Baleen: ML Admission & Prefetching for Flash Caches

Daniel Lin-Kit Wong[*] *Hao Wu*[†], *Carson Molder*[§], *Sathya Gunasekar*[†], *Jimmy Lu*[†], *Snehal Khandkar*[†]
Abhinav Sharma[†], *Daniel S. Berger*[‡], *Nathan Beckmann, Gregory R. Ganger*
Carnegie Mellon University; [†]Meta; [‡]Microsoft & University of Washington; [§]UT Austin

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, and using flash caches to provide a more cost-effective storage layer underlying everything from blobstores to data warehouses.

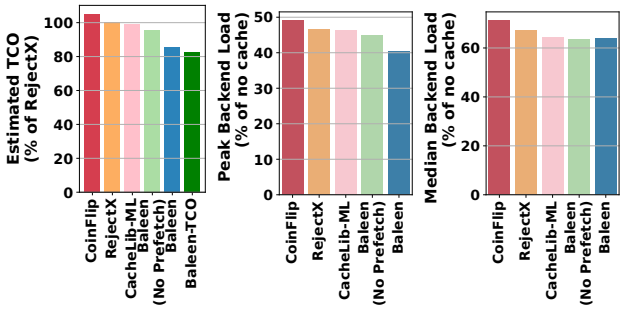
However, flash caches must address the limited write endurance of flash by limiting the long-term average flash write rate to avoid premature wearout. To do so, most flash caches must use admission policies to filter cache insertions and maximize the workload-reduction value of each flash write.

The Baleen flash cache uses coordinated ML admission and prefetching to reduce peak backend load. After learning painful lessons with our early ML policy attempts, we exploit a new cache residency model (which we call *episodes*) to guide model training. We focus on optimizing for an end-to-end system metric (*Disk-head Time*) that measures backend load more accurately than IO miss rate or byte miss rate. Evaluation using Meta traces from seven storage clusters shows that Baleen reduces *Peak Disk-head Time* (and hence the number of backend hard disks required) by 12% over state-of-the-art policies for a fixed flash write rate constraint. Baleen-TCO, which chooses an optimal flash write rate, reduces our estimated total cost of ownership (TCO) by 18%. Code and traces are available at github.com/wonglkd/Baleen-FAST24.

1 Introduction

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, 14]. Popular policies have included random admission and history-based policies that reject items without sufficient recent usage.



(a) Estimated TCO **(b)** Peak load **(c)** Median load

Figure 1: Baleen-TCO reduces (estimated) TCO by 17% over the best baseline on 7 Meta traces by choosing the optimal flash write rate. For the default flash write rate, Baleen reduces peak load by an average of 12% over the best baseline. IO and byte miss rates were reduced by 14% and 2%(Suppl A.1).

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, 21, 25, 28, 29, 57].

Making ML policies introspectable is key to their adoption by systems practitioners [55]. 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. In Baleen, we decompose the flash caching problem into admission, prefetching, and eviction (§3.3). This helped us align policy decisions to well-understood and efficient ML techniques for supervised learning. We do, however, want to

[*]Correspondence to dlwong@cs.cmu.edu

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.

This paper 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* (§3.4), 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* (3.5), 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.

Baleen is our resulting ML-guided Flash cache policy. We evaluate it by its savings in *Peak Disk-head Time* (§3.1), a measure of peak backend load, and we find 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% (Suppl A.1).

Contributions This paper makes 3 primary contributions: (1) a new cache residency model (*episodes*) that enables a useful comparison point (OPT) and improves ML training effectiveness; (2) ML-guided cache policies that optimize for Disk-head Time and TCO, not hit rate; (3) Baleen, which uses episodes to train coordinated ML admission and prefetching policies, saving 12% in peak load and 17% in (estimated) TCO over our best baseline (Fig 1).

2 Background

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 [16, 35, 42].

Accesses are made to byte ranges within **blocks**. Blocks are mapped to a location on backing HDDs and subdivided into many smaller units called **segments** that can be individually cached. (Tectonic has 8 MB 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.

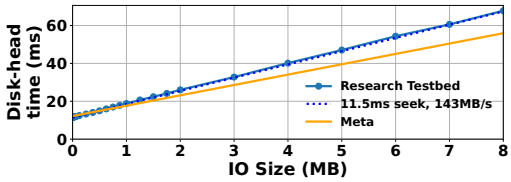


Figure 2: Disk-head Time (DT) for one IO. When a HDD performs an IO, the disk head **seeks** before it **reads** data. For tiny IOs, throughput is limited by *IOPS*; for large IOs, by *bandwidth*. DT encompasses both metrics and generalizes to variable-size IOs.

Each cluster has 10,000s of storage nodes independently serving requests. Each node has 378 TB in HDDs [35], 400 GB in flash cache, and 10 GB in DRAM cache (37,800:40:1). This paper focuses 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 [32]. Newer HDDs offer increased storage density, resulting in shrinking throughput (IOPS and bandwidth) per GB as more GBs are served by the same disk head.

Disk-head time (defined in §3.1) 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).

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.

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 [56]. 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 [60].

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, 55].

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 [55] 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 Challenges in flash caching

Challenges in flash admission Flash admission policies are difficult to design for many reasons. 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.

Challenges for ML admission We describe 4 challenges:

Correct optimization metric 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.

Asymmetrical misprediction cost 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 20 of Supp A.8). Policies trading off too many FNs for FPs suffer a performance hit [55].

Class imbalance 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.

APs 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.

Challenges for prefetching policies 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.

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 [14], 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, 13, 14, 22, 37, 43], bandwidth [41, 42] or write rate without considering the larger system the cache is part of. This makes them less performant and robust.
- **Not focused on peak.** 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 [42].
- **Not co-designed.** Many systems focus on a single aspect like flash admission [5, 13, 14] or eviction [3, 8, 22, 26, 37, 41–43, 47, 61] 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 ([1]: admission and eviction; [56]: admission and prefetching).

3 Exploring potential gains in flash caching

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.

3.1 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: $DT_i = t_{seek} + n \cdot t_{read}$

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

$$\sum_i DT_i = Fetches_{IOs} \cdot t_{seek} + Fetches_{Bytes} \cdot t_{read} \quad (1)$$

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 3, 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).

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.

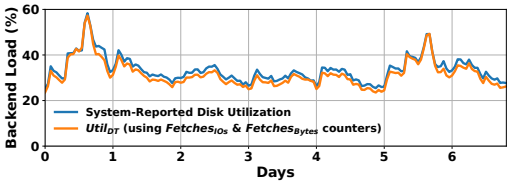


Figure 3: DT validated in production. Our DT formula (plugging counters into Eq 1) matches measured disk utilization (blue) closely. The peak of 58% occurs on Day 0.

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: $PeakDT = Util_{DT}^{P100}$

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 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.

