written (y-axis), whereas the benefit (in hits) depends on the exact combination of sub-episodes admitted. See Table 9.4 for examples.

Table 9.4: Costs and benefits are illustrated for sample decisions. These are with reference to the sub-episodes presented in Fig 9.10a and Fig 9.10b. EA stands for Eviction Age.

Admit	Hits (IOPS)	Hits (Segments)	Segments	Useful time in cache	Total time in cache ($\propto \frac{1}{\text{cache size}}$)
{orange}	0	4	2	2 * 7 (episode timespan) = 14	14 + 2 * EA
{yellow}	3	7	1	1 * 7 = 7	7 + 1 * EA
{orange, yellow}	5	11	3	3 * 7 = 21	21 + 3 * EA

Figure 9.10 shows 3 different ways of modeling segment-aware admission:

1. by splitting only along the segment dimension (1D bin-packing),
2. by splitting along both the segment and time dimension (improved version of 1D bin-packing),
3. allowing further splits along the time dimension (2D bin-packing).

Together, Figure 9.10 and Table 9.4 illustrate the decision space that a segment-aware policy must consider.

Making our analytical model and simulators segment-aware added another level of complexity as they now had to keep track of which of the 64 segments were being admitted. Our revised segment-aware analytical model estimated a potential reduction of Disk-head Time by 11%.

To get decisions for an online policy, we posed the problem of segment-aware admission as a bin-packing problem. We developed a greedy solution to the fractional knapsack version of this problem and ran it in simulation to give an upper bound of the benefit attainable. Further, we developed a greedy heuristic algorithm shown in Algorithm 1 to determine admission decisions.

Algorithm 1 Greedy segmentaware admission policy

```
Require: PQ: Priority queue weighted by $\frac{benefit}{cost}$
  // Generate all possible episodes and put into priority queue
  for each block do
      for each episode do
          for each unique segment range do
              // Create a sub-episode
              $Cost \leftarrow$ number of flash writes (segments to be written)
              $Benefit \leftarrow$ hits (excluding first access in each episode)
              PQ.insert($SubEpisode$(Cost, Benefit))
              EpisdesByBlock.append(SubEpisodeId)
  // Run greedy algorithm to iteratively pick the best SubEpisodes to admit
  $WritesRemaining \leftarrow WriteBudget$
  while $WritesRemaining > 0$ do
      $Eps \leftarrow$ PQ.pop()                          ▷ Pick episode with highest benefit/cost
      $JustWritten \leftarrow []$
      for Segment in Eps.SegmentRange do
          if IsWritten[Block][Segment] is false then        ▷ Mark segments as written
              $IsWritten[Block][Segment] \leftarrow True$
              $WritesRemaining \leftarrow WritesRemaining - 1$
              $JustWritten.append(Segment)$
      for each OtherEps that overlaps with Eps.SegmentRange do
          // Update Marginal Cost
          $OtherEps.Cost \leftarrow OtherEps.Cost - \sum_{s \in Eps.SegmentRange, s \in JustWritten} 1$
```

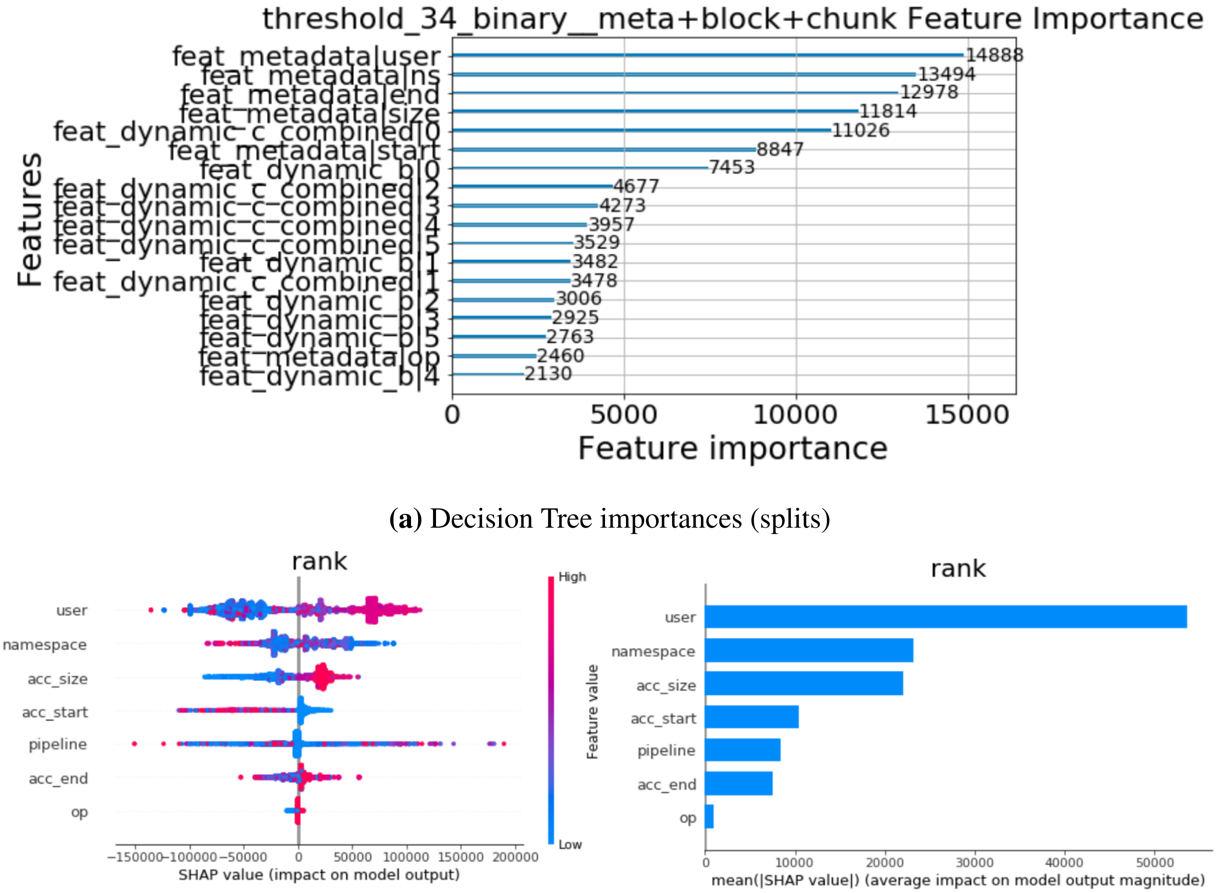
At the time, we were deciding between prioritizing development on segment-aware admission versus getting prefetching to work. Our results from the fractional algorithm showed that the potential gains from prefetching had far more impact than even a perfect segment-aware policy could have. The impact of prefetching turned out to be substantial, as we discussed during in the chapter of Baleen.

