LHD: Improving Cache Hit Rate by Maximizing Hit Density

Nathan Beckmann
Carnegie Mellon University
beckmann@cs.cmu.edu

Haoxian Chen
University of Pennsylvania
hxchen@seas.upenn.edu

Asaf Cidon
Stanford University/Barracuda Networks
asaf@cidon.com

Abstract

Cloud application performance is heavily reliant on the hit rate of datacenter key-value caches. Key-value caches typically use least recently used (LRU) as their eviction policy, but LRU's hit rate is far from optimal under real workloads. Prior research has proposed many eviction policies that improve on LRU, but these policies make restrictive assumptions that hurt their hit rate, and they can be difficult to implement efficiently.

We introduce least hit density (LHD), a novel eviction policy for key-value caches. LHD predicts each object's expected hits-per-space-consumed (*hit density*), filtering objects that contribute little to the cache's hit rate. Unlike prior eviction policies, LHD does not rely on heuristics, but rather rigorously models objects' behavior using conditional probability to adapt its behavior in real time.

To make LHD practical, we design and implement RankCache, an efficient key-value cache based on memcached. We evaluate RankCache and LHD on commercial memcached and enterprise storage traces, where LHD consistently achieves better hit rates than prior policies. LHD requires much less space than prior policies to match their hit rate, on average 8× less than LRU and 2–3× less than recently proposed policies. Moreover, RankCache requires no synchronization in the common case, improving request throughput at 16 threads by 8× over LRU and by 2× over CLOCK.

1 Introduction

The hit rate of distributed, in-memory key-value caches is a key determinant of the end-to-end performance of cloud applications. Web application servers typically send requests to the cache cluster over the network, with latencies of about 100 µs, before querying a much slower database, with latencies of about 10 ms. Small increases in cache hit rate have an outsize impact on application performance. For example, increasing hit rate by just 1% from 98% to 99% halves the number of requests to the database. With the latency numbers used above, this decreases the mean service time from 210 µs to 110 µs (nearly 2×) and, importantly for cloud applications, halves the tail of long-latency requests [21].

To increase cache hit rate, cloud providers typically scale the number of servers and thus total cache capacity [37]. For example, Facebook dedicates tens of thousands of continuously running servers to caching. However, adding servers is not tenable in the long run, since hit rate increases logarithmically as a function of cache capacity [3, 13, 20]. Prohibitively large amounts of memory are needed to significantly impact hit rates.

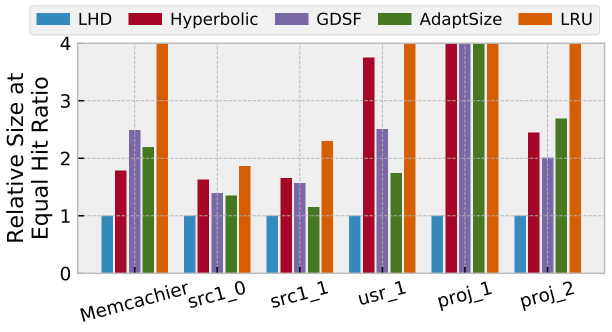


Figure 1: Relative cache size needed to match LHD's hit rate on different traces. LHD requires roughly one-fourth of LRU's capacity, and roughly half of that of prior eviction policies.

This paper argues that improving the eviction policy is much more effective, and that there is significant room to improve cache hit rates. Popular key-value caches (e.g., memcached, Redis) use least recently used (LRU) or variants of LRU as their eviction policy. However, LRU is far from optimal for key-value cache workloads because: *(i)* LRU's performance suffers when the workload has variable object sizes, and *(ii)* common access patterns expose pathologies in LRU, leading to poor hit rate.

These shortcomings of LRU are well documented, and prior work has proposed many eviction policies that improve on LRU [4, 14, 16, 25, 35, 38, 40]. However, these policies are not widely adopted because they typically require extensive parameter tuning, which makes their performance unreliable, and globally synchronized state, which hurts their request throughput. Indeed, to achieve acceptable throughput, some systems use eviction policies such as CLOCK or FIFO that sacrifice hit rate to reduce synchronization [22, 33, 34].

More fundamentally, prior policies make assumptions that do not hold for many workloads, hurting their hit rate. For example, most policies prefer recently used objects, all else equal. This is reasonable—such objects are often valuable—, but workloads often violate this assumption. Prior policies handle the resulting pathologies by adding new mechanisms. For example, ARC [35] adds a second LRU list for newly admitted objects, and AdaptSize [9] adds a probabilistic filter for large objects.

We take a different approach. Rather than augmenting or recombining traditional heuristics, we seek a new mechanism that just "does the right thing". The key motivating question for this paper is: *What would we want to know about objects to make good caching decisions, independent of workload?*

Our answer is a metric we call *hit density*, which measures how much an object is expected to contribute to the cache's hit rate. We infer each object's hit density from what we know about it (e.g., its age or size) and then evict the object with least hit density (LHD). Finally, we present an efficient and straightforward implementation of LHD on memcached called RankCache.

1.1 Contributions

We introduce ***hit density***, an intuitive, workload-agnostic metric for ranking objects during eviction. We arrive at hit density from first principles, without any assumptions about how workloads tend to reference objects.

Least hit density (LHD) is an eviction policy based on hit density. LHD monitors objects online and uses conditional probability to predict their likely behavior. LHD draws on many different object features (e.g., age, frequency, application id, and size), and easily supports others. Dynamic ranking enables LHD to adapt its eviction strategy to different application workloads over time without any hand tuning. For example, on a certain workload, LHD may initially approximate LRU, then switch to most recently used (MRU), least frequently used (LFU), or a combination thereof.

RankCache is a key-value cache based on memcached that efficiently implements LHD (and other policies). RankCache supports arbitrary *ranking functions*, making policies like LHD practical. RankCache approximates a true global ranking while requiring no synchronization in the common case, and adds little implementation complexity over existing LRU caches. RankCache thus avoids the unattractive tradeoff in prior systems between hit rate and request throughput, showing it is possible to achieve the best of both worlds.

1.2 Summary of Results

We evaluate LHD on a weeklong commercial memcached trace from Memcachier [36] and storage traces from Microsoft Research [48]. LHD significantly improves hit rate prior policies—e.g., reducing misses by half vs. LRU and one-quarter vs. recent policies—and also avoids pathologies such as performance cliffs that afflict prior policies. Fig. 1 shows the cache size (i.e., number of caching servers) required to achieve the same hit rate as LHD at 256 MB on Memcachier and 64 GB on Microsoft traces. LHD requires much less space than prior eviction policies, saving the cost of thousands of servers in a modern datacenter. On average, LHD needs 8× less space than LRU, 2.4× less than GDSF [4], 2.5× less than Hyperbolic [11], and 2.9× less than AdaptSize [9]. Finally, at 16 threads, RankCache achieves 16× higher throughput than list-based LRU and, at 90% hit rate, 2× higher throughput than CLOCK.

2 Background and Motivation

We identify two main opportunities to improve hit rate beyond existing eviction policies. First, prior policies make implicit assumptions about workload behavior that hurt their hit rate when they do not hold. Second, prior policies rely on implementation primitives that unnecessarily limit their design. We avoid these pitfalls by going back to first principles to design LHD, and then build RankCache to realize it practically.

2.1 Implicit assumptions in eviction policies

Eviction policies show up in many contexts, e.g., OS page management, database buffer management, web proxies, and processors. LRU is widely used because it is intuitive, simple to implement, performs reasonably well, and has some worst-case guarantees [12, 47].

However, LRU also has common pathologies that hurt its performance. LRU uses only *recency*, or how long it has been since an object was last referenced, to decide which object to evict. In other words, LRU assumes that recently used objects are always more valuable. But common access patterns like scans (e.g., AB. . . Z AB. . . Z . . .) violate this assumption. As a result, LRU caches are often polluted by infrequently accessed objects that stream through the cache without reuse.

Prior eviction policies improve on LRU in many different ways. Nearly all policies augment recency with additional mechanisms that fix its worst pathologies. For example, ARC [35] uses two LRU lists to distinguish newly admitted objects and limit pollution from infrequently accessed objects. Similarly, AdaptSize [9] adds a probabilistic filter in front of an LRU list to limit pollution from large objects. Several recent policies split accesses across multiple LRU lists to eliminate performance cliffs [6, 18, 51] or to allocate space across objects of different sizes [10, 17, 18, 37, 41, 43, 49].