3.2 TCO dominated by backend servers required

In the absence of actual cost numbers, we approximate TCO (total cost of ownership) based on public information. [56] defines TCO as the total cost of HDD reads and written flash bytes, assuming a fixed flash cache size and that other costs (CPU, RAM, power, network) are negligible. We design a similar function, assuming 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: $TCO \propto Cost_{HDD} \cdot \#HDDs + Cost_{SSD} \cdot \#SSDs$

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 ($FlashWR_0$).

$$\text{TCO}_1 \propto \frac{\text{PeakDT}_1}{PeakDT_0} \cdot \#HDDs_0 + \frac{Cost_{SSD}}{Cost_{HDD}} \cdot \frac{\text{FlashWR}_1}{FlashWR_0} \cdot \#SSDs_0 \quad (2)$$

This gives us a TCO function based on a policy's Peak DT ($PeakDT_1$) and the flash write rate chosen ($FlashWR_1$). (See App A.4 for a line-by-line derivation.) The skewed ratio of HDD to SSD capacity (945:1 [35]) means that SSD cost is a fraction of TCO (3% on our workloads). Hence, reducing Peak DT (and HDDs needed) is key to reducing TCO.

3.3 Decomposing the caching problem

We define the caching problem as determining which times we should fetch, admit, and evict each segment to minimize

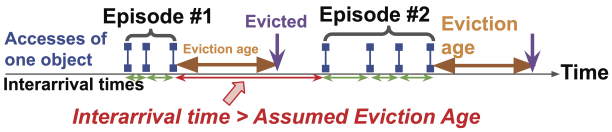


Figure 4: An episode is a group of accesses during a block's residency. Accesses (in blue) are grouped into two episodes as the interarrival time (in red) exceeds the assumed eviction age.

Figure 5: 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 (§ 3.6) is (1,3) and (2,3) respectively.

the backend's DT given a flash write rate limit.

We propose a heuristic decomposition of this problem into three sub-problems: 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 [40].

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. Here, we employ a simple eviction policy (in our case, LRU) as is used in production systems, leaving ML-based flash-aware eviction policies for future work.

3.4 Episodes: an offline model for flash caching

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 5, 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, 10, 15, 30]. In an LRU cache, the eviction age is the logical time between an item's last access & eviction. As shown in Fig 4, 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 [55]. 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.

3.5 OPT approximates optimal online AP

Using episodes, we can devise an admission policy (AP) for online simulation that approximates the optimal AP using offline information from the entire trace. First, each block's accesses are grouped into episodes using an assumed eviction age. Second, all episodes are scored and sorted. Last, 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): $Score(Episode) = \frac{DTSaved(Episode)}{Size(Episode)}$

3.6 Prefetching: what and when?

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

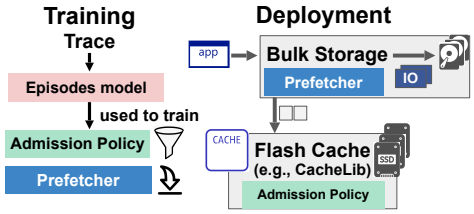


Figure 6: Architecture. An admission policy in CacheLib decides whether to admit items into flash. Prefetching (preloading of data beyond current request) takes place in Tectonic.

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 5, OPT-Range is (1,3) for Ep 1 and (2,3) for Ep 2. Whole-Block always fetches (1,64).

4 Baleen Implementation

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.

4.1 Training Baleen's ML admission policy

Episodes generated from the trace are used to train an admission policy, as shown in Fig 6. 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 [14], 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.

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.

Online flash caching simulator The training loop mentioned in the prior paragraph requires an online simulator to be run multiple times. 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 [44]. 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.

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-sourced 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.

4.2 Training Baleen's Prefetcher using episodes

Models are trained to solve two subproblems: what to prefetch, and when to prefetch.

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 §3.6). We use the same features as the ML admission model, 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.

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 3a). Further, prefetching mispredictions are costly in terms of DT consumed to fetch unused segments (which we call overfetch, see Eq 3b) 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 ($PFBenefit^{ML}_{eps}$, Eq 3c) be larger than ϵ (ML-When label, Eq 3e). 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).

$$UF: underfetch = true \text{ if ML-Range} \subset \text{OPT-Range} \tag{3a}$$

$$OF: overfetch = DTUsed(\text{extra segments}) \tag{3b}$$

$$PFBenefit^{OPT}_{eps} = DT^{NoPrefetch}_{eps} - DT^{OPT-Range}_{eps} \tag{3c}$$

$$PFBenefit^{ML}_{eps} = \begin{cases} 0 & \text{if underfetch} \\ PFBenefit^{OPT}_{eps} - OF & \text{otherwise} \end{cases} \tag{3d}$$

$$\text{ML-When}(eps) = PFBenefit^{ML-Range}_{eps} > \epsilon \tag{3e}$$

Prefetching is implemented 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 6). 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.

4.3 Optimizing for Peak DT and TCO

Baleen-TCO We designed a variant of Baleen, Baleen-TCO, that optimizes our TCO function (Eq 4b) by simulating Baleen over a range of flash write rates to get the respective Peak DT. Baleen-TCO then chooses the optimal flash write rate to minimize the TCO function.

Optimizing 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.

5 Evaluation

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

5.1 Experimental setup

We evaluate Baleen using a testbed and a simulator. We validate both with counters from production deployments. Our key results use simulation runs, but we validate individual points (e.g., Fig 11). We explain our setup, workloads, metrics, and the flash write rate and cache size constraints used.

ML training setup We wrote a Python module that generates episodes and trains the ML models. This plugs into a Python simulator for CacheLib we developed for training and prototyping (§4). We validate this Python simulator against testbed and production (§5.2). The episode module takes in a trace and returns the ML models. We then run simulation loops to converge on an assumed eviction age and admission policy threshold. LightGBM [19] was used for training and inference, with 500 rounds of boosting and 63 leaves.

Implementation in CacheLib We implemented support for ML admission and prefetching policies.We emulate calls to Tectonic so that every miss issues a real IO of the right size against HDDs, and measure the wall-clock time as DT consumed. Static features are stored in the CacheLib payload, while history counts are tracked by CacheLib. We use CacheLib's region-based LRU with a region size of 142 kB.

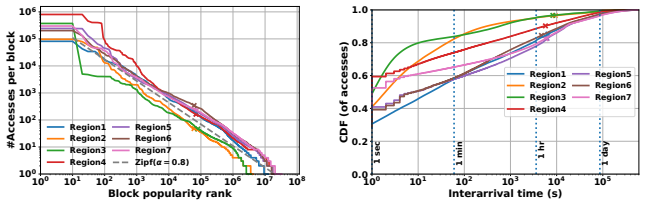
Overhead Baleen's overheads are low in the context of caching for bulk storage systems. *CPU overhead:* Baleen adds 4 inferences per IO miss (admission, start & end of ML-Range, ML-When). The system is limited by the latency of disk IOs upon misses (10–70ms per IO) rather than ML inferences (30 microseconds 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%).

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 (see Suppl A.9).

Table 1: Key statistics of traces.