Summary Our judgment call is that segment-awareness did not yield sufficient benefits to justify including it in Baleen and adding significant complexity (and source of bugs), but we would not rule out the possibility of a different trade-off being made for other bulk storage systems with different workload characteristics.

9.5 Benefit attribution for Baleen and quantifying gap to OPT

Understanding how Baleen derives its benefits over other policies is essential for other practitioners to understand if the gains are also applicable to their own systems.

Baleen's metadata features are essential to its performance We attempted to assess the importance of Baleen's features in 3 ways, in decreasing order of convenience: 1) Decision tree importances, 2) SHAP (SHapley Additive exPlanations) values, and 3) ablation studies in which we evaluated Baleen end-to-end using a subset of features for training. Our conclusion was that there is no substitute for ablation studies, especially when the relationship between ML model accuracy and end-to-end system performance (in Disk-head Time) is a noisy one. That being said, decision tree importances are cheap to compute and are still helpful as a quick sanity check. Decision-tree classifiers (including the Gradient Boosting Machines we use) offer feature importances for free, most commonly by counting the number of times each feature is used in a split during the model building process.



(a) Decision Tree importances (splits)

(b) SHAP values

Figure 9.11: Examples of feature importance methods

Using ablation studies, we determined that the metadata features (user, namespace, op) were the most useful, followed by the size-related features (start, end, size) and then the dynamic usage-based features. We also evaluated additional features such as the shard but found they did not improve performance by any appreciable amount.

Size-awareness is essential and Baleen learns it implicitly. In early experiments with ML admission, we took an existing ML policy and added size-awareness to it by taking the model's output probability and dividing it by the number of flash writes required to admit the item, which gave a 5%-savings in mean DT.

We found that with Baleen's episodes model, we could do away with this explicit size-awareness and instead optimize for size-awareness end-to-end by having it incorporated in the scoring function of OPT. As long as size-related features (IO start, IO size) were provided, Baleen was able to learn size-awareness implicitly.

Baleen suffers from late admissions but is limited by feature quality Fig 5.8 shows a remaining gap of 16%, indicating significant room for improvement. Episode-based analysis shows 9% of DT is lost to late admissions (i.e., where episodes are admitted after the first access). We observed Baleen learning to reject almost all items on the first access (a behavior similar to RejectX). Many training examples shared identical features (on the first miss) but had different labels. Baleen thus predicted the most probable label for each feature set (i.e., Bayes Optimal classifier behavior). We validated this finding by approximating a Bayes optimal classifier in analysis, and found an appreciable gap between it and OPT. Since dynamic, history-based features cannot differentiate unseen items, we hypothesize that better metadata features are required to distinguish the few true positives.

9.6 Prefetching on PUT

Prefetching-on-PUT can yield an additional hit on the first-ever access to the item. However, the odds make it a challenging problem as many written blocks are not touched again for the duration of our traces. The classes are even more severely skewed than for plain admission or prefetching, making it a burden rather than a boon for admission most of the time.

Table 9.5: PUT statistics of traces

Dataset	Year	PUT-Only Blocks	#PUT / #Acc
Region1	2019	46%	13%
Region2	2019	81%	14%
Region3	2019	46%	16%
Region4	2021	40%	10%
Region5	2023	33%	9%
Region6	2023	38%	10%
Region7	2023	38%	12%

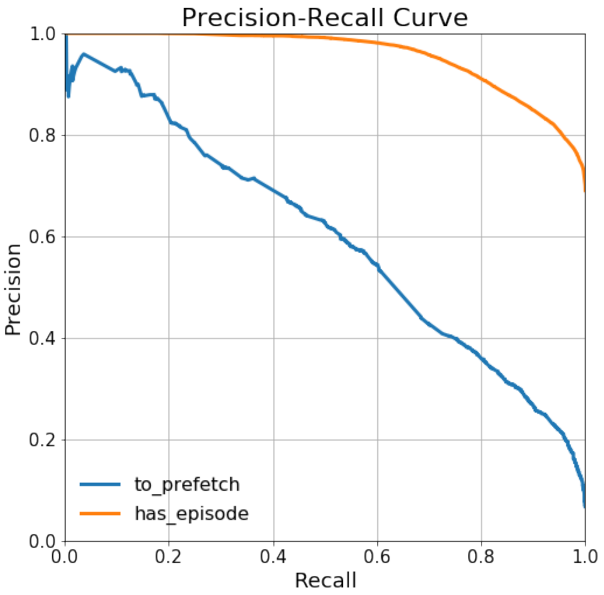


Figure 9.12: Predicting Prefetch on PUT. We show the recall and precision curves of two ML classifiers needed for Prefetch-on-PUT to work. `HasEpisode` denotes whether an item inserted upon PUT will get a hit before being evicted, whereas `ToPrefetch` predicts whether the episode will be admitted by the OPT given the flash write rate constraints. A classifier with a recall of 1 returns true on all true positives and a classifier with a precision of 1 has zero false positives.