All of these policies use LRU lists as a core mechanism, and hence retain recency as built-in assumption. Moreover, their added mechanisms can introduce new assumptions and pathologies. For example, ARC assumes that frequently accessed objects are more valuable by placing them in a separate LRU list from newly admitted objects and preferring to evict newly admitted objects. This is often an improvement on LRU, but can behave pathologically.

Other policies abandon lists and rank objects using a heuristic function. GDSF [4] is a representative example. When an object is referenced, GDSF assigns its rank using its frequency (reference count) and global value L:

$$\text{GDSF Rank} = \frac{\text{Frequency}}{\text{Size}} + L \tag{1}$$

On a miss, GDSF evicts the cached object with the lowest rank and then updates L to this victim's rank. As a result, L increases over time so that recently used objects have higher rank. GDSF thus orders objects according to some combination of recency, frequency, and size. While it is intuitive that each of these factors should play some role, it is not obvious why GDSF combines them in this particular formula. Workloads vary widely (Sec. 3.5), so no factor will be most effective in general. Eq. 1 makes implicit assumptions about how important each factor will be, and these assumptions will not hold across all workloads. Indeed, subsequent work [16, 27] added weighting parameters to Eq. 1 to tune GDSF for different workloads.

Hence, while prior eviction policies have significantly improved hit rates, they still make implicit assumptions that lead to sub-optimal decisions. Of course, all online policies must make some workload assumptions (e.g., adversarial workloads could change their behavior arbitrarily [47]), but these should be minimized. We believe the solution is *not* to add yet more mechanisms, as doing so quickly becomes unwieldy and requires yet more assumptions to choose among mechanisms. Instead, our goal is to find a new mechanism that leads to good eviction decisions across a wide range of workloads.

2.2 Implementation of eviction policies

Key-value caches, such as memcached [23] and Redis [1], are deployed on clusters of commodity servers, typically based on DRAM for low latency access. Since DRAM caches have a much lower latency than the backend database, the main determinant of end-to-end request latency is cache hit rate [19, 37].

Request throughput: However, key-value caches must also maintain high request throughput, and the eviction policy can significantly impact throughput. Table 1 summarizes the eviction policies used by several popular and recently proposed key-value caches.

Key-Value Cache	Allocation	Eviction Policy
memcached [23]	Slab	LRU
Redis [1]	jemalloc	LRU
Memshare [19]	Log	LRU
Hyperbolic [11]	jemalloc	GD
Cliffhanger [18]	Slab	LRU
GD-Wheel [32]	Slab	GD
MICA [34]	Log	≈LRU
MemC3 [22]	Slab	≈LRU

Table 1: Allocation and eviction strategies of key-value caches. GD-Wheel and Hyperbolic's policy is based on GreedyDual [53]. We discuss a variant of this policy (GDSF) in Sec. 2.1.

Most key-value caches use LRU because it is simple and efficient, requiring $O(1)$ operations for admission, update, and eviction. Since naïve LRU lists require global synchronization, most key-value caches in fact use approximations of LRU, like CLOCK and FIFO, that eliminate synchronization except during evictions [22, 33, 34]. Policies that use more complex ranking (e.g., GDSF) pay a price in throughput to maintain an ordered ranking (e.g., $O(\log N)$ operations for a min-heap) and to synchronize other global state (e.g., L in Eq. 1).

For this reason, most prior policies restrict themselves to well-understood primitives, like LRU lists, that have standard, high-performance implementations. Unfortunately, these implementation primitives restrict the design of eviction policies, preventing policies from retaining the most valuable objects. List-based policies are limited to deciding how the lists are connected and and which objects to admit to which list. Similarly, to maintain data structure invariants, policies that use min-heaps (e.g., GDSF) can change ranks only when an object is referenced, limiting their dynamism.

We ignore such implementation restrictions when designing LHD (Sec. 3), and consider how to implement the resulting policy efficiently in later sections (Secs. 4 & 5).

Memory management: With objects of highly variable size, another challenge is memory fragmentation. Key-value caches use several memory allocation techniques (Table 1). This paper focuses on the most common one, slab allocation. In slab allocation, memory is divided into fixed 1 MB *slabs*. Each slab can store objects of a particular size range. For example, a slab can store objects between 0–64 B, 65–128 B, or 129–256 B, etc. Each object size range is called a *slab class*.

The advantages of slab allocation are its performance and bounded fragmentation. New objects always replace another object of the same slab class, requiring only a single eviction to make space. Since objects are always inserted into their appropriate slab classes, there is no external fragmentation, and internal fragmentation is bounded. The disadvantage is that the eviction policy is implemented on each slab class separately, which can hurt overall hit rate when, e.g., the workload shifts from larger to smaller objects.

Other key-value caches take different approaches. However, non-copying allocators [1] suffer from fragmentation [42], and log-structured memory [19, 34, 42] requires a garbage collector that increases memory bandwidth and CPU consumption [19]. RankCache uses slab-based allocation due to its performance and bounded fragmentation, but this is not fundamental, and LHD could be implemented on other memory allocators.

3 Replacement by Least Hit Density (LHD)

We propose a new replacement policy, LHD, that dynamically predicts each object's *expected hits per space consumed*, or *hit density*, and evicts the object with the lowest. By filtering out objects that contribute little to the cache's hit rate, LHD gradually increases the average hit rate. Critically, LHD avoids ad hoc heuristics, and instead ranks objects by rigorously modeling their behavior using conditional probability. This section presents LHD and shows its potential in an idealized setting. The following sections will present RankCache, a high-performance implementation of LHD.

3.1 Predicting an object's hit density

Our key insight is that policies must account for both *(i)* the probability an object will hit in its lifetime; and *(ii)* the resources it will consume. LHD uses the following function to rank objects:

$$\boxed{\text{Hit density} = \frac{\text{Hit probability}}{\text{Object size} \times \text{Expected time in cache}}} \quad (2)$$

Eq. 2 measures an object's contribution to the cache's hit rate (in units of hits per byte-access). We first provide an example that illustrates how this metric adapts to real-world applications, and then show how we derived it.

3.2 LHD on an example application

To demonstrate LHD's advantages, consider an example application that scans repeatedly over a few objects, and accesses many other objects with Zipf-like popularity distribution. This could be, for example, the common media for a web page (scanning) plus user-specific content (Zipf). Suppose the cache can fit the common media and some of the most popular user objects. In this case, each scanned object is accessed frequently (once per page load for all users), whereas each Zipf-like object is accessed much less frequently (only for the same user). The cache should ideally therefore *keep the scanned objects and evict the Zipf-like objects when necessary*.

Fig. 2a illustrates this application's access pattern, namely the distribution of time (measured in accesses) between references to the same object. Scanned objects produce a characteristic peak around a single reference time, as all are accessed together at once. Zipf-like objects yield a long tail of reference times. Note that in this example 70% of references are to the Zipf-like objects and 30% to scanned objects, but the long tail of popularity in Zipf-like objects leads to a low reference probability in Fig. 2a.

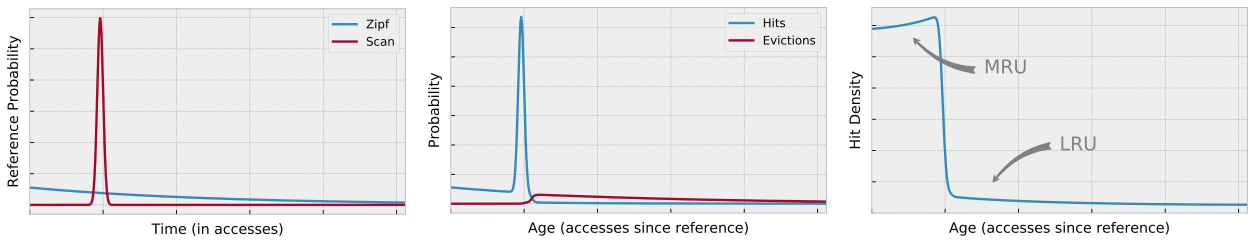
Fig. 2b illustrates LHD's behavior on this example application, showing the distribution of hits and evictions vs. an object's *age*. Age is the number of accesses since an object was last referenced. For example, if an object enters the cache at access T, hits at accesses $T + 4$ and $T + 6$, and is evicted at access $T + 12$, then it has two hits at age 4 and 2 and is evicted at age 6 (each reference resets age to zero). Fig. 2b shows that LHD keeps the scanned objects and popular Zipf references, as desired.