Dataset	Req Rate (s^{-1})	Access size (MB)	CMR[1]	OHW[2]	Admit-All Writes (MB/s)
Region1	244	3.41	18%	54%	316
Region2	106	2.85	39%	83%	121
Region3	139	2.42	19%	48%	45
Region4	406	2.87	14%	53%	280
Region5	364	2.62	18%	59%	480
Region6	404	2.74	14%	55%	478
Region7	426	2.23	17%	62%	492

[1] CMR (Compulsory miss rate): ratio of blocks to accesses;
[2] OHW (One-hit-wonder): % of blocks with no reuse.



(a) Block popularity (log-log). **(b)** Interarrivals to same block.

Figure 7: Block popularity and access interarrival. In a, lower values of α indicate it is harder to cache, and × denotes 400 GB. In b, × denotes eviction ages for Baleen at 400 GB & 3 DWPD.

Workloads We report results for 7 production traces from Meta. Each workload sampled production traffic from a Tectonic [35] cluster, which can span an entire datacenter, including traffic for hundreds of applications. Traces were recorded in 3 different years. The popularity distribution of blocks (Fig 7a) fit a Zipf($\alpha = 0.8$) distribution, where the i-th most popular block has a relative frequency of $1/i^{\alpha}$. Fig 7b 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. 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. Full details are available in Supp A.8.

Each trace is sampled from an entire cluster (each numbering

thousands of nodes). A fraction of traffic is sampled at every node and the resulting samples aggregated to get a trace. The sampling rate and number of nodes are recorded, and used in further downsampling of the trace. For quicker simulation runs, the trace is sampled on the block key space, with each block weighted by number of accesses, with the cache size scaled down proportionally. 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 and Assumptions The savings from using Baleen are dominated by the degree by which it reduces the no. of HDDs required to handle peak load. Therefore, our evaluation focuses on Peak DT (see § 3.1). To aid comparison across traces, we normalize each policy's Peak DT by the Peak DT required with no cache. We also show estimated TCO savings over RejectX (using Eq 4b).

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)). Simulator results used 0.1-5%-traces. The sample percentage is higher for smaller workloads. We scale to a 400 GB-equivalent flash cache and our target flash write rate.

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 10 show Baleen performing at different flash write rates and cache sizes. Baleen-TCO also allows for a different flash-to-HDD cost ratio to be substituted in. For HDDs, our simulations assume a constant average seek time and bandwidth in the DT formula (Eq 1). These parameters vary minimally across disks, as illustrated in Fig 3 (simple formula closely matches actual disk utilization in production). Baleen includes a small benchmark to measure these constants for a given disk.

Baselines We compared Baleen to 4 baselines: RejectX, CoinFlip, and two state-of-the-art ML baselines, Flashield [14] 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 §5.2) and uses non-episode-related features.

5.2 Baleen reduces Peak DT over baselines

Fig 1b shows Baleen reduces Peak DT over RejectX by an average of 12% across all traces for a fixed target flash write rate. Fig 9 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 Suppl A.5). If we consider only workloads Flashield could train a model on, Baleen outperformed Flashield by 18%.

Validation of simulator and testbed Fig 11a shows us validating Baleen on our simulator against Baleen on our testbed. Further, we took the additional step of showing that our testbed is consistent with production counters, and show it matches closely (Fig 11b).

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 [41]. In addition, we quantify the benefit of episodes by comparing Baleen to an earlier ML admission policy that did not use episodes. Adding prefetching to the non-episode-based ML admission would cause it to perform worse than without prefetching.

Benefits consistent at higher write rates and larger cache sizes Fig 10 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. Graphs for all 7 traces are available in Supp A.11.

5.3 Baleen-TCO chooses optimal flash write rate

Fig 8 shows Baleen-TCO reducing TCO by 17% over CacheLib-ML and 18% over RejectX. Workloads have different optimal flash write rates; Baleen-TCO picks the best flash write rate for each, as illustrated in Fig 12. 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). Flash writes account for 2% to 5% of TCO (3% on average).

5.4 Prefetch selectively, in tandem with admission

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.

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 13). Baleen admission is used in all cases, with only the prefetching policy varied. We note this comes with a small increase in Median DT.

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 14. Prefetching only on ML-When or on Partial-Hit consistently does better than

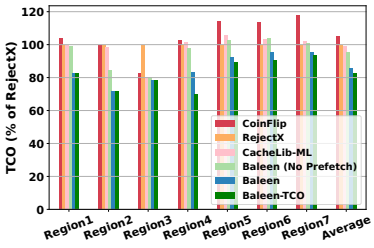


Figure 8: Baleen-TCO reduces TCO.

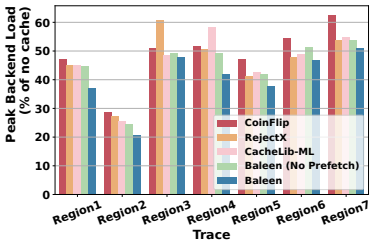


Figure 9: Baleen reduces Peak DT.

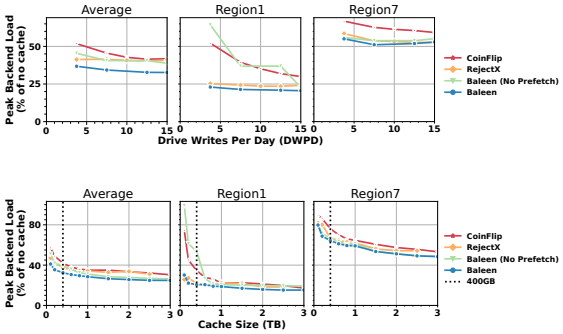
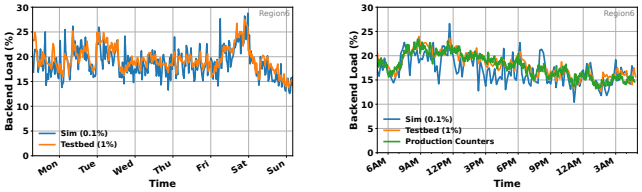


Figure 10: Benefits at higher write rates & cache sizes.



(a) Sim vs Testbed, Baleen **(b)** Testbed vs Prod, RejectX

Figure 11: Validation of simulator and testbed.

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.

5.5 Optimizing the right metric: Peak DT

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 example, for the prefetching option "ML-Range on Every Miss" from Fig 14. relative to no prefetching, the mean DT used ratio increases from 67% to 73% despite the IO hit ratio increasing from 46% to 47%.

DT during peak periods 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. We show further analysis in Supp A.3.

5.6 Other ML-guided cache results/experiences

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.

GBM better than deep models (Transformer & MLP) We compared GBM to more complex ML architectures (a Transformer-based architecture we designed and MLP). We found that GBM performs best (0.2% better than Cache Transformer), despite only having features for the current access. A challenge we faced when training these deep models were the highly imbalanced classes. Details are in Supp C.

Explicitly optimizing Peak DT Fig 15 shows DT varying over time, with a peak-to-mean ratio of 2. A policy wanting to optimize Peak DT should be aware of the current load level and able to adapt to it. We performed a simple extension where we only admitted to the cache during periods of high load. We found that while this saved flash writes, it did not reduce Peak DT. This suggests that more fundamental changes (e.g., scoring episodes by their usefulness in reducing Peak DT) will be required to optimize explicitly for peak load.