We trained two classifiers: one to predict `ToPrefetch`, whether a PUT is followed by an episode that will be admitted by OPT; and `HasEpisode`, whether the time between a PUT and the subsequent GET is less than the cache eviction age.

Table 9.5 shows that an average of 46% of blocks across traces only have PUTs, meaning that This means that a classifier must be extremely accurate or else incur costly false positives.

Unfortunately, as shown by Fig 9.12, the ML classifier we trained does not exhibit a good precision-recall curve (to the right and up is better), with precision dropping rapidly as recall increases. We trained two classifiers: one to predict `ToPrefetch`, whether a PUT is followed by an episode that will be admitted by OPT; and `HasEpisode`, whether the time between a PUT and the subsequent GET is less than the cache eviction age. For a RAM cache, predicting `HasEpisode` correctly is sufficient, but for a flash cache, the policy also has to predict `ToPrefetch` correctly. Both need to be accurate in order for prefetching on PUT to work for a flash cache. Thus, we deem it not suitable for our workloads, but would not rule it out for other workloads with higher incidences of read-after-writes.

9.7 Lessons from ML deployment in production

We summarize a few lessons gleaned from 3 years of ML deployment in production caches at Meta. We were able to address some of these issues with Baleen, while others remain for future work.

ML model performance does not always translate to production system performance. The same algorithm performs differently when moved from offline to online settings, and again when moved from development to production environments. Evaluation in production is slow (many days needed to collect data in real time) and laborious (restarts, aborts, debugging). This makes it challenging to tune thresholds and evaluate improvements to ML policies. The plethora of directions makes it hard to decide on the best path forward without extensive exploratory research. This motivated our episodes model that allows for the principled design of ML policies that can directly optimize systems metrics like DT under write rate constraints, and quickly evaluate the end-to-end impact of hypothetical improvements without the effort to implement them in production or debugging unrelated production noise.

Rethink use of DRAM in flash caching. The typical use of DRAM is as a small cache before [5, 21] (or after [96]) flash, with admission decisions made on DRAM evictions. When Meta moved the admission policy from post-DRAM to pre-DRAM, there was surprisingly minimal impact on end-to-end metrics such as hit rates or Disk-head Time. The initial motivation was saving DRAM bandwidth, as this became a bottleneck with Admit-All rates near 500 MB/s (Table 3.1). The impact was small – while a DRAM cache may appear to absorb hits, it is simply stealing them from the flash cache. Since DRAM eviction ages (a few seconds) are so much shorter than flash (2+ hrs), almost every item worth caching needs to be in flash. Further, the write costs of an item are proportional to its size, and any potential avoidance of flash writes is limited by how small the DRAM cache is (2.5% of flash cache). Flexible placement of the admission policy enables optimizations such as prefetching, which must be done prior to inserting into the topmost cache. In summary, we need to find better uses for DRAM than simply adding it before a flash cache as part of a hybrid cache.

9.8 Summary

In this chapter, we presented multiple studies showing features we considered but which ultimately did not make it into Baleen. Leveraging our analytical model and simulators, we were able to estimate the potential benefits of different approaches and discern which were the most promising avenues to peruse.

A common thread in our explorations was that the benefit assessed from our proposals did not always match our initial intuition. For every modification that resulted in an improvement, there were many we tried which did not. While our intuition did improve over time, being able to prototype quickly and cut off explorations at the analytical model stage before progressing to a full implementation was always helpful.

We were also able to identify the limiting factors in the workloads that limited the benefit of modifications (and conversely, what it would take to see benefits.)

Chapter 10

Conclusions, lessons learned and future directions

This dissertation proposes the following thesis statement:

ML flash caching policies can reduce total cost in bulk storage systems, but in order to outperform heuristics in well-tuned production systems, they must have a flexible and principled design that can adapt to diverse workloads.

In investigating this thesis statement, I conducted numerous studies in which I applied machine learning (ML) to different areas of flash caching for bulk storage systems. I assessed the viability of each application of ML to flash caching on real production workloads using my simulator (BCacheSim) and a CacheLib testbed.

Outperforming heuristics in well-tuned production systems. Baleen represents a collection of our best policies (ML admission and ML prefetching) working in concert to deliver a significant reduction of 12% in backend load (Peak Disk-head Time) on 8 distinct clusters from 4 different years, with 10 workloads in total. We benchmarked our policies against the time-tested heuristics used in production and state-of-the-art ML baselines.

Reducing total cost in bulk storage systems. Baleen-TCO demonstrated a substantial reduction of 17% of total costs based on our TCO (total cost of operation) formula, which quantifies media costs, the largest cost component of bulk storage.

Principled approach to ML policy design. To help one understand what optimal looked like, I developed the episode model and OPT, an approximation of optimal for flash admission.

I also established Peak Disk-head Time as the correct metric to use for evaluating flash caches in bulk storage systems. In designing each application of ML, I used our analytical model to determine upper bounds, then designed a episode-based optimal for label generation, followed by training machine learning models to imitate the episode-based optimal.

Flexible design to adapt to diverse workloads. I evaluated strategies to mitigate the peak with adaptive selectivity based on load levels, and evaluated Baleen's performance under workload drift over time.

10.1 Lessons learned

Optimizing the wrong metric is an easy misstep: hit rate is not necessarily the right metric for caching problems. We learned a painful lesson with our early ML policy attempts when we took for granted that optimizing hit rate was correct because it was the unquestioned metric for caching in both industry and academia. (Indeed, hit rate is so entrenched that some people thought we were hiding something by not showing it.) Our initial prototypes for prefetching and admission increased IO hit rate, but was actually worse for DT. To overcome this, we redesigned our ML admission policy and introduced a prefetching confidence prediction (ML-When). Going back to the drawing board to redesign our caching policies cost us an extra year, but it was a valuable experience that gifted us Disk-head Time (a re-discovery), which was an elegant leap from previous incremental approaches that tried to balance object hit rate and byte hit rate.