LHD does not know whether an object is a scanned object or a Zipf-like object until ages pass the scanning peak. It must conservatively protect all objects until this age, and all references at ages less than the peak therefore result in hits. LHD begins to evict objects immediately after the peak, since it is only at this point it knows that any remaining objects must be Zipf-like objects, and it can safely evict them.

Finally, Fig. 2c shows how LHD achieves these outcomes. It plots the predicted hit density for objects of different ages. The hit density is high up until the scanning peak, because LHD predicts that objects are potentially one of the scanned objects, and might hit quickly. It drops after the scanning peak because it learns they are Zipf objects and therefore unlikely to hit quickly.

Discussion: Given that LHD evicts the object with the lowest predicted hit density, what is its emergent behavior on this example? The object ages with the lowest predicted hit density are those that have aged past the scanning peak. These are guaranteed to be Zipf-like objects, and their hit density decreases with age, since their implied popularity decreases the longer they have not been referenced. LHD thus evicts older objects; i.e., LRU.

However, if no objects older than the scanning peak are available, LHD will prefer to evict the *youngest* objects, since these have the lowest hit density. This is



(a) Summary of access pattern. **(b)** Distribution of hits and evictions. **(c)** Predicted hit density.

Figure 2: How LHD performs on an application that scans over 30% of objects and Zipf over the remaining 70%.

the most recently used (MRU) eviction policy, or *anti-LRU*. MRU is the correct policy to adopt in this example because *(i)* without more information, LHD cannot distinguish between scanning and Zipf-like objects in this age range, and *(ii)* MRU guarantees that some fraction of the scanning objects will survive long enough to hit. Because scanning objects are by far the most frequently accessed objects (Fig. 2a), keeping as many scanned objects as possible maximizes the cache's hit rate, even if that means evicting some popular Zipf-like objects.

Overall, then, LHD prefers to evict objects older than the scanning peak and evicts LRU among these objects, and otherwise evicts MRU among younger objects. This policy caches as many of the scanning objects as possible, and is the best strictly age-based policy for this application. LHD adopts this policy automatically based on the cache's observed behavior, without any pre-tuning required. By adoping MRU for young objects, LHD avoids the potential performance cliff that recency suffers on scanning patterns. We see this behavior on several traces (Sec. 3.5), where LHD significantly outperforms prior policies, nearly all of which assume recency.

3.3 Analysis and derivation

To see how we derived hit density, consider the cache in Fig. 3. Cache space is shown vertically, and time increases from left to right. (Throughout this paper, time is measured in accesses, not wall-clock time.) The figure shows how cache space is used over time: each block represents an object, with each reference or eviction starting a new block. Each block thus represents a single *object lifetime*, i.e., the idle time an object spends in the cache between hits or eviction. Additionally, each block is colored green or red, indicating whether it ends in a hit or eviction, respectively.

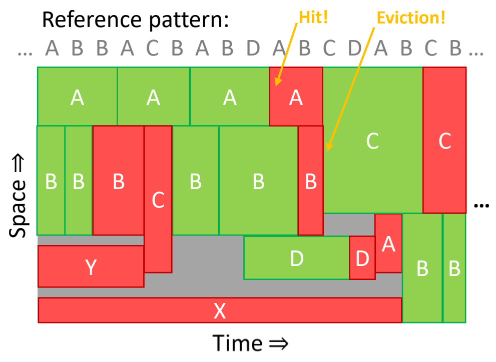


Figure 3: Illustration of a cache over time. Each block depicts a single object's lifetime. Lifetimes that end in hits are shown in green, evictions in red. Block size illustrates resources consumed by an object; hit density is inversely proportional to block size.

Fig. 3 illustrates the challenge replacement policies face: they want to maximize hits given limited resources. In other words, they want to fit as many green blocks into the figure as possible. Each object takes up resources proportional to *both* its size (block height) and the time it spends in the cache (block width). Hence, the replacement policy wants to keep *small objects that hit quickly*.

This illustration leads directly to hit density. Integrating uniformly across the entire figure, each green block contributes 1 hit spread across its entire block. That is, resources in the green blocks contribute hits at a rate of: 1 hit/(size $\times$ lifetime). Likewise, lifetimes that end in eviction (or space lost to fragmentation) contribute zero hits. Thus, if there are N hits and M evictions, and if object i has size S_i bytes and spends L_i accesses in the cache, then the cache's overall hit density is:

$$\frac{\sum_{\text{Lifetimes}} \overbrace{1+1+...+1}^{\text{Hits}} \quad + \quad \overbrace{0+0+...+0}^{\text{Evictions}}}{\sum_{\text{Lifetimes}} \underbrace{S_1\times L_1+...+S_N\times L_N}_{\text{Hit resources}} + \underbrace{S_1\times L_1+...+S_M\times L_M}_{\text{Eviction resources}}}$$

The cache's overall hit density is directly proportional to its hit rate, so maximizing hit density also maximizes the hit rate. Furthermore, it follows from basic arithmetic that replacing an object with one of higher density will increase the cache's overall hit density.[1]

LHD's challenge is to predict an object's hit density, without knowing whether it will result in a hit or eviction, nor how long it will spend in the cache.

Modeling object behavior: To rank objects, LHD must compute their hit probability and the expected time they will spend in the cache. (We assume that an object's size is known and does not change.) LHD infers these quantities in real-time using probability distributions. Specifically, LHD uses distributions of hit and eviction age.

The simplest way to infer hit density is from an object's age. Let the random variables H and L give hit and lifetime age; that is, $\mathbb{P}[H=a]$ is the probability that an object hits at age a, and $\mathbb{P}[L=a]$ is the probability that an object is hit *or* evicted at age a. Now consider an object of age a. Since the object has reached age a, we know it cannot hit or be evicted at any age earlier than a. Its hit probability conditioned on age a is:

$$\text{Hit probability} = \mathbb{P}[\text{hit}|\text{age } a] = \frac{\mathbb{P}[H>a]}{\mathbb{P}[L>a]} \tag{3}$$

Similarly, its expected remaining lifetime[2] is:

$$\text{Lifetime} = \mathbb{E}[L-a|\text{age } a] = \frac{\sum_{x=1}^{\infty} x\cdot\mathbb{P}[L=a+x]}{\mathbb{P}[L>a]} \tag{4}$$

Altogether, the object's hit density at age a is:

$$\text{Hit density}_{\text{age}}(a) = \frac{\sum_{x=1}^{\infty}\mathbb{P}[H=a+x]}{\text{Size}\cdot\left(\sum_{x=1}^{\infty} x\cdot\mathbb{P}[L=a+x]\right)} \tag{5}$$

[1]Specifically, if $a/b > c/d$, then $(a+c)/(b+d) > c/d$.
[2]We consider the *remaining* lifetime to avoid the sunk-cost fallacy.

3.4 Using classification to improve predictions

One nice property of LHD is that it intuitively and rigorously incorporates additional information about objects. Since LHD is based on conditional probability, we can simply condition the hit and eviction age distributions on the additional information. For example, to incorporate reference frequency, we count how many times each object has been referenced and gather separate hit and eviction age distributions for each reference count. That is, if an object that has been referenced twice is evicted, LHD updates only the eviction age distribution *of objects that have been referenced twice*, and leaves the other distributions unchanged. LHD then predicts an object's hit density using the appropriate distribution during ranking.

To generalize, we say that an object belongs to an equivalence class c; e.g., c could be all objects that have been referenced twice. LHD predict this object's hit density as:

$$\text{Hit density}(a, c) = \frac{\sum_{x=1}^{\infty} \mathbb{P}[H = a + x|c]}{\text{Size} \cdot \left(\sum_{x=1}^{\infty} x \cdot \mathbb{P}[L = a+x|c]\right)} \tag{6}$$

where $\mathbb{P}[H = a|c]$ and $\mathbb{P}[L = a|c]$ are the conditional hit and lifetime age distributions for class c.

3.5 Idealized evaluation

To demonstrate LHD's potential, we simulate an idealized implementation of LHD that globally ranks objects. Our figure of merit is the cache's miss ratio, i.e., the fraction of requests resulting in misses. To see how miss ratio affects larger system tradeoffs, we consider the cache size needed to achieve equal miss ratios.