Baleen benefits from size-awareness. An earlier ML model required explicit size-awareness for a 5%-savings in mean DT. Baleen learns it implicitly if size-related features are supplied.

Gap between Baleen and OPT Fig 13 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). Since dynamic, history-based features cannot differentiate unseen items, we hypothesize that better metadata features are required to distinguish the few true positives.

Segment-aware admission & prefetching Baleen operates at the block level and can only choose to admit or reject the entire access range, rather than individual segments (unlike RejectX). Episode-based analysis showed a potential reduction of DT by 11%. However, we were unable to realize this.

Prefetch on PUT This would yield an additional hit on the first-ever access to the item. However, this is difficult as many written blocks are not touched again for the duration of our

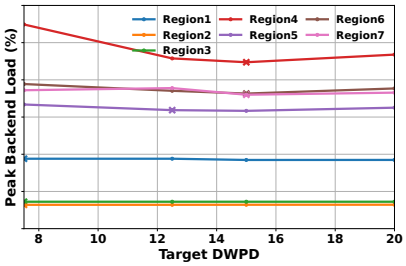


Figure 12: Baleen-TCO chooses higher flash write rates when needed to lower peak backend load (and TCO). × denotes the optimal flash write rate for that workload.

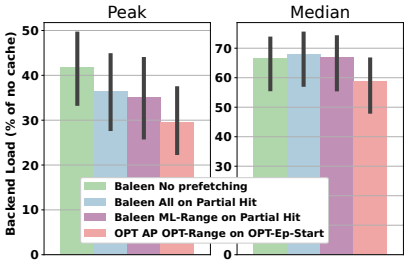


Figure 13: ML-Range saves Peak DT. ML-Range outperforms the baseline (whole block) and No Prefetching at the expense of Median DT.

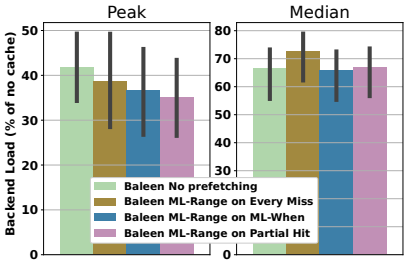


Figure 14: Choose *when* to prefetch. Indiscriminate prefetching (on Every Miss) can hurt. Using ML-When or Partial Hit reduces Peak DT without compromising Median DT.

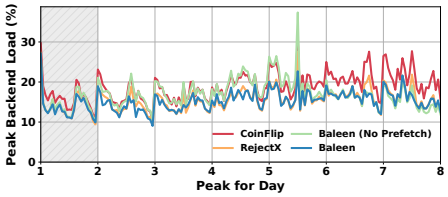


Figure 15: Testbed backend load on Region1. Day 1 (shaded) is used as training data. Peak is on Day 5 and is lowest for Baleen.

traces, meaning that a classifier must be extremely accurate or else incur costly false positives. This could work for other workloads with higher incidences of read-after-writes.

Early eviction If items could be evicted immediately after their last access in an episode, instead of waiting to leave the cache, this would eliminate dead time and result in a greater effective cache size. Episode-based analysis showed mean DT could potentially be reduced by 11%. However, our current ML models are not accurate enough to realize this.

6 Lessons from deploying ML in production

We summarize a few lessons gleaned from 3 years of deploying ML in production caches at Meta.

Optimizing the wrong metric is an easy misstep. 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). Picking the right end-to-end metric is important.

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, 14] (or after [55]) flash, with admission decisions made on DRAM evictions. We moved the admission policy from post-DRAM to pre-DRAM, with minimal impact on end-to-end metrics. The initial motivation was saving DRAM bandwidth, as this became a bottleneck with Admit-All rates near 500 MB/s (Table 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.

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 3) 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 and 2023.

7 Additional related work

Production flash caching systems CacheSack [55, 56] 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 only solves the admission problem. 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. Section 5.6 shows that this approach is significantly less effective than Baleen. Kangaroo [31] 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 [23] and Tencent [58].

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 discusses RejectX. Section 5 extensively compares Baleen to random (CoinFlip) and RejectX. LARC [18] is equivalent to RejectX and was the default admission policy used at Google prior to CacheSack. TinyLFU [13] 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 [14] 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 Supp A.5). More targeted applications of ML aim to exclude one-hit-wonders [48] or items that have no reads [59]. Reinforcement learning has also been used to train a feedforward neural network for admission policies on CDNs, given a broad set of features [20]. 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 [36]. Recent work instead uses simpler eviction policies such as CLOCK or FIFO, and leaves the heavy lifting to the admission policy [55]. Smart policies for data placement seek to reduce write amplification [9], and can be used in tandem with Baleen.

Prefetching policies CacheSack [56] incorporated static prefetching policies as choices for their optimization function. [62] 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 [27, 38, 46]. In databases, Leaper trains a ML prefetcher to exploit reuse at the key range level [54].

Models for caching and offline optimal Bélády's MIN algorithm is the optimal eviction policy [4]. [41] 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 Suppl A.7). Raven [17] is a probabilistic approximation of MIN. [11] 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 arbitary 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 [41], and RL-Bélády [51]. A key challenge to using RL is the long delays for rewards. [6] Others seek to go beyond Bélády, such as LRU-BaSE [49]. MAT [52] reduces ML inference overhead by using a heuristic to filter out likely candidates. HALP [42] augments a heuristic with ML for the YouTube CDN. Deep learning has also been applied to learn forward reuse distance with LSTMs [24] and reinforcement learning [50]. [39] uses a support vector machine with features they derived from training an LSTM. [12] 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.

8 Conclusion

Baleen uses ML to guide both prefetching and cache admission, reducing peak disk time by 12% and TCO by 17% on real workload traces, compared to state-of-the-art non-ML policies. Although applying ML to caching policies is an expected advancement, Baleen's design arose from false-step lessons and a cache residency (episodes) formulation 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.

Acknowledgments

We thank Meta, Google, and the members and companies of the PDL Consortium (Amazon, Google, Hitachi, Honda, IBM Research, Intel Corporation, Meta, Microsoft Research, Ora-

cle, Pure Storage, Salesforce, Samsung, Two Sigma, Western Digital) and VMware for their interest, insights, feedback, and support. We thank the anonymous reviewers and our shepherd, Ali Anwar, for their helpful comments and suggestions. This research is supported in part by NSF grant CNS1956271.

References

[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] 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.

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

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

[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.

[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.

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

[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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

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

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

[29] 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.

[30] 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.

[31] 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.

[32] 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-d 2020. Accessed: 2022-10-04.

[33] 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. Accessed: 2023-09-20.

[34] 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. Accessed: 2023-09-20.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

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

[51] 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.

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

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

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

[62] 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.

9 Artifact Appendix

Abstract

Our artifact is targeted at those seeking to reproduce the results found in the Baleen paper. It contains a Python simulator, an implementation of our cache residency model (*episodes*), and scripts for downloading traces. You may view our artifact and its README at https://github.com/wonglkd/Baleen-FAST24.