Finding a way to approximate optimal is key to ML for caching (and ML for systems). A ML model can be no better than the training data given to it. Most ML models are supervised, and thus a way is needed to generate (approximate) optimal labels for the ML policy to imitate in making decisions. This dissertation presented such oracular models including OPT for admission and OPT-Range for prefetching. This is also a boost for model introspectability, which is essential to winning over systems practitioners given a widespread skepticism of ML in the systems community, as well as for debugging. Being able to benchmark against an approximate optimal and to know how much headroom is present before investing significant effort in developing a ML approach has been very valuable.

ML-based caching should aim for encapsulation of ML, caching, and storage. Designing bespoke ML for caching solutions requires coordination between ML experts (for model training), caching experts (for integration), and the storage backend owner (for deployment and monitoring). This involves one more area of expertise than most other ML for systems problems. There is no clear path to single ownership of the problem, making it difficult to sustain over time. It is hard for a service owner to prioritize spending engineering resources to aid the design phase of unproven ML solutions. Baleen provides an analytic framework that ML experts could optimize DT on without requiring caching expertise. Designing ML models around episodes makes it easier for caching experts to reason about. Having the DT formula correspond closely to measured DT (Fig 4.1) in production assures caching and storage experts that a reduction in calculated DT will translate to a drop in disk utilization. Further, with setups that are tightly-coupled by hand and not automatically, performance regressions may occur as systems and workloads change. Models often performed the best when they were first deployed and slowly regressed over time even with retraining using the same set of features. In contrast, Baleen was designed primarily using traces from 2019 but also demonstrates improvements on traces from 2021, 2023 and 2024.

Making the right assumptions and approximations. The beauty of the episodes model is in its decoupling of decisions that were formerly dependent on the cache state (and thus other decisions that would affect cache contents). A single parameter (the assumed eviction age) was sufficient to summarize the cache state for the most common eviction policies (LRU, FIFO). For prefetching, choosing the right granularity to make decisions at and determining that the segment range was sufficient to capture most of the benefit (as opposed to working with the much larger decision space of individual segments) was the right decision and approximation to make.

Data-driven development is especially important in ML for caching. I quickly learned the importance of data-driven development and prioritization of possible improvements. In the beginning, I eyeballed small samples of the trace in order to brainstorm possible modifications, but later realized that the situations I observed, while interesting, were not necessarily common enough to be worth solving.

Knowing which fidelity and context to develop in at the right time is key to making good progress. We ended up with 4 contexts in which policies could be evaluated: the analytical model, BCacheSim simulation, testbed CacheLib, and production CacheLib. There is a time to work on something in the analytical model, a time to work on it in simulation, and a time to work on it in CacheLib. A significant determinant of the velocity of research is influenced by the art of choosing how much time to spend working on an improvement in the respective contexts. Given the trade-offs between development velocity and testing fidelity, a funnel-shaped pipeline is right: some improvements should be filtered out at the analytical model stage, and some after simulations, with only simulation-proven policies making their way into the CacheLib implementation. Looking back, for a number of studies, less time should have been spent at the analytical model stage (where we figure out the maximum benefit) with more time at the simulation stage (where we figure out the achievable benefit).

There is a time to evaluate a component in isolation, and a time to evaluate it end-to-end. Getting prefetching to work required a delicate balance between both, particularly since prefetching required a good admission policy to work, and admission by itself did not yield as much benefits as expected.

10.2 Limitations: data, data, data

I discovered that data, rather than model architectures, was almost always the limiting factor in my explorations.

Discriminative features from automatic mining of unstructured text data. Academics suffer from not having enough rich features and longer traces, whereas the woe of industry practitioners is having too many sources of features and not enough time to clean it up into a trace. Much of the untapped signal lies in unstructured text data such as logs and tags, which is also the same data that is difficult to satisfactorily anonymize for public release. One promising avenue are embedding models that could be trained on the unstructured data and then used as a source of black box features for other downstream models. Coming up with such intermediate representations [104] was the original, grander vision of [105, 106]. The advent of large language models (LLMs) may unlock hitherto untapped sources of features with much lower engineering effort than power. This would be helpful for applications such as the prediction of episode timespan, ML for eviction and ML for prefetching-on-put.

Longer continuous caching traces to study drift. To understand gradual drift that takes place on the timescale of multiple months, we would require a continuous trace that spans the full time period. An important enabler to this would be an automatic trace collection pipeline that can store multiple months of data, to avoid the need for manual collection and stitching together of traces.

10.3 Future directions

We discuss avenues for future research arising from the work in this dissertation.

Use of episodes in other caching work. Episodes can be used to model other caching problems, and not just for admission and flash caching. Besides eviction policies, it could be used to modeling problems of placement and tiering.

Use of Disk-head Time and variants in storage Despite its name, the use of the Disk-head Time concept (where there is a constant setup time per access and a part that scales with bytes transferred) can be extended to more than HDD-backed systems. In fact, the concept can be extended to any situation where there are multiple tiers of storage and where there is a constant setup time in fulfilling requests, including in the cloud. This includes cloud object storage (such as Amazon S3 and Cloudflare R2), CDNs (content delivery networks), and storage hierarchies with multiple types of flash. Another way to look at it is that Disk-head Time will be useful whenever there is a need to optimize both object miss rate and byte miss rate.