Methodology: Unfortunately, we are unaware of a public trace of large-scale key-value caches. Instead, we evaluate two sets of traces: *(i)* a weeklong, commercial trace provided by Memcachier [36] containing requests from hundreds of applications, and *(ii)* block traces from Microsoft Research [48]. Neither trace is ideal, but together we believe they represent a wide range of relevant behaviors. Memcachier provides caching-as-a-service and serves objects from a few bytes to 1 MB (median: 100 B); this variability is a common feature of key-value caches [5, 22]. However, many of its customers massively overprovision resources, forcing us to consider scaled-down cache sizes to replicate miss ratios seen in larger deployments [37]. Fortunately, scaled-down caches are known to be good models of behavior at larger sizes [6, 30, 51]. Meanwhile, the Microsoft Research traces let us study larger objects (median: 32 KB) and cache sizes. However, its object sizes are much less variable, and block trace workloads may differ from key-value workloads.

We evaluate 512 M requests from each trace, ignoring the first 128 M to warm up the cache. For the shorter traces, we replay the trace if it terminates to equalize trace length across results. All included traces are much longer than LHD's reconfiguration interval (see Sec. 5).

Since it is too expensive to compute Eq. 2 for every object on each eviction, evictions instead sample 64 random objects, as described in Sec. 4.1. LHD monitors hit and eviction distributions online and, to escape local optima, devotes a small amount of space (1%) to "explorer" objects that are not evicted until a very large age.

What is the best LHD configuration?: LHD uses an object's age to predict its hit density. We also consider two additional object features to improve LHD's predictions: an object's last hit age and its app id. LHD_{APP} classifies objects by hashing their app id into one of N classes (mapping several apps into each class limits overheads). We only use LHD_{APP} on the Memcachier trace, since the block traces lack app ids. $\text{LHD}_{\text{LAST HIT}}$ classifies objects by the age of their last hit, analogous to LRU-K [38], broken into N classes spaced at powers of 2 up to the maximum age. (E.g., with max age = 64 K and $N = 4$, classes are given by last hit age in $0 < 16\,\text{K} < 32\,\text{K} < 64\,\text{K} < \infty$).

We swept configurations over the Memcachier and Microsoft traces and found that both app and last-hit classification reduce misses. Furthermore, these improvements come with relatively few classes, after which classification yields diminishing returns. Based on these results, we configure LHD to classify by last hit age (16 classes) and application id (16 classes). We refer to this configuration as LHD+ for the remainder of the paper.

How does LHD+ compare with other policies?: Fig. 4 shows the miss ratio across many cache sizes for LHD+, LRU, and three prior policies: GDSF [4, 16], AdaptSize [9], and Hyperbolic [11]. GDSF and Hyperbolic use different ranking functions based on object recency, frequency, and size (e.g., Eq. 1). AdaptSize probabilistically admits objects to an LRU cache to avoid polluting the cache with large objects (Sec. 6). LHD+ achieves the best miss ratio across all cache sizes, outperforms LRU by a large margin, and outperforms Hyperbolic, GDSF, and AdaptSize, which perform differently across different traces. No prior policy is consistently close to LHD+'s hit ratio.

Moreover, Fig. 4 shows that LHD+ needs less space than these other policies to achieve the same miss ratio, sometimes substantially less. For example, on Memcachier, a 512 MB LHD+ cache matches the hit rate of a 768 MB Hyperbolic cache, a 1 GB GDSF, or a 1 GB AdaptSize cache, and LRU does not match the performance even with 2 GB. In other words, LRU requires more than 4× as many servers to match LHD+'s hit rate.

Averaged across all sizes, LHD+ incurs 45% fewer misses than LRU, 27% fewer than Hyperbolic and GDSF and 23% fewer than AdaptSize. Moreover, at the largest

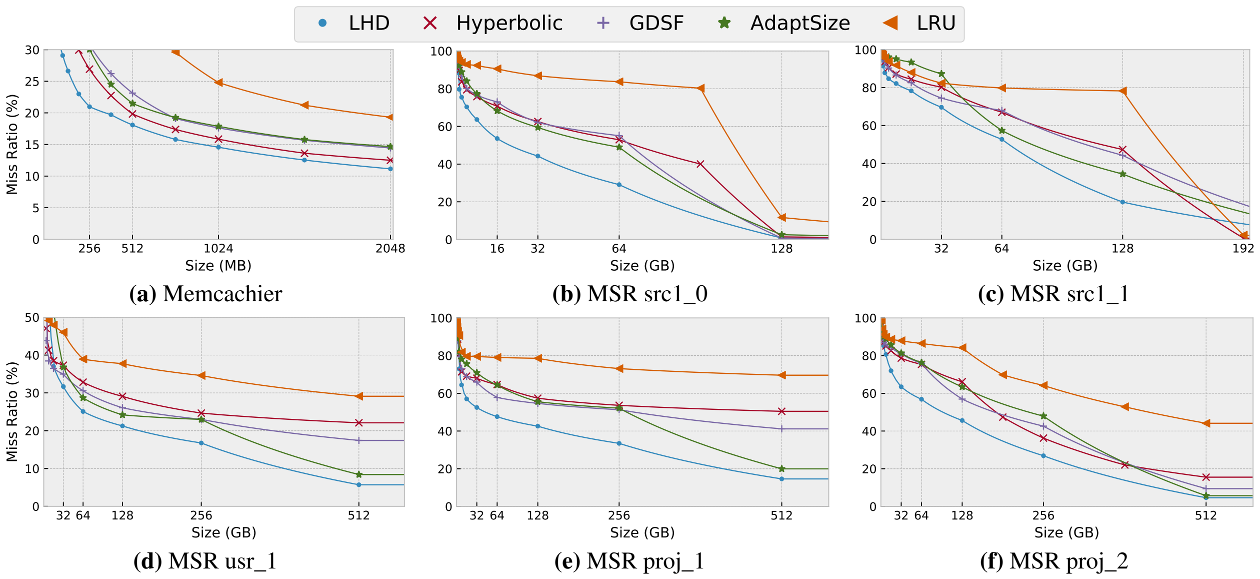


Figure 4: Miss ratio for LHD+ vs. prior policies over 512 M requests and cache sizes from 2 MB to 2 GB on Memcachier trace and from 128 MB to 512 GB on MSR traces. LHD+ consistently outperforms prior policies on all traces.

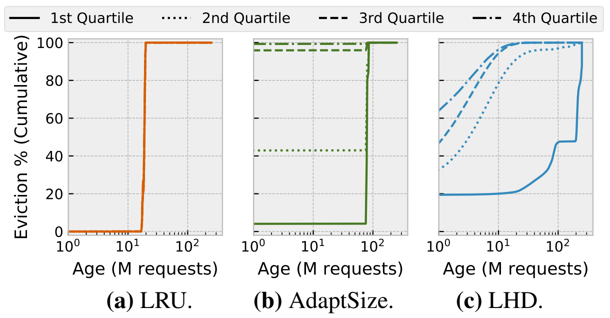


Figure 5: When policies evict objects, broken into quartiles by object size. LRU evicts all objects at roughly the same age, regardless of their size, wasting space on big objects. AdaptSize bypasses most large objects, losing some hits on these objects, while also ignoring object size after admission, still wasting space. LHD dynamically ranks objects to evict larger objects sooner, allocating space across all objects to maximize hits.

sizes, LHD+ incurs very few non-compulsory misses, showing it close to exhausting all possible hits.

Where do LHD+'s benefits come from?: LHD+'s dynamic ranking gives it the flexibility to evict the least valuable objects, without the restrictions or built-in assumptions of prior policies. To illustrate this, Fig. 5 compares when LRU, AdaptSize, and LHD evict objects on the Memcachier trace at 512 MB. Each line in the figure shows the cumulative distribution of eviction age for objects of different sizes; e.g., the solid line in each figure shows when the smallest quartile of objects are evicted.

LRU ignores object size and evicts all objects at roughly the same age. Because of this, LRU wastes space on large objects and must evict objects when they are relatively young (age$\approx$30 M), hurting its hit ratio. AdaptSize improves on LRU by bypassing most large objects so that admitted objects survive longer (age$\approx$75 M). This lets AdaptSize get more hits than LRU, at the cost of forgoing some hits to the bypassed objects. However, since AdaptSize evicts LRU after admission, it still wastes space on large, admitted objects.

LHD+ is not limited in this way. It can admit all objects and evict larger objects sooner. This earns LHD+ more hits on large objects than AdaptSize, since they are not bypassed, and lets small objects survive longer than AdaptSize (age$\approx$200 M), getting even more hits.