Scope

Our artifact allows users to easily 1) test our Python cache simulator with small-scale experiments, and 2) plot paper figures using supplied intermediate results.

We list the specific key claims and corresponding figures:

1. Baleen reduces estimated total cost of ownership (TCO), peak backend load (Fig 1) and miss rates (Fig A.1) (`fig-01a,08,13-202309-tco.ipynb`, `fig-01bc,17-202309.ipynb`)
2. Baleen does so across a range of traces (Fig 8, Fig 9) (`fig-01a,08,13-202309-tco.ipynb`, `fig-09-202309.ipynb`)
3. Baleen performs well across a range of cache sizes and flash write rates (Fig 10, 24, 25) (`fig-10a,24-wr-20230414.ipynb`, `fig-10b,25-csize-20230424.ipynb`)
4. Baleen benefits from smart prefetching that predicts the right range (Fig 13) and when to prefetch (Fig 14) (`fig-13,14-prefetching-20230424.ipynb`)

We also include additional notebooks that:

1. show how Baleen-TCO picks the optimal write rate (Fig 12), (`fig-01a,08,13-202309-tco.ipynb`)
2. show breakdown of benefit at peak (Fig 18) and (`fig-18-peak-hrs-20230424.ipynb`)
3. describe statistical properties of the workloads (Fig 7). (`fig-07,19,20-tracestats-20230504.ipynb`)

Caveats

When reproducing the results, we expect trends to be the same but small differences in the actual results due to two reasons: 1) Meta's exact constants for the disk-head time function will not be released, meaning that results will not be exactly the same; instead, in the released code, we use constants (seek time and bandwidth) measured on the hard disks in our university testbed; 2) the testbed code modified a proprietary internal version of CacheLib and that will not be released at this time. However, we expect the simulator to closely match the testbed (and have presented supporting evidence to that effect).

While all the necessary code and data is supplied to reproduce our results, setting up the simulator with a cluster scheduling system would be recommended if re-running all experiments (624 machine-days were utilized; each simulation of a ML policy takes at least 30 minutes, multiplied by 7 traces and 10 samples each). Helper code is included to facilitate runs on a cluster, but this will need to be adapted for your own cluster (see `BCacheSim/episodic_analysis/local_cluster.py`).

Contents

Our artifact includes the full traces used in the paper, a Python module (BCacheSim) that contains the flash cache simulator, an implementation of the episodes model, and code to train the policies. Further detail on the directory structure can be found in the README.

We also provide a walkthrough video that shows the authors reproducing the results on the Chameleon Cloud platform: http://tiny.cc/BaleenArtifactYT

Hosting

The artifact is hosted in a GitHub repository, in the main branch: https://github.com/wonglkd/Baleen-FAST24. For ease of reproduction, the artifact is also hosted on the Chameleon Cloud platform, a free academic cloud supported by NSF: https://www.chameleoncloud.org/experiment/share/aa6fb454-6452-4fc8-994a-b028bfc3c82d Users can choose to either use the artifact on their own machines or Chameleon.

Requirements

If using Chameleon Cloud, no local dependencies are required apart from the ability to SSH and a web browser. If using your own computer, the primary software dependency is Python 3.10 with specific packages listed in a `requirements.txt` in the repository. If you wish to run experiments in parallel on a cluster, a job scheduling system like Brooce (which we used) is recommended.

Baleen was developed on the Carnegie Mellon University Parallel Data Lab's Emulab testbed using Meta traces.

Time to reproduce

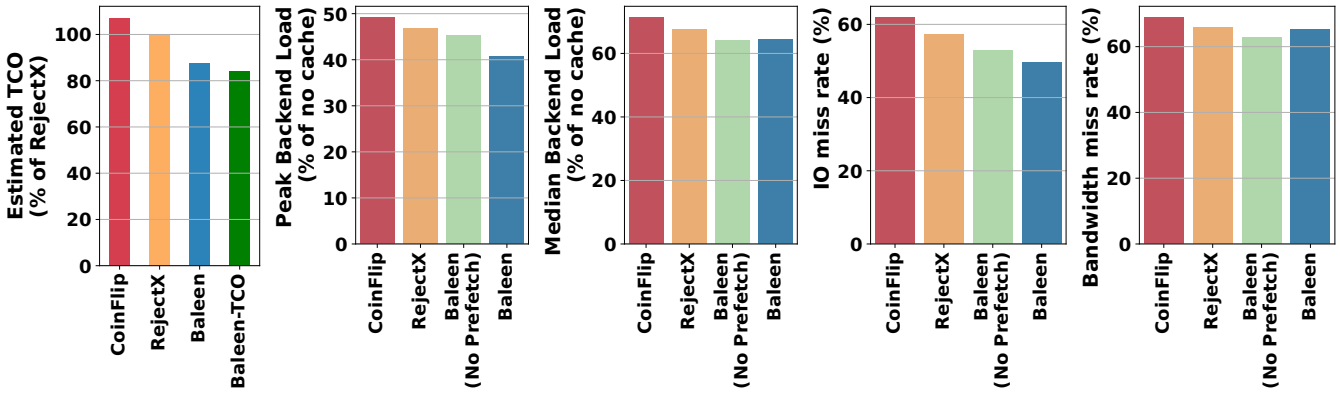
About 3 hours is required on Chameleon Cloud to run a set of basic experiments, and plot figures using the intermediate results supplied. To re-run all experiments from scratch would take 624 machine-days (based on the logged time it took to simulate the runs used). As a guideline, each simulation of a ML policy takes at least 30 minutes, multiplied by 7 traces and 10 samples each.

Troubleshooting and suppoort

A list of common issues and remedies is included in the README. GitHub issues are the preferred means of communication. You may also contact the corresponding author via email.

A Supplemental Material

A.1 Comparison to IO miss rate and bandwidth miss rate



(a) Estimated TCO **(b)** Peak load **(c)** Median load **(d)** IO miss rate **(e)** Byte miss rate

Figure 16: We provide IO miss rate and byte miss rate, two commonly used caching metrics, for comparison.

