Queueing Systems (2026) 110:28
https://doi.org/10.1007/s11134-026-09986-1

Why increasing the hit ratio can hurt cache throughput

Ziyue Qiu[1] · **Juncheng Yang**[1] · **Mor Harchol-Balter**[1]

Received: 3 May 2025 / Revised: 12 March 2026 / Accepted: 24 March 2026
© The Author(s) 2026

Abstract

Software caches are an intrinsic component of almost every computer system. Consequently, caching algorithms, particularly eviction policies, are the topic of many papers. Most of these prior papers evaluate the caching algorithm based on its *hit ratio*, namely the fraction of requests that are found in the cache, as opposed to disk. The "hit ratio" is viewed as a proxy for traditional performance metrics like system throughput or response time. Intuitively, it makes sense that a higher hit ratio should lead to higher throughput (and lower response time), since more requests are found in the cache (low access time) as opposed to the backend storage (high access time). This paper challenges this intuition. We show that increasing the hit ratio can actually *hurt* the throughput (and response time) for many caching algorithms. Specifically, higher hit ratios can lead to contention on the hit path, resulting in lower throughput. This is relevant in light of recent caching systems papers that have noticed that many LRU-based eviction policies can suffer from high contention and thus have aimed at developing solutions that reduce lock contention. Our paper provides a methodological framework to precisely determine where the contention is and how the hit ratio affects this. In particular, we are able to pinpoint the exact hit ratio beyond which throughput degrades, across a wide variety of eviction algorithms. Our investigation follows a three-pronged approach involving (i) queueing modeling and analysis, (ii) simulation to validate the accuracy of the queueing model, and (iii) implementation and measurement. We also show that the phenomenon of decreasing throughput at higher hit ratios is likely to be more pronounced in future systems, where the trend is toward faster disks and more cores per CPU.

This paper is an extended version of a paper with a similar title that appeared in the EAI ValueTools 2024 conference [1]. This paper reframes the original paper in the context of recent prior systems work. Furthermore, this paper includes an analysis of more caching algorithms and also answers questions raised during the conference. .

✉ Mor Harchol-Balter
harchol@cs.cmu.edu

Ziyue Qiu
ziyueqiu@cs.cmu.edu

[1] Computer Science Department, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213, USA

Published online: 20 April 2026

Keywords Caches · Hit ratio · Performance evaluation · Scalability · Modification analysis · Queueing theory · LRU · Eviction policies · Lock contention · Concurrency · Bottleneck analysis

Mathematics Subject Classification 68M20 · 60K30 · 60K25 · 68M01

1 Introduction

DRAM-based software caches are widely deployed in today's infrastructure. Examples range from simple and small page caches in laptops and mobile phones to large multi-layer distributed and heterogeneous key-value caches and object caches in the data centers, e.g., Meta Cachelib [2] and Google CliqueMap [3], and include many types of caches in between [4–6].

The purpose of the cache is to allow fast data access. Typically, cached items can be accessed anywhere from 100 to 10,000 times faster than those on disk [7]. The principle of caching is very simple: Store items that are *likely to be accessed soon* in the cache. Store everything else on disk. Note that, for ease of exposition, we will always refer to the backend storage as "the disk." In practice however, modern high-performant backend storage is often built on SSDs, as will be the case in the experiments in this paper.

Utilizing a cache improves *throughput*, the average number of requests served per second (RPS). This is particularly important in data processing applications where the goal is to process as many data requests as possible per unit time. Examples include the caches used in big-data systems (Hadoop, HDFS [14], Alluxio [8]), deep learning systems (Pytorch [15]), and databases (RocksDB [16]).

1.1 Cache eviction algorithms

All caches have a common component: the cache *eviction algorithm*. The eviction algorithm decides which item to evict when the cache is full. Cache eviction algorithms are often designed around the Least-Recently-Used (LRU) algorithm [17–20], which evicts the least recently *accessed* item in the cache. This is commonly achieved by ordering the items in the cache via a global linked list. Because of its popularity, this paper will focus on LRU cache eviction and its variants. LRU and variants of LRU are widely used because data access patterns often show *temporal locality*, where recently used data have a higher chance to get reused [21, 22]; LRU is optimized for workloads with high temporal locality.

Recent caching systems papers have observed that LRU can sometimes result in high lock contention (see Sect. 2.3 for references). In an effort to reduce contention, some modern production systems have opted to use variants of LRU.

One solution for reducing the contention caused by LRU is to instead use First-In-First-Out (FIFO), which evicts the least recently *inserted* item, i.e., the oldest item. While FIFO has been shown to alleviate some lock contention (see Sect. 2.3), it is not optimized to keep the most recently accessed items in the cache, as compared with

LRU. A related drawback of FIFO is that it requires a larger cache size to achieve the same hit ratio as LRU.