Finally, although many applications are recency-friendly, several applications in the Memcachier trace as well as most of the Microsoft Research traces show that this is not true in general. As a result, policies that include recency (i.e., nearly all policies, including GDSF, Hyperbolic, and AdaptSize), suffer from pathologies like performance cliffs [6, 18]. For example, LRU, GDSF, and Hyperbolic suffer a cliff in src1_0 at 96 MB and proj_2 at 128 MB. LHD avoids these cliffs and achieves the highest performance of all policies (see Sec. 6).

4 RankCache Design

LHD improves hit rates, but implementability and request throughput also matter in practice. We design RankCache to efficiently support arbitrary ranking functions, including hit density (Eq. 5). The challenge is that, with arbitrary ranking functions, the rank-order of objects can change constantly. A naïve implementation would scan all cached objects to find the best victim for each eviction, but this is far too expensive. Alternatively, for some restricted ranking functions, prior work has used priority queues (i.e., min-heaps), but these queues require expensive global synchronization to keep the data structure consistent [9].

RankCache solves these problems by *approximating* a global ranking, avoiding any synchronization in the common case. RankCache does not require synchronization even for evictions, unlike prior high-performance caching systems [22, 34], letting it achieve high request throughput with non-negligible miss rates.

4.1 Lifetime of an eviction in LHD

Ranks in LHD constantly change, and this dynamism is critical for LHD, since it is how LHD adapts its policy to the access pattern. However, it would be very expensive to compute Eq. 5 for all objects on every cache miss. Instead, two key techniques make LHD practical: *(i)* precomputation and *(ii)* sampling. Fig. 6 shows the steps of an eviction in RankCache, discussed below.

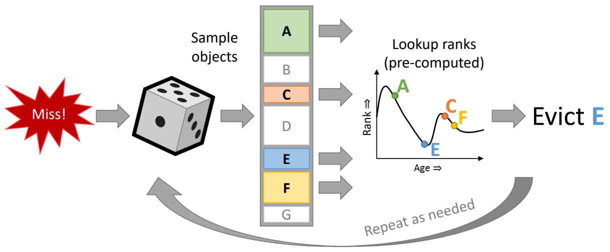


Figure 6: Steps for an eviction in RankCache. First, randomly sample objects, then lookup their precomputed rank and evict the object with the worst rank.

Selecting a victim: RankCache randomly samples cached objects and evicts the object with the worst rank (i.e., lowest hit density) in the sample. With a large enough sample, the evicted object will have eviction priority close to the global maximum, approximating a perfect ranking. Sampling is an old idea in processor caches [44, 46], has been previously proposed for web proxies [39], and is used in some key-value caches [1, 11, 19]. Sampling is effective because the quality of a random sample depends on its size, *not the size of the underlying population* (i.e., number of cached objects). Sampling therefore lets RankCache implement dynamic ranking functions in constant time with respect to the number of cached objects.

Sampling eliminates synchronization: Sampling makes cache management concurrent. Both linked lists and priority queues have to serialize GET and SET operations to maintain a consistent data structure. For example, in memcached, where LRU is implemented by a linked list, every cache hit promotes the hit object to the head of the list. On every eviction, the system first evicts the object from the tail of the list, and then inserts the new object at the head of the list. These operations serialize all GETs and SETs in memcached.

To avoid this problem, systems commonly sacrifice hit ratio: by default, memcached only promotes objects if they are older than one minute; other systems use CLOCK [22] or FIFO [33], which do not require global updates on a cache hit. However, *these policies still serialize all evictions*.

Sampling, on the other hand, allows each item to update its metadata (e.g., reference timestamp) independently on a cache hit, and evictions can happen concurrently as well except when two threads select the same victim. To handle these rare races, RankCache uses memcached's built-in versioning and optimistic concurrency: evicting threads sample and compare objects in parallel, then lock the victim and check if its version has changed since sampling. If it has, then the eviction process is restarted. Thus, although sampling takes more operations per eviction, it increases concurrency, letting RankCache achieve higher request throughput than CLOCK/FIFO under high load.

Few samples are needed: Fig. 7 shows the effect of sampling on miss ratio going from associativity (i.e., sample size) of one to 128. With only one sample, the cache randomly replaces objects, and all policies perform the same. As associativity increases, the policies quickly diverge. We include a sampling-based variant of LRU, where an object's rank equals its age. LRU, Hyperbolic, and LHD+ all quickly reach diminishing returns, around associativity of 32. At this point, true LRU and sampling-based LRU achieve identical hit ratios.

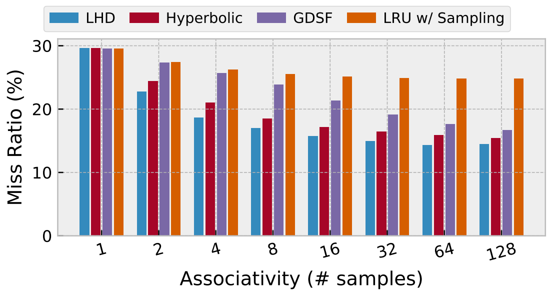


Figure 7: Miss ratios at different associativities.

Since sampling happens at each eviction, lower associativity is highly desirable from a throughput and latency perspective. Therefore, RankCache uses an associativity of 64.

We observe that GDSF is much more sensitive to associativity, since each replacement in GDSF updates global state (L, see Sec. 2.1). In fact, GDSF still has not converged at 128 samples. GDSF's sensitivity to associativity makes it unattractive for key-value caches, since it needs expensive data structures to accurately track its state (Fig. 10). Hyperbolic [11] uses a different ranking function without global state to avoid this problem.

Precomputation: RankCache precomputes object ranks so that, given an object, its rank can be quickly found by indexing a table. In the earlier example, RankCache would precompute Fig. 2c so that ranks can be looked up directly from an object's age. With LHD, RankCache periodically (e.g., every one million accesses) recomputes its ranks to remain responsive to changes in application behavior. This approach is effective since application behavior is stable over short time periods, changing much more slowly than the ranks themselves fluctuate. Moreover, Eq. 5 can be computed efficiently in linear time [8], and RankCache configures the maximum age to keep overheads low (Sec. 5).

4.2 Approximating global rankings with slabs

RankCache uses slab allocation to manage memory because it ensures that our system achieves predictable $O(1)$ insertion and eviction performance, it does not require a garbage collector, and it has no external fragmentation. However, in slab allocation, each slab class evicts objects independently. Therefore, another design challenge is to approximate a global ranking when each slab allocation implements its own eviction policy.

Similar to memcached, when new objects enter the cache, RankCache evicts the lowest ranked object from the same slab class. RankCache approximates a global ranking of all objects by periodically rebalancing slabs among slab classes. It is well-known that LRU effectively evicts objects once they reach a characteristic age that depends on the cache size and access pattern [15]. This fact has been used to balance slabs across slab classes to approximate global LRU by equalizing eviction age across slab classes [37]. RankCache generalizes this insight, such that caches essentially evict objects once they reach a *characteristic rank*, rather than age, that depends on the cache size and access pattern.

Algorithm: In order to measure the average eviction rank, RankCache records the cumulative rank of evicted objects and the number of evictions. It then periodically moves a slab from the slab class that has the highest average victim rank to that with the lowest victim rank.

However, we found that some slab classes rarely evict objects. Without up-to-date information about their average victim rank, RankCache was unable to rebalance slabs away from them to other slab classes. We solved this problem by performing one "fake eviction" (i.e., sampling and ranking) for each slab class during rebalancing. By also averaging victim ranks across several decisions, this mechanism gives RankCache enough information to effectively approximate a global ranking.

RankCache decides whether it needs to rebalance slabs every 500 K accesses. We find that this is sufficient to converge to the global ranking on our traces, and more frequent rebalancing is undesirable because it has a cost: when a 1 MB slab is moved between slab classes, 1 MB of objects are evicted from the original slab class.

Evaluation: Fig. 8 shows the effect of rebalancing slabs in RankCache. It graphs the distribution of victim rank for several different implementations, with each slab class shown in a different color. The right-hand figure shows RankCache with sampling-based LRU, and the left shows RankCache with LHD+. An idealized, global policy has victim rank tightly distributed around a single peak—this demonstrates the accuracy of our characteristic eviction rank model. Without rebalancing, each slab evicts objects around a different victim rank, and is far from the global policy. With rebalancing, the victim ranks are much more tightly distributed, and we find this is sufficient to approximate the global policy.