A.2 Median DT

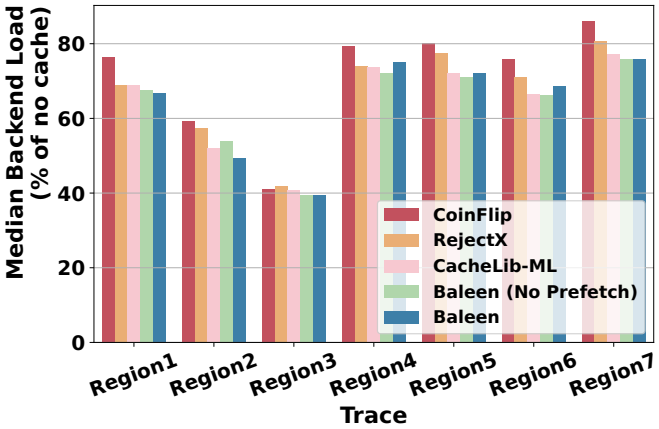
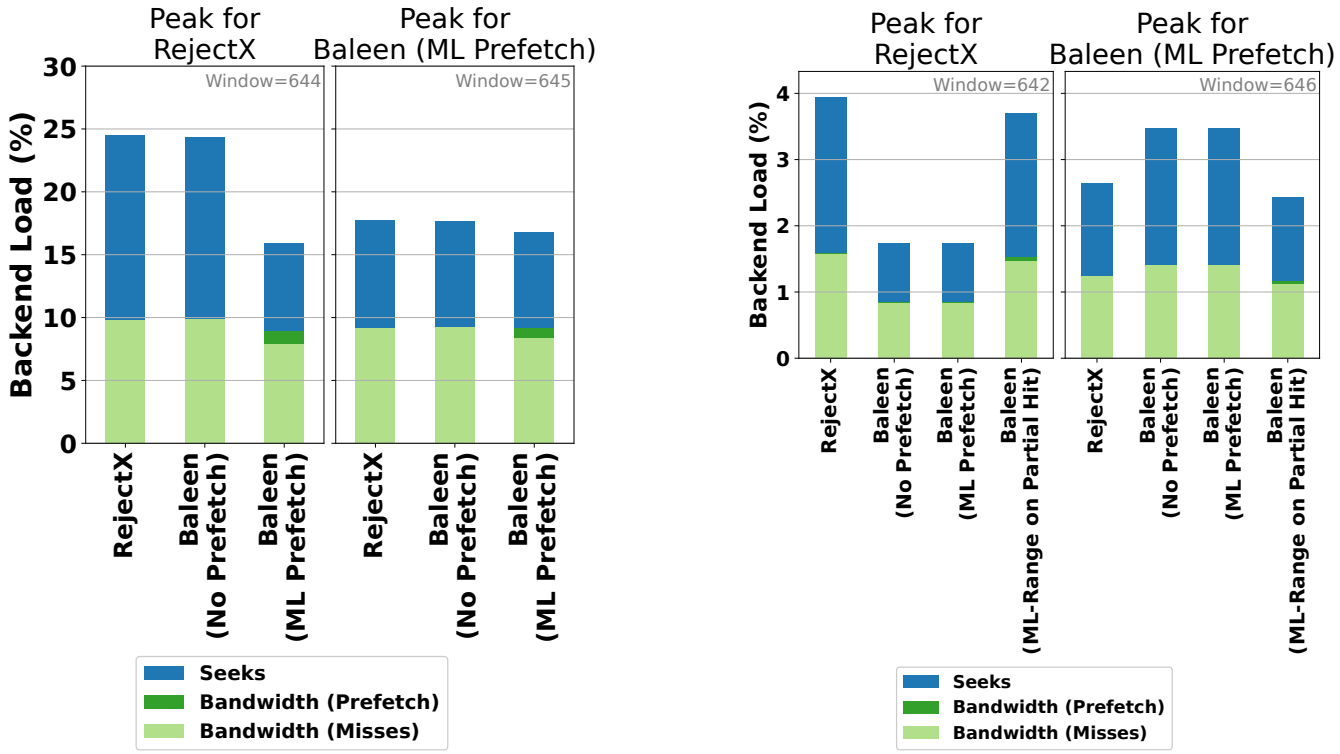


Figure 17: Median DT

A.3 Breakdown of DT during peak periods



(a) Region1 **(b)** Region7

Figure 18: Breakdowns of DT at Peaks. Each graph shows the peak 10-min window for that setup. Baleen's DT reduction is mostly due to reduced seeks.

In Fig 18a, we break down DT at the peak hours and show most of the Peak DT reduction is from eliminating seeks rather than data transfer. This is in line with how prefetching saves DT.

Fig 18 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 would be aware of the current load level and able to adapt to it.

A.4 TCO function: step-by-step

$$\mathrm{TCO}_1 \propto \frac{\mathrm{PeakDT}_1}{PeakDT_0} \cdot \#HDDs_0 + \frac{Cost_{SSD}}{Cost_{HDD}} \cdot \frac{\mathrm{FlashWR}_1}{FlashWR_0} \cdot \#SSDs_0 \tag{4a}$$

$$TCO_1 \propto PeakDT_1 \cdot R_1 + FlashWR_1 \cdot R_2 \tag{4b}$$

$$R_1 = \frac{1}{PeakDT_0} \tag{4c}$$

$$R_2 = \frac{1}{FlashWR_0} \cdot \frac{\#SSDs_0}{\#HDDs_0} \cdot \frac{Cost_{SSD}}{Cost_{HDD}} \tag{4d}$$

$$= \frac{1}{FlashWR_0} \cdot \frac{1}{36} \cdot \frac{170}{281} \tag{4e}$$

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 [35], we know that each node has 1x 1-TB SSD and 36x 10-TB HDDs ($\frac{\#HDDs_0}{\#SSDs_0} = 36$).

From [35], we know that each node has 1x 1-TB SSD and 36x 10-TB HDDs ($\frac{\#HDDs_0}{\#SSDs_0} = 36$). We substitute the 2023 price of a 10TB HDD ($281) and a 1 TB SSD ($170) on Newegg [33, 34] ($\frac{Cost_{SSD}}{Cost_{HDD}} = \frac{170}{281}$), i.e., the HDD is 6x cheaper per TB than the SSD. (For comparison, a 2020 industry report showed a 10x difference [32].)

A.5 Comparison to 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 [53]. 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%.

A.6 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 §6.

A.7 Comparison to LRB's Relaxed Belady

LRB [41] 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).

A.8 Workloads

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.

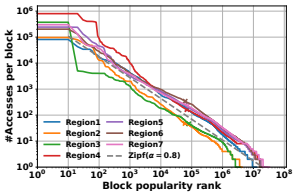
1. Regions 1-3 (2019): each a data warehouse
2. Region4 (2021): data warehouse
3. Region5 (2023): 10 "tenants", largest being data warehouse and blob store
4. Region6 (2023): 10 "tenants", largest being data warehouse and blob store
5. Regions 4-6 are from different geographical regions.

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 [35] paper for further details.

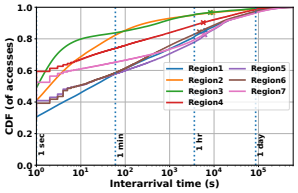
Table 2: Full statistics of traces.

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

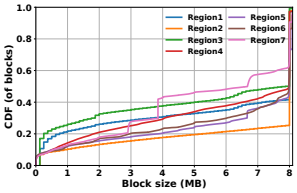
[1] Compulsory miss rate refers to the ratio of blocks to accesses;
[2] One-hit-wonder rate is the fraction of blocks with no reuse.



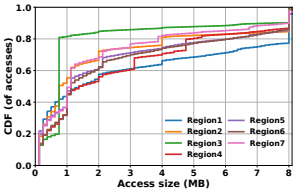
(a) Block popularity (log-log). × denotes 400 GB.