Bibliography

[1] Zahaib Akhtar, Yaguang Li, Ramesh Govindan, Emir Halepovic, Shuai Hao, Yan Liu, and Subhabrata Sen. Avic: a cache for adaptive bitrate video. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 305–317, 2019. 2.5, 2.1

[2] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2205–2217, 2022. 2.6

[3] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *NSDI*, pages 389–403, 2018. 2.5, 2.1, 2.3, 4.3, 9.1.1

[4] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966. 1, 2.6, 9.1.1

[5] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768, 2020. 1.1, 2.3, 2.3.1, 2.5, 2.6, 2.1, 2.2, 2.3, 3.5, 5.1.1, 5.4.2, 8.1, 9.5, 9.2.1, 9.7

[6] Daniel S Berger. Towards lightweight and robust machine learning for CDN caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pages 134–140, 2018. 1, 1.1, 2.6, 9.1.1

[7] Daniel S Berger, Philipp Gland, Sahil Singla, and Florin Ciucu. Exact analysis of TTL cache networks. *Performance Evaluation*, 79:2–23, 2014. 4.2

[8] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. AdaptSize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 483–498, 2017. 2.5, 2.6, 2.1, 4.3, 8.1

[9] Daniel S Berger, Nathan Beckmann, and Mor Harchol-Balter. Practical bounds on optimal caching with variable object sizes. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):1–38, 2018. 9.1.1

[10] Albert Bifet and Ricard Gavalda. Learning from time-changing data with adaptive windowing. In *Proceedings of the 2007 SIAM international conference on data mining*, pages 443–448. SIAM, 2007. 8.1

[11] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. Hyperbolic caching: Flexible caching for web applications. In *USENIX Annual Technical Conference*, 2017. 2.3

[12] Scott A Brandt, Carlos Maltzahn, Anna Povzner, Roberto Pineiro, Andrew Shewmaker, and Tim Kaldewey. An integrated model for performance management in a distributed system. *OSPERT 2008*, page 25, 2008. 2.6

[13] Dariusz Brzezinski and Jerzy Stefanowski. Reacting to different types of concept drift: The accuracy updated ensemble algorithm. *IEEE Transactions on Neural Networks and Learning Systems*, 25(1):81–94, 2013. 8.1

[14] Chandranil Chakraborttii and Heiner Litz. Reducing write amplification in flash by death-time prediction of logical block addresses. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, pages 1–12, 2021. 2.6

[15] Hao Che, Ye Tung, and Zhijun Wang. Hierarchical web caching systems: Modeling, design and experimental results. *IEEE Journal on Selected Areas in Communications*, 20(7): 1305–1314, 2002. 4.2

[16] Yue Cheng, Fred Douglis, Philip Shilane, Grant Wallace, Peter Desnoyers, and Kai Li. Erasing Belady's limitations: In search of flash cache offline optimality. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 379–392, 2016. 2.6, 9.1.1

[17] Jakub Chłędowski, Adam Polak, Bartosz Szabucki, and Konrad Tomasz Żołna. Robust learning-augmented caching: An experimental study. In *International Conference on Machine Learning*, pages 1920–1930. PMLR, 2021. 2.6, 8.1

[18] Gil Einziger, Roy Friedman, and Ben Manes. TinyLFU: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31, 2017. 2.5, 2.6, 2.1, 2.2, 8.1, 9.1.4

[19] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive software cache management. In *Proceedings of the 19th International Middleware Conference*, pages 94–106, 2018. 2.2, 8.1

[20] Gil Einziger, Ohad Eytan, Roy Friedman, and Benjamin Manes. Lightweight robust size aware cache management. *ACM Transactions on Storage (TOS)*, 18(3):1–23, 2022. 2.6, 2.1, 2.2

[21] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 65–78, 2019. 1.1, 2.5, 2.6, 2.1, 2.3, 3.5, 5.1.1, 8.1, 9.2.1, 9.2.3, 9.7

[22] Ryan Elwell and Robi Polikar. Incremental learning of concept drift in nonstationary environments. *IEEE Transactions on Neural Networks*, 22(10):1517–1531, 2011. 8.1

[23] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen Kat. It's time to revisit LRU vs. FIFO. In *Proceedings of the 12th USENIX Conference on Hot Topics in Storage and File Systems*, pages 12–12, 2020. 2.3, 9.2.3

[24] Christine Fricker, Philippe Robert, and James Roberts. A versatile and accurate approximation for LRU cache performance. In *2012 24th international teletraffic congress (ITC 24)*,

pages 1–8. IEEE, 2012. 4.2

[25] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM computing surveys (CSUR)*, 46(4):1–37, 2014. 8.1, 8.1

[26] Sathya Gunasekar. From DRAM to SSDs, challenges with caching at FB scale. https://www.youtube.com/watch?v=RQUHnKbbOkI&t=490s, 2021. 9.2.1

[27] Mor Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013. 4.5

[28] Dean Hildebrand and Denis Serenyi. Colossus under the hood: a peek into Google's scalable storage system, 2021. 2.1

[29] Xinyue Hu, Eman Ramadan, Wei Ye, Feng Tian, and Zhi-Li Zhang. Raven: belady-guided, predictive (deep) learning for in-memory and content caching. In *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies*, pages 72–90, 2022. 2.6

[30] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, and Cheng Chen. Improving flash-based disk cache with lazy adaptive replacement. *ACM Transactions on Storage (TOS)*, 12(2): 1–24, 2016. 2.6

[31] Ellango Jothimurugesan, Kevin Hsieh, Jianyu Wang, Gauri Joshi, and Phillip B Gibbons. Federated learning under distributed concept drift. In *International Conference on Artificial Intelligence and Statistics*, pages 5834–5853. PMLR, 2023. 8.1, 8.5

[32] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30, 2017. 3.5

[33] Vadim Kirilin, Aditya Sundarrajan, Sergey Gorinsky, and Ramesh K Sitaraman. RL-Cache: Learning-based cache admission for content delivery. *IEEE Journal on Selected Areas in Communications*, 38(10):2372–2385, 2020. 2.6