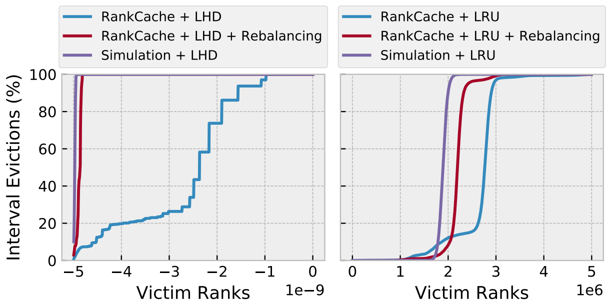


Figure 8: Distribution of victim rank for slab allocation policies with and without rebalancing vs. true global policy. LHD+ is on the left, LRU on the right.

5 RankCache Implementation

We implemented RankCache, including its LHD ranking function, on top of memcached [23]. RankCache is backwards compatible with the memcached protocol and is a fairly lightweight change to memcached v1.4.33.

The key insight behind RankCache's efficient implementation is that, by design, RankCache is an approximate scheme (Sec. 4). We can therefore tolerate loosely synchronized events and approximate aging information. Moreover, RankCache does not modify memcached's memory allocator, so it leverages existing functionality for events that require careful synchronization (e.g., moving slabs).

Aging: RankCache tracks time through the total number of accesses to the cache. Ages are *coarsened* in large increments of *COARSENESS* accesses, up to a *MAX_AGE*. *COARSENESS* and *MAX_AGE* are chosen to stay within a specified error tolerance (see appendix); in practice, coarsening introduces no detectable change in miss ratio or throughput for reasonable error tolerances (e.g., 1%).

Conceptually, there is a global timestamp, but for performance we implement distributed, fuzzy counters. Each server thread maintains a thread-local access count, and atomic-increments the global timestamp periodically when its local counter reaches *COARSENESS*.

RankCache must track the age of objects to compute their rank, which it does by adding a 4 B timestamp to the object metadata. During ranking, RankCache computes an object's coarsened age by subtracting the object timestamp from the global timestamp.

Ranking: RankCache adds tables to store the ranks of different objects. It stores ranks up to *MAX_AGE* per class, each rank a 4 B floating-point value. With 256 classes (Sec. 3), this is 6.4 MB total overhead. Ranks require no synchronization, since they are read-only between reconfigurations, and have a single writer (see below). We tolerate races as they are infrequently updated.

Monitoring behavior: RankCache monitors the distribution of hit and eviction age by maintaining histograms of hits and evictions. RankCache increments the appropriate counter upon each access, depending on whether it was a hit or eviction and the object's coarsened age. To reduce synchronization, these are also implemented as distributed, fuzzy counters, and are collected by the updating thread (see below). Counters are 4 B values; with 256 classes, hit and eviction counters together require 12.6 MB per thread.

Sampling: Upon each eviction, RankCache samples objects from within the same slab class by randomly generating indices and then computing the offset into the appropriate slab. Because objects are stored at regular offsets within each slab, this is inexpensive.

Efficient evictions: For workloads with non-negligible miss ratios, evictions are the rate-limiting step in RankCache. To make evictions efficient, RankCache uses two optimizations. First, rather than adding an object to a slab class's free list and then immediately claiming it, RankCache directly allocates the object within the same thread after it has been freed. This avoids unnecessary synchronization.

Second, RankCache places object metadata in a separate, contiguous memory region, called the *tags*. Tags are stored in the same order as objects in the slab class, making it easy to find an object from its metadata. Since slabs themselves are stored non-contiguously in memory, each object keeps a back pointer into the tags to find its metadata. Tags significantly improve spatial locality during evictions. Since sampling is random by design, without separate tags, RankCache suffers 64 (associativity) cache misses per eviction. Compact tags allow RankCache to sample 64 candidates with just 4 cache misses, a 16× improvement in locality.

Background tasks: Both updating ranks and rebalancing slabs are off the critical path of requests. They run as low-priority background threads and complete in a few milliseconds. Periodically (default: every 1 M accesses), RankCache aggregates histograms from each thread and recomputes ranks. First, RankCache averages histograms with prior values, using an exponential decay factor (default: 0.9). Then it computes LHD for each class in linear time, requiring two passes over the ages using an algorithm similar to [8]. Also periodically (every 500 K accesses), RankCache rebalances one slab from the slab with the highest eviction rank to the one with the lowest, as described in Sec. 4.2.

Across several orders of magnitude, the reconfiguration interval and exponential decay factor have minimal impact on hit rate. On the Memcachier trace, LHD+'s non-compulsory miss rate changes by 1% going from reconfiguring every 10 K to 10 M accesses, and the exponential decay factor shows even smaller impact when it is set between 0.1 and 0.99.

5.1 RankCache matches simulation

Going left-to-right, Fig. 9 compares the miss ratio over 512 M accesses on Memcachier at 1 GB for *(i)* stock memcached using true LRU within each slab class; RankCache using sampling-based LRU as its ranking function *(ii)* with and *(iii)* without rebalancing; RankCache using LHD+ *(iv)* with and *(v)* without rebalancing; and *(vi)* an idealized simulation of LHD+ with global ranking.

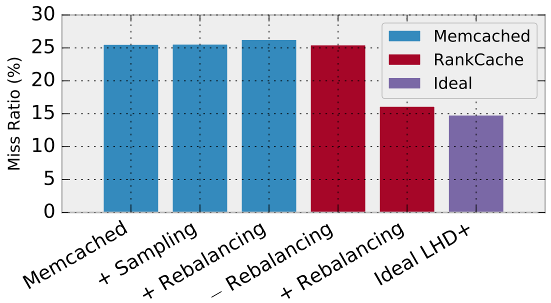


Figure 9: RankCache vs. unmodified memcached and idealized simulation. Rebalancing is necessary to improve miss ratio, and effectively approximates a global ranking.

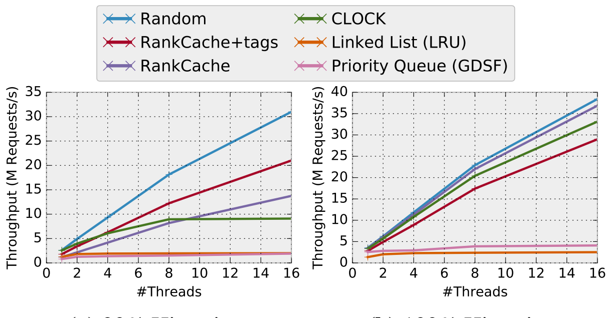
As the figure shows, RankCache with slab rebalancing closely matches the miss ratio of the idealized simulation, but without slab rebalancing it barely outperforms LRU. This is because LHD+ operating independently on each slab cannot effectively take into account object size, and hence on an LRU-friendly pattern performs similarly to LRU. The small degradation in hit ratio vs. idealized simulation is due to forced, random evictions during slab rebalancing.

5.2 RankCache with LHD+ achieves both high hit ratio and high performance

Methodology: To evaluate RankCache's performance, we stress request serving within RankCache itself by conducting experiments within a single server and bypassing the network. Each server thread pulls requests off a thread-local request list. We force all objects to have the same size to maximally stress synchronization in each policy. Prior work has explored techniques to optimize the network in key-value stores [22, 33, 34]; these topics are not our contribution.

We compare RankCache against list-based LRU, GDSF using a priority queue (min-heap), and CLOCK. These cover the main implementation primitives used in key-value caches (Sec. 2). We also compare against random evictions to show peak request throughput when the eviction policy does no work and maintains no state. (Random pays for its throughput by suffering many misses.)

Scalability: Fig. 10 plots the aggregate request throughput vs. number of server threads on a randomly generated trace with Zipfian object popularities. We present throughput at 90% and 100% hit ratio; the former represents a realistic deployment, the latter peak performance.



(a) 90% Hit ratio. **(b)** 100% Hit ratio.

Figure 10: RankCache's request throughput vs. server threads. RankCache's performance approaches that of random, and outperforms CLOCK with non-negligible miss ratio.