(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 19: Distributions of block popularity, access interarrival times, block sizes, and access sizes for three traces. In a, lower values of α indicate it is harder to cache.

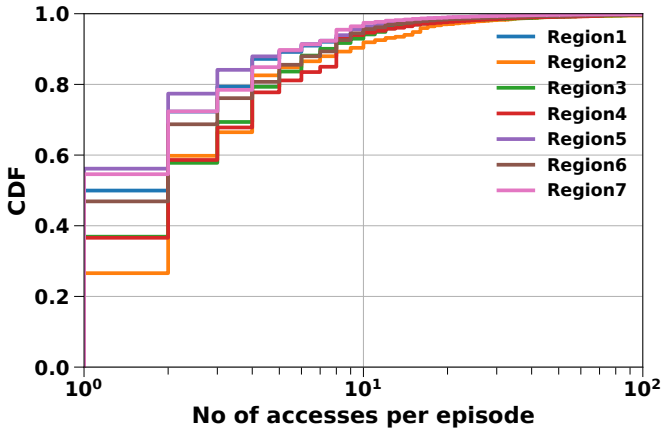


Figure 20: Distribution of hits per episode. This reflects the possible hits accrued from admitting an item.

A.9 Testbed hardware

As we did not have direct access to production hardware, we ran simulations (using our Python simulator) and testbed evaluations (using our modified version of CacheLib) on our academic testbed. This research testbed was 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, as the total number of runs required is the total number of policy configurations multiplied by 7 traces and 10 samples from each trace.

A.10 Validating simulator and testbed

Fig 21 shows that testbed and simulator are faithful to production counters. 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.

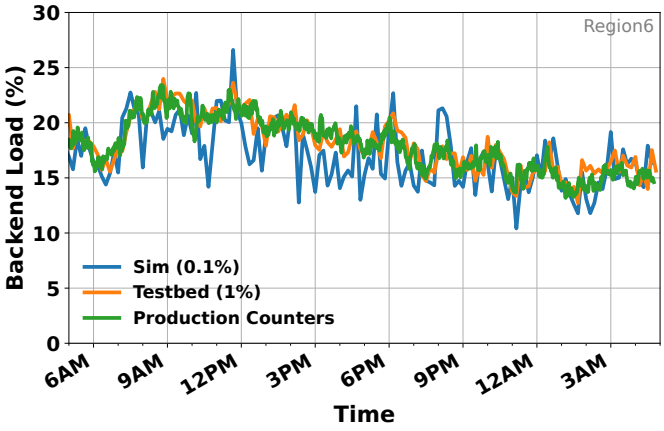


Figure 21: Sim-Testbed-Production comparison, RejectX, 1 day

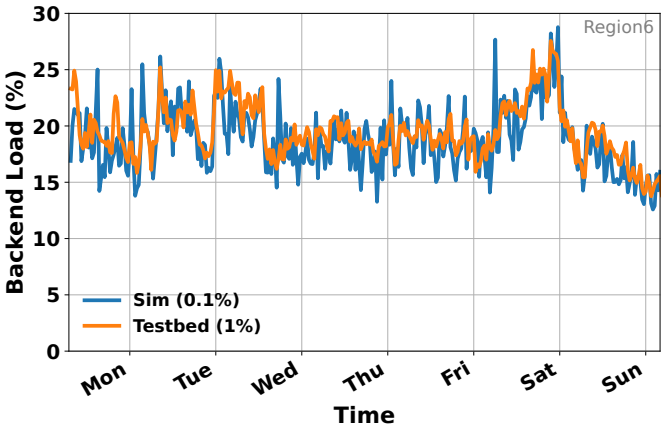


Figure 22: Testbed-Production comparison, Baleen, 1 week

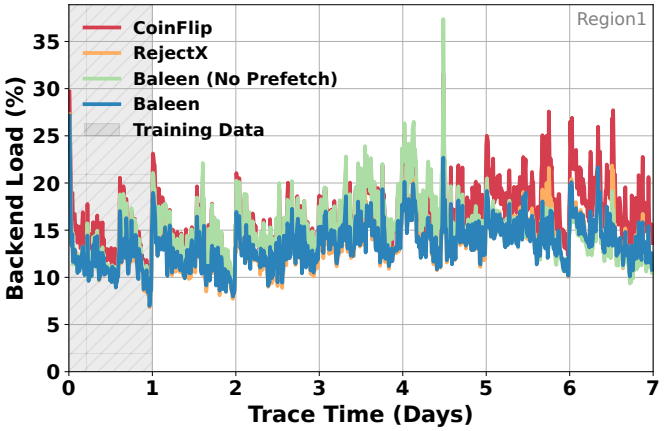


Figure 23: Testbed backend load over time, on the Region1 trace. Peak-to-mean ratio is 2. Granularity is 10 mins.

A.11 Write Rates and Cache Sizes for all traces

In Fig 10 we showed an average across all traces and selected traces; here we show data for all 7 traces.

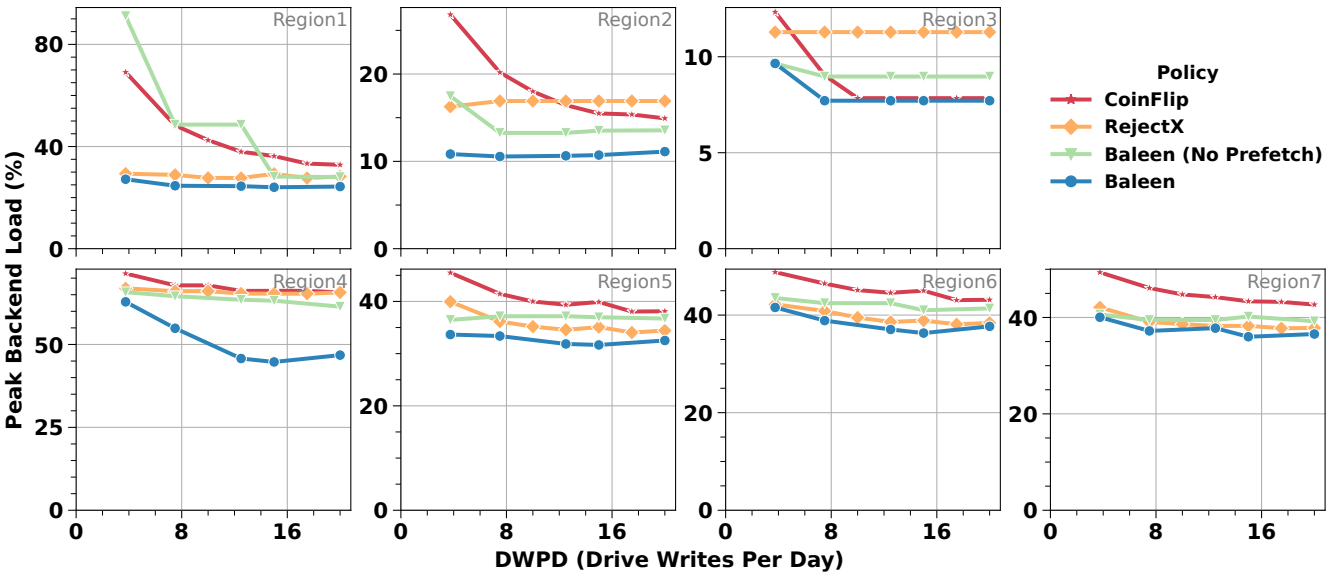


Figure 24: Benefits consistent as write rate increases.