[34] J Zico Kolter and Marcus A Maloof. Dynamic weighted majority: An ensemble method for drifting concepts. *The Journal of Machine Learning Research*, 8:2755–2790, 2007. 8.1

[35] Bartosz Krawczyk and Alberto Cano. Online ensemble learning with abstaining classifiers for drifting and noisy data streams. *Applied Soft Computing*, 68:677–692, 2018. 8.1

[36] Mathias Lecuyer, Joshua Lockerman, Lamont Nelson, Siddhartha Sen, Amit Sharma, and Aleksandrs Slivkins. Harvesting randomness to optimize distributed systems. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets)*, pages 178–184, 2017. 1, 1.1

[37] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *Proceedings of the 10th ACM International Systems and Storage Conference*, pages 1–11, 2017. 2.3

[38] Cheng Li, Philip Shilane, Fred Douglis, and Grant Wallace. Pannier: Design and analysis of a container-based flash cache for compound objects. *ACM Transactions on Storage (TOS)*,

13(3):1–34, 2017. 2.5, 2.1, 2.3

[39] Jinhong Li, Qiuping Wang, Patrick PC Lee, and Chao Shi. An in-depth analysis of cloud block storage workloads in large-scale production. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 37–47. IEEE, 2020. 2.6, 2.3

[40] Pengcheng Li and Yongbin Gu. Learning forward reuse distance. *arXiv preprint arXiv:2007.15859*, 2020. 2.6, 9.1.1

[41] Pengcheng Li, Colin Pronovost, William Wilson, Benjamin Tait, Jie Zhou, Chen Ding, and John Criswell. Beating OPT with statistical clairvoyance and variable size caching. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, 2019. 9.1.1

[42] Chieh-Jan Mike Liang, Hui Xue, Mao Yang, Lidong Zhou, Lifei Zhu, Zhao Lucis Li, Zibo Wang, Qi Chen, Quanlu Zhang, Chuanjie Liu, et al. AutoSys: The design and operation of Learning-Augmented systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 323–336, 2020. 1, 1.1

[43] John DC Little. A proof for the queuing formula: L= λ w. *Operations research*, 9(3): 383–387, 1961. 4.5

[44] Evan Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. An imitation learning approach for cache replacement. In *International Conference on Machine Learning*, pages 6237–6247. PMLR, 2020. 2.5, 2.1, 8.1

[45] Jiangchuan Liu and Bo Li. A QoS-based joint scheduling and caching algorithm for multimedia objects. *World Wide Web*, 7:281–296, 2004. 2.6

[46] Lanyue Lu, Peter Varman, and Kshitij Doshi. Graduated qos by decomposing bursts: Don't let the tail wag your server. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 12–21. IEEE, 2009. 7.1

[47] Christopher R Lumb and Richard Golding. D-sptf: Decentralized request distribution in brick-based storage systems. *ACM SIGPLAN Notices*, 39(11):37–47, 2004. 2.6

[48] Christopher R Lumb, Jiri Schindler, Gregory R Ganger, et al. Freeblock scheduling outside of disk firmware. In *FAST*, volume 2, pages 275–288, 2002. 2.6

[49] Martin Maas. A taxonomy of ML for systems problems. *IEEE Micro*, 40(05):8–16, 2020. 1, 1.1

[50] Ankur Mallick, Kevin Hsieh, Behnaz Arzani, and Gauri Joshi. Matchmaker: Data drift mitigation in machine learning for large-scale systems. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 77–94, 2022. URL `https://proceedings.mlsys.org/paper_files/paper/2022/file/069a002768bcb31509d4901961f23b3c-Paper.pdf`. 8, 8.1, 8.1, 8.2, 8.5

[51] Hongzi Mao, Shaileshh Bojja Venkatakrishnan, Malte Schwarzkopf, and Mohammad Alizadeh. Variance reduction for reinforcement learning in input-driven environments. *arXiv preprint arXiv:1807.02264*, 2018. 1, 1.1

[52] Valentina Martina, Michele Garetto, and Emilio Leonardi. A unified approach to the performance analysis of caching systems. In *IEEE INFOCOM 2014-IEEE Conference on*

Computer Communications, pages 2040–2048. IEEE, 2014. 4.2

[53] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S Berger, Nathan Beckmann, and Gregory R Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 243–262, 2021. 2.6, 2.2

[54] Chris Mellor. Enterprise SSDs cost ten times more than nearline disk drives. https://web.archive.org/web/20221004225419/https://blocksandfiles.com/2020/08/24/10x-enterprise-ssd-price-premium-over-nearline-disk-drives/, 2020. Accessed: 2022-10-04. 2.2, 6.2

[55] Dushyanth Narayanan, Austin Donnelly, Eno Thereska, Sameh Elnikety, and Antony IT Rowstron. Everest: Scaling down peak loads through i/o off-loading. In *OSDI*, volume 8, pages 15–28, 2008. 7.1

[56] Newegg. Newegg: Seagate Exos X18 ST10000NM018G 10TB 7200 RPM 256MB Cache SATA 6.0Gb/s 3.5" Hard Drives. https://web.archive.org/web/20230921032117/https://www.newegg.com/seagate-exos-x18-st10000nm018g-10tb/p/N82E16822185024?Item=N82E16822185024, 2023. Accessed: 2023-09-20. 6.2

[57] Newegg. Newegg: Dell Intel D3-S4620 960GB SATA 6Gb/s 2.5-inch Enterprise SSD. https://web.archive.org/web/20230921032102/https://www.newegg.com/dell-d3-s4620-960gb/p/2U3-000S-00104?Item=9SIA994K4B2373, 2023. Accessed: 2023-09-20. 6.2