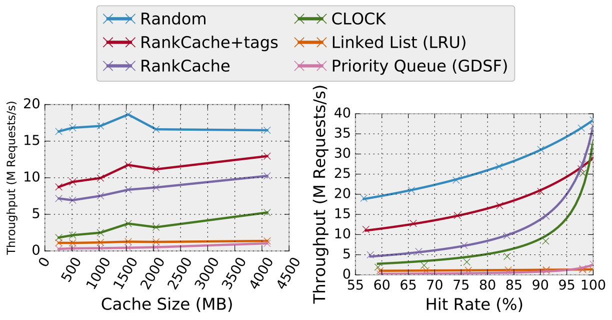
RankCache scales nearly as well as random because sampling avoids nearly all synchronization, whereas LRU and GDSF barely scale because they serialize all operations. Similarly, CLOCK performs well at 100% hit ratio, but serializes evictions and underperforms RankCache with 10% miss ratio. Finally, using separate tags in RankCache lowers throughput with a 100% hit ratio, but improves performance even with a 10% miss ratio.

Trading off throughput and hit ratio: Fig. 11a plots request throughput vs. cache size for these policies on the Memcachier trace. RankCache achieves the highest request throughput of all policies except random, and tags increase throughput at every cache size. RankCache increases throughput because *(i)* it eliminates nearly all synchronization and *(ii)* LHD+ achieves higher hit ratio than other policies, avoiding time-consuming evictions.

Fig. 11b helps explain these results by plotting request throughput vs. hit ratio for the different systems. These numbers are gathered by sweeping cache size for each policy on a uniform random trace, equalizing hit ratio across policies at each cache size. Experimental results are shown as points, and we fit a curve to each dataset by assuming that:

$$\text{Total service time} = \#\,\text{GETs} \times \text{GET time} + \#\,\text{SETs} \times \text{SET time}$$

As Fig. 11b shows, this simple model is a good fit, and thus GET and SET time are independent of cache size.



(a) vs. cache size. **(b)** vs. hit ratio.

Figure 11: Request throughput on Memcachier trace at 16 server threads. RankCache with LHD achieves the highest request throughput of all implementations, because it reduces synchronization and achieves a higher hit ratio than other policies. Tags are beneficial except at very high hit ratios.

Fig. 11b shows how important hit ratio is, as small improvements in hit ratio yield large gains in request throughput. This effect is especially apparent on CLOCK because it synchronizes on evictions, but not on hits. Unfortunately, CLOCK achieves the lowest hit ratio of all policies, and its throughput suffers as a result. In constrast, LHD+ pushes performance higher by improving hit ratio, and RankCache removes synchronization to achieve the best scaling of all implementations.

Response latency: Fig. 12 shows the average response time of GETs and SETs with different policies running at 1 and 16 server threads, obtained using the same procedure as Fig. 11b. The 16-thread results show that, in a parallel setting, RankCache achieves the lowest per-operation latency of all policies (excluding random), and in particular using separate tags greatly reduces eviction time. While list- or heap-based policies are faster in a sequential setting, RankCache's lack of synchronization dominates with concurrent requests. Because CLOCK synchronizes on evictions, its evictions are slow at 16 threads, explaining its sensitivity to hit ratio in Fig. 11b. RankCache reduces GET time by 5× vs. list and prio-queue, and SET time by 5× over CLOCK.

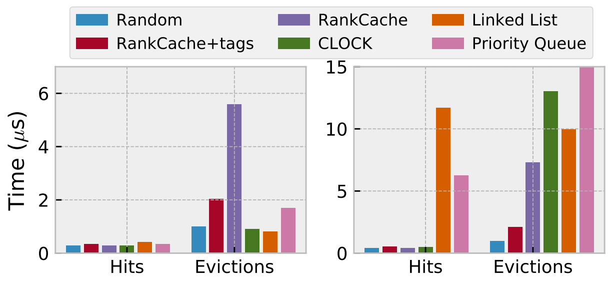


Figure 12: Request processing time for hits and evictions at a single thread (left) and 16 threads (right).

In a real-world deployment, RankCache's combination of high hit ratio and low response latency would yield greatly reduced mean and tail latencies and thus to significantly improved end-to-end response latency.

6 Related Work

Prior work in probabilistic eviction policies: EVA, a recent eviction policy for processor caches [7, 8], introduced the idea of using conditional probability to balance hits vs. resources consumed. There are several significant differences between LHD and EVA that allow LHD to perform well on key-value workloads.

First, LHD and EVA use different ranking functions. EVA ranks objects by their net contribution measured in hits, not by hit density. This matters, because EVA's ranking function does not converge in key-value cache workloads and performs markedly worse than LHD. Second, unlike processor caches, LHD has to deal with variable object sizes. Object size is one of the most important characteristics in a key-value eviction policy. RankCache must also rebalance memory across slab classes to implement a global ranking. Third, LHD classifies objects more aggressively than is possible with the implementation constraints of hardware policies, and classifies by last hit age instead of frequency, which significantly improves hit ratio.

Key-value caches: Several systems have tried to improve upon memcached's poor hit ratio under objects of varying sizes. Cliffhanger [18] uses shadow queues to incrementally assign memory to slab classes that would gain the highest hit ratio benefit. Similarly, Dynacache [17], Moirai [49], Mimir [43] and Blaze [10] determine the appropriate resource allocation for objects of different sizes by keeping track of LRU's stack distances. Twitter [41] and Facebook [37] periodically move memory from slabs with a high hit ratio to those with a low hit ratio. Other systems have taken a different approach to memory allocation than memcached. Memshare [19] and MICA [34] utilize log-structured memory allocation. In the case of all the systems mentioned above, the memory allocation is intertwined with their eviction policy (LRU).

Similar to RankCache, Hyperbolic caching [11] also uses sampling to implement dynamic ranking functions. However, as we have demonstrated, Hyperbolic suffers from higher miss ratios, since it is a recency-based policy that is susceptible to performance cliffs, and Hyperbolic did not explore concurrent implementations of sampling as we have done in RankCache.

Replacement policies: Prior work improves upon LRU by incorporating more information about objects to make better decisions. For example, many policies favor objects that have been referenced frequently in the past, since intuitively these are likely to be referenced again soon. Prominent examples include LRU-K [38], SLRU [29], 2Q [28], LRFU [31], LIRS [26], and ARC [35]. There is also extensive prior work on replacement policies for objects of varying sizes. LRU-MIN [2], HYBRID [52], GreedyDual-Size (GDS) [14], GreedyDual-Size-Frequency (GDSF) [4, 16], LNC-R-W3 [45], AdaptSize [9], and Hyperbolic [11] all take into account the size of the object.

AdaptSize [9] emphasizes object admission vs. eviction, but this distinction is only important for list-based policies, so long as objects are small relative to the cache's size. Ranking functions (e.g., GDSF and LHD) can evict low-value objects immediately, so it makes little difference if they are admitted or not (Fig. 5).

Several recent policies explicitly avoid cliffs seen in LRU and other policies [6, 11, 18]. Cliffs arise when policies' built-in assumptions are violated and the policy behaves pathologically, so that hit ratios do not improve until all objects fit in the cache. LHD also avoids cliffs, but does so by avoiding pathological behavior in the first place. Cliff-avoiding policies achieve hit ratios along the cliff's convex hull, and no better [6]; LHD matches or exceeds this performance on our traces.

Tuning eviction policies: Many prior policies require application-specific tuning. For example, SLRU divides the cache into S partitions. However, the optimal choice of S, as well as how much memory to allocate to each partition, varies widely depending on the application [24, 50]. Most other policies use weights that must be tuned to the access pattern (e.g., [2, 11, 27, 38, 45, 52]). For example, GD* adds an exponential parameter to Eq. 1 to capture burstiness [27], and LNC-R-W3 has separate weights for frequency and size [45]. In contrast to LHD, these policies are highly sensitive to their parameters. (We have implemented LNC-R-W3, but found it performs worse than LRU without extensive tuning at each size, and so do not present its results.)

7 Conclusions

This paper demonstrates that there is a large opportunity to improve cache performance through non-heuristic approach to eviction policies. Key-value caches are an essential layer for cloud applications. Scaling the capacity of LRU-based caches is an unsustainable approach to scale their performance. We have presented a practical and principled approach to tackle this problem, which allows applications to achieve their performance goals at significantly lower cost.

Acknowledgements

We thank our anonymous reviewers, and especially our shepherd, Jon Howell, for their insightful comments. We also thank Amit Levy and David Terei for supplying the Memcachier traces, and Daniel Berger for his feedback and help with implementing AdaptSize. This work was funded by a Google Faculty Research Award and supported by the Parallel Data Lab at CMU.