Many more advanced eviction algorithms also exist [23–39].

1.2 The quest for higher hit ratio

The overall *performance goal* of a cache is to improve the request throughput and reduce the request latency. Despite this, researchers have resorted to using the cache *hit ratio* (fraction of requests found in the cache) as a *proxy* for measuring performance [2, 5, 40–49]. Maximizing the hit ratio makes intuitive sense for improving system performance since we want to maximize the fraction of accesses that can be completed quickly from the cache and minimize the fraction of accesses that need to go to the slow disk.

But what if this intuition is wrong? What if increasing the hit ratio actually hurts performance?

This is the question investigated in this paper. The cache eviction algorithms we evaluate are summarized in Table 1.

1.3 A 3-pronged approach to determining if higher hit ratio helps

While many papers have analyzed the cache *hit ratio*, e.g., [50–64], the question of how the hit ratio affects throughput has been overlooked. Perhaps this is because it seems so obvious that increasing the hit ratio can only help.

We take a three-pronged approach to determine whether a higher hit ratio, in fact, improves throughput.

1.3.1 A. Queueing model for upper bounding throughput

First we develop a novel queueing model of our cache for a range of popular eviction policies, based on measurements from our implementation. We then use queueing theory to derive an *upper bound on the throughput* of this queueing model, as a function of the hit ratio, p_{hit}. For example, our bound for the case of LRU is given in Fig. 1 via the red solid line. The analytic upper bound clearly shows that throughput first increases with hit ratio, then levels off, and then decreases. The decrease in throughput starts at the critical point p^*_{hit}.

1.3.2 B. Simulation of the queueing model

Because exact analysis of the queueing model is not possible, we next simulate the queueing model to obtain its exact throughput. For LRU, the result is shown in Fig. 1 via the blue dotted line.

Table 1 Table shows the algorithms we evaluated

Algorithm	Description	Production System	Our Findings: Does increasing the hit ratio always help?
LRU	Accessed item is moved to front of queue. Evict the item at end of queue	Alluxio [8], RocksDB [9], LevelDB [9]	No
FIFO	Evicts the oldest item	ATS [10]	Yes
Probabilistic LRU	Only moves accessed item to the head of queue with some probability $1-q$	HHVM [11]	Depends on q
FIFO-Reinsertion a.k.a. CLOCK	Item at end of queue gets a second chance through the queue before eviction	RocksDB [13]	Yes
Segmented LRU	Uses two LRU queues to differentiate items that have been accessed twice	Linux [12]	No
S3-FIFO [13]	Uses a small FIFO queue to evict most new and unpopular objects	RedPanda	Yes

Detailed descriptions of the algorithms are in Sect. 4. A classification of 12 additional algorithms is given in Table 2

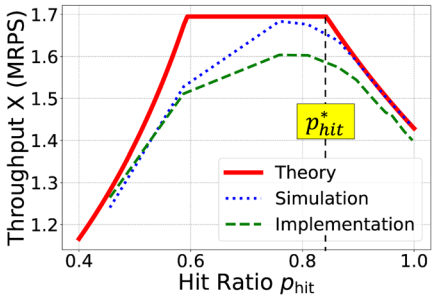


Fig. 1 Throughput under LRU (measured in millions of requests per second) increases as the hit ratio increases initially but then drops when the hit ratio gets high. Throughput starts to decline after critical point p^*_{hit}

1.3.3 C. Implementation of the caching system

Finally we implement our caching system with a range of eviction algorithms. Our prototype builds on Meta's HHVM Cache [11, 65], but we add a feature to emulate varying disk speeds and Multi-Programming Limits. Our results generalize to many other in-memory caches, e.g., CacheLib [2], Memcached [66], Intel OCF [67], BCache [68], and RocksDB LRU Cache [9]. For LRU, the result of our implementation is shown via the green dashed line in Fig. 1. Importantly, the simulation result is within 5% of the implementation result. This tells us the performance predicted by our queueing model provides a very good estimate of system performance.

1.4 Why increasing the hit ratio can hurt, in brief

The traditional intuition favoring high hit ratios goes like this: Every request goes to the cache, but only some fraction (the misses) additionally go to the disk. Hence, by reducing the miss ratio, we reduce the request latency and improve the throughput and mean response time.