[58] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. Facebook's Tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231, 2021. 2.1, 3.1.2, 6.1, 6.2, 7.1

[59] Ali Pesaranghader and Herna L Viktor. Fast hoeffding drift detection method for evolving data streams. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2016, Riva del Garda, Italy, September 19-23, 2016, Proceedings, Part II 16*, pages 96–111. Springer, 2016. 8.1

[60] Phitchaya Mangpo Phothilimthana, Saurabh Kadekodi, Soroush Ghodrati, Selene Moon, and Martin Maas. Thesios: Synthesizing accurate counterfactual i/o traces from i/o samples. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, pages 1016–1032. Association for Computing Machinery, 2024. ISBN 9798400703867. doi: 10.1145/3620666.3651337. URL https://doi.org/10.1145/3620666.3651337. 3.1, 3

[61] Timothy Pritchett and Mithuna Thottethodi. SieveStore: a highly-selective, ensemble-level disk cache for cost-performance. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 163–174, 2010. 2.6

[62] Liana V Rodriguez, Alexis Gonzalez, Pratik Poudel, Raju Rangaswami, and Jason Liu. Unifying the data center caching layer: Feasible? profitable? In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, pages 50–57, 2021. 7.1

[63] Liana V Rodriguez, Farzana Beente Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with CACHEUS. In *FAST*, pages 341–354, 2021. 2.5, 2.1, 2.3, 9.1.1, 9.2.5, 9.7

[64] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014. 8.1

[65] Shan-Hsiang Shen and Aditya Akella. An information-aware QoE-centric mobile video cache. In *Proceedings of the 19th annual international conference on Mobile computing & networking*, pages 401–412, 2013. 2.6

[66] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 413–425, 2019. 2.6, 9.1.1

[67] Leon Sixt, Evan Zheran Liu, Marie Pellat, James Wexler, Milad Hashemi Been Kim, and Martin Maas. Analyzing a caching model. *arXiv preprint arXiv:2112.06989*, 2021. 2.4

[68] Zhenyu Song, Daniel S Berger, Kai Li, Anees Shaikh, Wyatt Lloyd, Soudeh Ghorbani, Changhoon Kim, Aditya Akella, Arvind Krishnamurthy, Emmett Witchel, et al. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, 2020. 2.5, 2.6, 2.1, 2.2, 2.3, 4.3.1, 5.3.1, 9.1.1, 9.1.4

[69] Zhenyu Song, Kevin Chen, Nikhil Sarda, Deniz Altınbüken, Eugene Brevdo, Jimmy Coleman, Xiao Ju, Pawel Jurczyk, Richard Schooler, and Ramki Gummadi. HALP: Heuristic aided learned preference eviction policy for YouTube content delivery network. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023. 2.1, 2.5, 2.6, 2.1, 2.3, 7.1, 8.1

[70] George C Stierhoff and Alfred G Davis. A history of the IBM Systems Journal. *IEEE Annals of the History of Computing*, 20(1):29–35, 1998. 1

[71] Ashraf Tahmasbi, Ellango Jothimurugesan, Srikanta Tirthapura, and Phillip B Gibbons. Driftsurf: Stable-state/reactive-state learning under concept drift. In *International Conference on Machine Learning*, pages 10054–10064. PMLR, 2021. 8.1

[72] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, Kai Li, Wen Xia, Yucheng Zhang, Yujuan Tan, Phaneendra Reddy, Leif Walsh, et al. RIPQ: Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386, 2015. 2.5, 2.1, 9.1.1

[73] Tianqi Tang, Sheng Li, Lifeng Nai, Norm Jouppi, and Yuan Xie. Neurometer: An integrated power, area, and timing modeling framework for machine learning accelerators industry track paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 841–853. IEEE, 2021. 3.2

[74] Alexey Tsymbal. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin*, 106(2):58, 2004. 8.1, 8.1, 8.5

[75] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017. 9.3

[76] Olivier Verscheure, Chitra Venkatramani, Pascal Frossard, and Lisa Amini. Joint server scheduling and proxy caching for video delivery. *Computer Communications*, 25(4): 413–423, 2002. 2.6

[77] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ml-based LeCaR. In *HotStorage*, pages 928–936, 2018. 2.5, 2.1, 2.3, 9.1.1

[78] Matthew Wachs and Gregory R Ganger. Co-scheduling of disk head time in cluster-based storage. In *2009 28th IEEE International Symposium on Reliable Distributed Systems*, pages 278–287. IEEE, 2009. 2.6

[79] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R Ganger. Argon: Performance insulation for shared storage servers. In *FAST*, volume 7, pages 5–5, 2007. 2.6

[80] Hua Wang, Jiawei Zhang, Ping Huang, Xinbo Yi, Bin Cheng, and Ke Zhou. Cache what you need to cache: Reducing write traffic in cloud cache via "one-time-access-exclusion" policy. *ACM Transactions on Storage (TOS)*, 16(3):1–24, 2020. 2.6

[81] Hui Wang and Peter Varman. Statistical workload shaping for storage systems. In *2009 International Conference on High Performance Computing (HiPC)*, pages 274–283. IEEE, 2009. 7.1

[82] Peng Wang and Yu Liu. Optimizing replacement policies for content delivery network caching: Beyond Belady to attain a seemingly unattainable byte miss ratio. *arXiv preprint arXiv:2212.13671*, 2022. 2.6, 2.1, 2.3, 8.1, 9.1.1

[83] Qiuping Wang, Jinhong Li, Patrick PC Lee, Tao Ouyang, Chao Shi, and Lilong Huang. Separating data via block invalidation time inference for write amplification reduction in Log-Structured storage. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 429–444, 2022. 2.6, 2.3