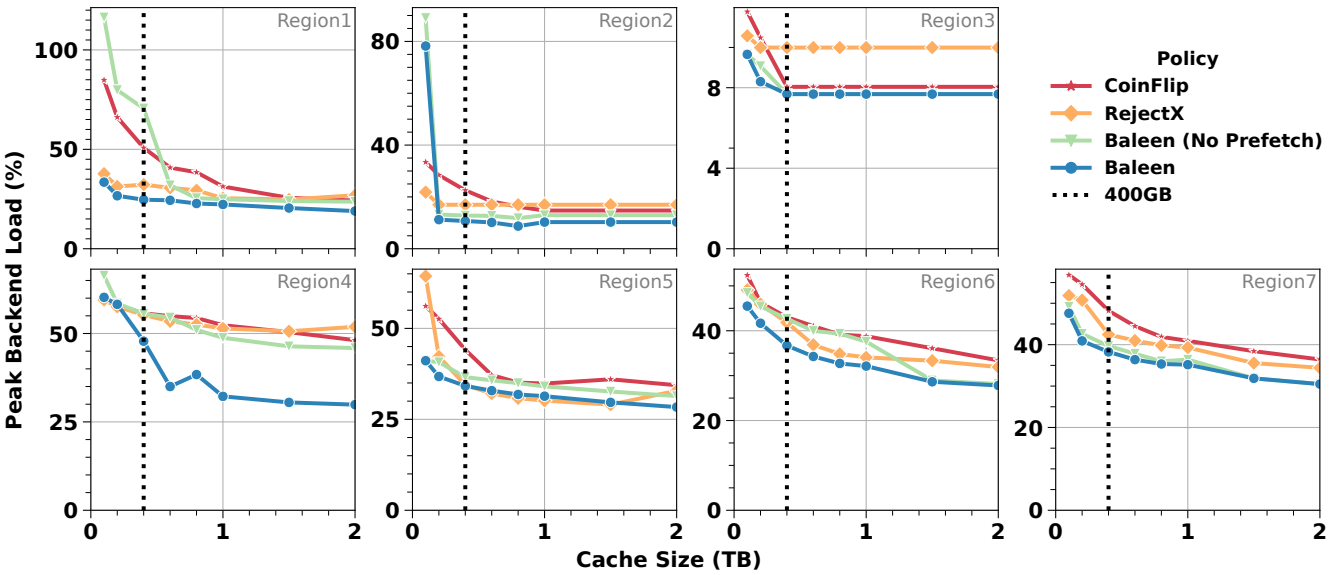


Figure 25: Benefits consistent as cache size increases.

B CacheLib deployment

B.1 CacheLib settings

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.

We added functionality to CacheBench (CacheLib's benchmark suite) to replay Tectonic traces.

C 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 [45] 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. Further details are in Supp C.1.

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. 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. 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. Baleen hence uses GBM given that it performs best and is the most efficient of the options explored.

C.1 Architecture

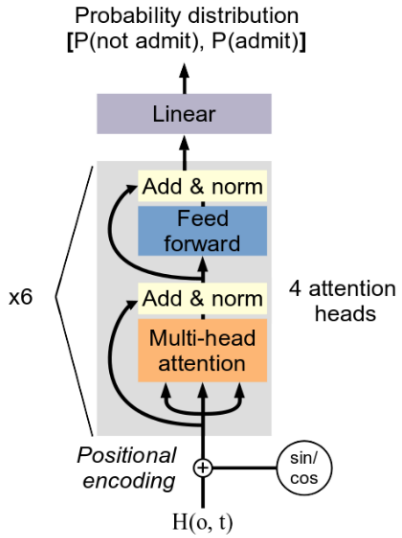


Figure 26: Cache Transformer architecture.

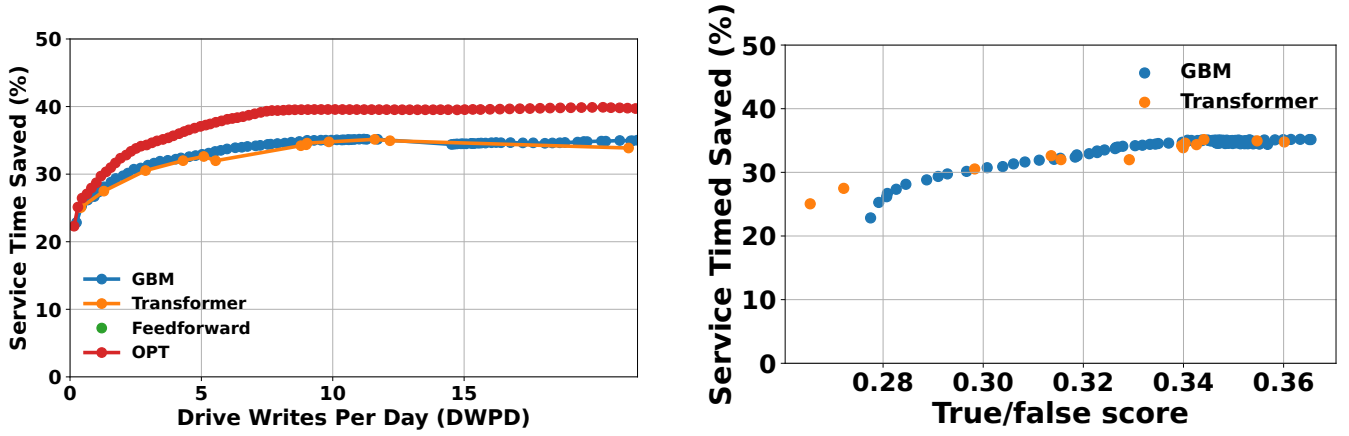
As shown in Fig 26, the Cache Transformer architecture consists of a series of Transformer encoders stacked together, with a linear classifier at the end. Before being passed to the first encoder, the windows are normalized and a sinusoidal positional encoding is applied. The encoders serve the purpose 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.

In summary: 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.

C.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.

C.3 Evaluation



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

(b) Disk-head Time Saved against true/false linear regression score.

Figure 27: 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.

Table 3: 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	Precision	Recall	F_1
MLP feedforward ($h = 1$)	0.41	90.2%	88.5%	28.1 MB/s	48.1% (-8.6%)	85.6%	35.5%	0.502
Transformer ($h = 16$)	0.18	92.6%	89.5%	42.9 MB/s	49.3% (-6.5%)	66.7%	50.7%	0.576
GBM ($h = 1$)	-	93.8%	91.1%	37.9 MB/s	49.4% (-6.3%)	76.8%	51.9%	0.619
OPT	-	100%	100%	30.4 MB/s	52.7%	100%	100%	1