References

[1] Redis. http://redis.io/. 7/24/2015.

[2] M. Abrams, C. R. Standridge, G. Abdulla, S. Williams, and E. A. Fox. Caching proxies: Limitations and potentials. Technical report, Blacksburg, VA, USA, 1995.

[3] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of the Fourth International Conference on on Parallel and Distributed Information Systems*, DIS '96, pages 92–107, Washington, DC, USA, 1996. IEEE Computer Society.

[4] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin. Evaluating content management techniques for web proxy caches. *ACM SIGMETRICS Performance Evaluation Review*, 2000.

[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[6] N. Beckmann and D. Sanchez. Talus: A simple way to remove cliffs in cache performance. In *HPCA-21*, 2015.

[7] N. Beckmann and D. Sanchez. Modeling cache performance beyond LRU. *HPCA-22*, 2016.

[8] N. Beckmann and D. Sanchez. Maximizing cache performance under uncertainty. *HPCA-23*, 2017.

[9] D. S. Berger, R. K. Sitaraman, and M. 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, Boston, MA, 2017. USENIX Association.

[10] H. Bjornsson, G. Chockler, T. Saemundsson, and Y. Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 59. ACM, 2013.

[11] A. Blankstein, S. Sen, and M. J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 499–511, Santa Clara, CA, 2017. USENIX Association.

[12] A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive paging with locality of reference. *Journal of Computer and System Sciences*, 50(2):244–258, 1995.

[13] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134. IEEE, 1999.

[14] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, USITS'97, pages 18–18, Berkeley, CA, USA, 1997. USENIX Association.

[15] H. Che, Y. Tung, and Z. Wang. Hierarchical web caching systems: Modeling, design and experimental results. *IEEE Journal on Selected Areas in Communications*, 2002.

[16] L. Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories, 1998.

[17] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, July 2015. USENIX Association.

[18] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, Santa Clara, CA, Mar. 2016. USENIX Association.

[19] A. Cidon, D. Rushton, S. M. Rumble, and R. Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334, Santa Clara, CA, 2017. USENIX Association.

[20] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of WWW client-based traces. Technical report, Boston, MA, USA, 1995.

[21] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2), 2013.

[22] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 371–384, Berkeley, CA, USA, 2013. USENIX Association.

[23] B. Fitzpatrick. Distributed caching with Memcached. *Linux journal*, 2004(124):5, 2004.

[24] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 167–181, New York, NY, USA, 2013. ACM.

[25] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer. High performance cache replacement using re-reference interval prediction. In *ISCA-37*, 2010.

[26] S. Jiang and X. Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *SIGMETRICS Perform. Eval. Rev.*, 30(1):31–42, June 2002.

[27] S. Jin and A. Bestavros. GreedyDualâĹŮ web caching algorithm: exploiting the two sources of temporal locality in web request streams. *Computer Communications*, 24(2):174–183, 2001.

[28] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

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

[30] R. E. Kessler, M. D. Hill, and D. A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Transactions on Computers*, 1994.

[31] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. *SIGMETRICS Perform. Eval. Rev.*, 27(1):134–143, May 1999.

[32] C. Li and A. L. Cox. GD-Wheel: a cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems*, page 5. ACM, 2015.

[33] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 476–488, New York, NY, USA, 2015. ACM.

[34] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, Apr. 2014. USENIX Association.

[35] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[36] Memcachier. www.memcachier.com.

[37] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[38] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 297–306, New York, NY, USA, 1993. ACM.

[39] K. Psounis and B. Prabhakar. A randomized web-cache replacement scheme. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1407–1415. IEEE, 2001.

[40] M. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ISCA-34*, 2007.

[41] M. Rajashekhar and Y. Yue. Twemcache. blog.twitter.com/2012/caching-with-twemcache.

[42] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured Memory for DRAM-based Storage. In *FAST*, pages 1–16, 2014.

[43] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[44] D. Sanchez and C. Kozyrakis. The zcache: Decoupling ways and associativity. In *MICRO-43*, 2010.

[45] P. Scheuermann, J. Shim, and R. Vingralek. A case for delay-conscious caching of web documents. *Computer Networks and ISDN Systems*, 29(8):997–1005, 1997.

[46] A. Seznec. A case for two-way skewed-associative caches. In *ACM SIGARCH Computer Architecture News*, volume 21, pages 169–178. ACM, 1993.

[47] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, Feb. 1985.

[48] SNIA. MSR Cambridge Traces. http://iotta.snia.org/traces/388, 2008.

[49] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey. Software-defined caching: Managing caches in multi-tenant data centers. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 174–181. ACM, 2015.

[50] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. RIPQ: Advanced photo caching on flash for Facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386, Santa Clara, CA, Feb. 2015. USENIX Association.

[51] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park. Cache modeling and optimization using miniature simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 487–498, Santa Clara, CA, 2017. USENIX Association.

[52] R. P. Wooster and M. Abrams. Proxy caching that estimates page load delays. In *Selected Papers from the Sixth International Conference on World Wide Web*, pages 977–986, Essex, UK, 1997. Elsevier Science Publishers Ltd.

[53] N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11:525–541, 1994.

A Age coarsening with bounded error

RankCache chooses how much to coarsen ages and how many ages to track in order to stay within a user-specified error tolerance. RankCache is very conservative, so that in practice much more age coarsening and fewer ages can be used with no perceptible loss in hit rate.

Choosing a maximum age: The effect of age coarsening is to divide ages into equivalence classes in chunks of $\textsf{\textit{COARSENESS}}$, so that the maximum true age that can be tracked is $\textsf{\textit{COARSENESS}} \times \textsf{\textit{MAX\_AGE}}$. Any events above this maximum true age cannot be tracked. Hence, if the access pattern is a scan at a larger reuse distance than this, the cache will be unable to find these objects, even with an optimal ranking metric.

If the cache fits N objects and the scan contains M objects, then the maximum hit rate on the trace is N/M. To keep the error tolerance below ϵ, we must track ages up to $M \geq N/\epsilon$, hence:

$$\textsf{\textit{MAX\_AGE}} \geq \frac{N}{\textsf{\textit{COARSENESS}} \times \epsilon} \tag{7}$$

Choosing age coarsening: $\textsf{\textit{COARSENESS}}$ hurts performance by forcing RankCache to be conservative and keep objects around longer than necessary, until RankCache is certain that they can be safely evicted. The effect of large $\textsf{\textit{COARSENESS}}$ is to reduce effective cache capacity, since more space is spent on objects that will be eventually evicted. In the worst case, all evicted objects spend an additional $\textsf{\textit{COARSENESS}}$ accesses in the cache, reducing the space available for hits proportionally.

Coarsening thus "pushes RankCache down the hit rate curve". The lost hit rate is maximized when the hit rate curve has maximum slope. Since optimal eviction policies have concave hit rate curves [6], the loss from coarsening is maximized when the hit rate curve is a straight line. Once again, this is the hit rate curve of a scanning pattern with uniform object size.

Without loss of generality, assume objects have size $= 1$. The cache size equals the sum of the expected resources spent on hits and evictions [8],

$$N = \mathbb{E}[H] + \mathbb{E}[E]$$

In the worst case, coarsening increases space spent on evictions by

$$\mathbb{E}[E'] = \mathbb{E}[E] + \textsf{\textit{COARSENING}},$$

so space for hits is reduced

$$\mathbb{E}[H'] = \mathbb{E}[H] - \textsf{\textit{COARSENING}}$$

With a scan over M objects, the effect of coarsening is thus to reduce cache hit rate by

$$\text{Hit rate loss} = \frac{\textsf{\textit{COARSENING}}}{M}$$

This loss is maximized when M is small, but M cannot be too small since $M \leq N$ leads to zero misses.

To bound this error below ϵ, RankCache coarsens ages such that

$$\textsf{\textit{COARSENING}} \leq N \times \epsilon \tag{8}$$

Substituting into Eq. 7 yields

$$\textsf{\textit{MAX\_AGE}} \geq \frac{1}{\epsilon^2} \tag{9}$$

Implementation: Age coarsening thus depends only on the error tolerance and number of cached objects. RankCache monitors the number of cached objects and, every 100 intervals, updates $\textsf{\textit{COARSENING}}$ and $\textsf{\textit{MAX\_AGE}}$. We find that hit rate is insensitive to these parameters, so long as they are within the right order of magnitude.