[84] Gerhard Widmer and Miroslav Kubat. Learning in the presence of concept drift and hidden contexts. *Machine learning*, 23:69–101, 1996. 8.1

[85] Daniel Lin-Kit Wong, Hao Wu, Carson Molder, Sathya Gunasekar, Jimmy Lu, Snehal Khandkar, Abhinav Sharma, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Baleen: ML admission & prefetching for flash caches. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 347–371, Santa Clara, CA, February 2024. USENIX Association. ISBN 978-1-939133-38-0. URL https://www.usenix.org/conference/fast24/presentation/wong. 2.1, 2.2, 2.3

[86] Nan Wu and Pengcheng Li. Phoebe: Reuse-aware online caching with reinforcement learning for emerging storage models. *arXiv preprint arXiv:2011.07160*, 2020. 2.6, 9.1.1

[87] Gang Yan and Jian Li. RL-Bélády: A unified learning framework for content caching. In *Proceedings of the 28th ACM International Conference on Multimedia*, pages 1009–1017, 2020. 2.6, 2.1, 8.1, 9.1.1

[88] Dongsheng Yang, Daniel S Berger, Kai Li, and Wyatt Lloyd. A learned cache eviction framework with minimal overhead. *arXiv preprint arXiv:2301.11886*, 2023. 2.6, 2.1, 2.3, 8.1, 9.1.1

[89] Juncheng Yang, Yao Yue, and KV Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208, 2020. 2.2, 2.3, 8.1

[90] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 503–518, 2021. 2.3

[91] Juncheng Yang, Ziming Mao, Yao Yue, and KV Rashmi. GL-Cache: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 115–134, 2023. 2.1, 2.2, 2.3, 8.5, 9.1.1

[92] Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and KV Rashmi. Fifo can be better than lru: the power of lazy promotion and quick demotion. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 70–79, 2023. 9.1.4

[93] Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and K.V. Rashmi. Fifo can be better than lru: the power of lazy promotion and quick demotion. In *The 19th Workshop on Hot Topics in Operating Systems (HotOS 23)*, 2023. 9.2.3

[94] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. Fifo queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 130–149, 2023. 5.4.2, 9.1.4

[95] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. Leaper: A learned prefetcher for cache invalidation in lsm-tree based storage engines. *Proceedings of the VLDB Endowment*, 13 (12):1976–1989, 2020. 2.6

[96] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. CacheSack: Admission optimization for Google datacenter flash caches. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1021–1036, 2022. 1, 1.1, 2.3, 2.3.1, 2.6, 2.1, 2.3, 4.2, 5.1, 6, 8.1, 9.7

[97] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, Homer Wolfmeister, and Junaid Khalid. CacheSack: Theory and experience of Google's admission optimization for datacenter flash caches. *ACM Transactions on Storage*, 19(2):1–24, 2023. 2.3, 2.5, 2.6, 2.1, 3.1, 6, 6.1, 7.1, 8.1, 9.5, 9.2.1

[98] Jieming Yin, Subhash Sethumurugan, Yasuko Eckert, Chintan Patel, Alan Smith, Eric Morton, Mark Oskin, Natalie Enright Jerger, and Gabriel H Loh. Experiences with ML-driven design: A NoC case study. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 637–648. IEEE, 2020. 1, 1.1

[99] Lei Zhang, Reza Karimi, Irfan Ahmad, and Ymir Vigfusson. Optimal data placement for heterogeneous cache, memory, and storage systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(1):1–27, 2020. 9.1.1

[100] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and KV Rashmi. {SIEVE} is simpler than {LRU}: an efficient {Turn-Key} eviction algorithm for web caches. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1229–1246, 2024. 9.1.4

[101] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. OSCA: An Online-Model based cache allocation scheme in cloud block storage systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 785–798, 2020. 2.6, 2.3

[102] Yu Zhang, Ke Zhou, Ping Huang, Hua Wang, Jianying Hu, Yangtao Wang, Yongguang Ji, and Bin Cheng. A machine learning based write policy for SSD cache in cloud block storage. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1279–1282. IEEE, 2020. 2.6

[103] Mark Zhao, Satadru Pan, Niket Agarwal, Zhaoduo Wen, David Xu, Anand Natarajan, Pavan Kumar, Shiva Shankar P, Ritesh Tijoriwala, Karan Asher, Hao Wu, Aarti Basant, Daniel Ford, Delia David, Nezih Yigitbasi, Pratap Singh, Carole-Jean Wu, and Christos Kozyrakis. Tectonic-Shift: A composite storage fabric for large-scale ML training. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023. 2.3, 9.2.1

[104] Giulio Zhou. *Building reliable and transparent machine learning systems using structured intermediate representations*. PhD thesis, Carnegie Mellon University, 2024. 10.2

[105] Giulio Zhou and Martin Maas. Multi-task learning for storage systems. In *Proc. ML Syst. Workshop*, 2019. 10.2

[106] Giulio Zhou and Martin Maas. Learning on distributed traces for data center storage systems. *Proceedings of Machine Learning and Systems*, 3:350–364, 2021. 2.5, 2.1, 10.2

[107] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenjie Liu, and Tianming Yang. Improving cache performance for large-scale photo stores via heuristic prefetching scheme. *IEEE Transactions on Parallel and Distributed Systems*, 30 (9):2033–2045, 2019. 2.6

[108] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *USENIX Annual Technical Conference, General Track*, pages 91–104, 2001. 1

[109] Timothy Zhu, Anshul Gandhi, Mor Harchol-Balter, and Michael A. Kozuch. Saving cash by using less cache. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*, Boston, MA, June 2012. USENIX Association. URL https://www.usenix.org/conference/hotcloud12/workshop-program/presentation/zhu. 7.1

[110] Indrė Žliobaitė. Learning under concept drift: an overview. *arXiv preprint arXiv:1010.4784*, 2010. 8.1