Queueing theory provides a careful bottleneck analysis, explaining why the above intuition can be wrong. As disks have become faster (recall, the term "disk" refers to SSD-backed storage), and the number of CPU cores has increased, the disk has become able to support more concurrent requests more quickly. This has caused the bottleneck to shift from the disk access to various cache operations. While there are cache operations on both the "hit path" and the "miss path" (see Figs. 2, 6, and 11), the "demand" created by a cache operation depends on the product of its service time and the probability that we follow that path. Consequently, as p_{hit} increases, the dominant bottleneck can *shift* from a cache operation on the miss path to one on the hit path. Beyond this critical switching point, increasing p_{hit} only hurts performance (throughput decreases and response time increases). As we'll see, queueing theory produces an excellent prediction of the critical p^*_{hit} point, after which performance degrades.

1.5 Impact on contention avoidance

Concurrency and lock contention in caches is a well recognized, high-impact engineering problem (see Sect. 2.3). The impact of our queueing-theoretic bottleneck analysis is that it shows us exactly where the bottleneck occurs and how the bottleneck *shifts* as the hit ratio changes. The generality of our framework allows us to compare families of caching policies and obtain general insights under future scenarios including faster disk speeds and higher concurrency.

1.6 Contributions

The contributions of this paper are summarized below:

- This paper shows that increasing cache hit ratio can hurt throughput for many LRU-like cache eviction algorithms. We show this via a three-pronged approach, involving queueing theory, simulation, and implementation.
- We develop *queueing models* that allow us to understand the effect of the cache hit ratio on system throughput and mean response time. This is done for six different caching policies. The modeling is non-trivial and, to the best of our knowledge, no prior work has provided a formal model for mathematically examining the effect of contention on throughput in caching systems.
- While our queueing analysis only provides upper bounds, the analysis clearly indicates that throughput initially rises with hit ratio and then drops with hit ratio. The analysis also provides the critical switching point, p^*_{hit}.
- We also *implement* many caching policies, including LRU, FIFO, Probabilistic LRU, and CLOCK. We validate the correctness of our queueing models, by *simulating* the queueing models and showing that the simulation results match the implementation results within 5% for all caching policies studied.
- We evaluate the effect of changing disk latency as we move from older disk speeds (500 μs) to current disk speeds (100 μs) to future disk speeds (5 μs). As we move to future disk speeds, the critical hit ratio, p^*_{hit}, after which throughput starts to deteriorate, moves earlier and earlier (see Sect. 5).
- We also evaluate the effect of another trend, increasing the number of CPU cores (concurrency). We find that p^*_{hit} moves earlier for higher concurrency, i.e., higher Multi-Programming Level (MPL) (see Sect. 5).

2 Background

2.1 DRAM-based software caches

As explained in Sect. 1, DRAM-based software caches are extremely common today. In a DRAM-based software cache, DRAM is used to cache the accessed data, making cache access time quick and enabling high concurrency.

This paper focuses on DRAM-based software caches. SSD-based software caches, hardware caches, and network-connected caches are all outside the scope of this paper.

SSD-based software caches differ from DRAM-based software caches because the cache access is 100 times slower than in the DRAM system and is also less concurrent. Hardware caches behave very differently. They are typically set-associative caches (e.g., RRIP-based eviction algorithms [69, 70]), and there is no concept of a global linked list and hence no software operations. Network-connected caches also behave differently. Here the bottleneck is the time needed to move the data, rather than the time to manage software operations.

Future Trends The performance of DRAM-based software caches is highly influenced by the number of cores on which the software runs and the backend disks. Before 2000, CPUs had only one core [71], but a modern CPU has 32-192 cores [72]. The increase in CPU cores enables better performance, allowing more concurrent requests; however, it also presents challenges because the cores need to coordinate with each other.

2.2 Back-end storage (the disk)

Today's high-performant backend storage is often built on SSDs, where high-end SSDs have latencies around 5 μs [73, 74], lower-end SSDs show latencies of a few hundred μs [75, 76], and most commercial SSDs have latency in between [77–79]. In this paper, whenever we refer to the "disk," we have in mind one of these SSDs. These backend disks support massive concurrency [80, 81]. Because of the high concurrency, coupled with the fact that most requests are *not* served at the disk, there is typically very little or no queueing at the disk. In this paper, we therefore model as a zero-queueing infinite-server station.

Future Trends: In the future, we expect SSDs to only become faster. The impact of this is discussed in Sect. 5.

2.3 Throughput-oriented caching and contention avoidance

It is known that the throughput of an LRU cache does not increase linearly with concurrency due to lock contention. Because an LRU cache needs to maintain objects in the least-recently-used order, an object must be promoted (moved to the head of the global linked list) under locking; concurrent hits will contend for the lock. To address this issue in production systems, practitioners have developed several solutions. For example, Cachelib [2, 20] does not promote objects that have been promoted recently; in other words, it reverts back to FIFO for potentially popular objects; HHVM [11] won't do a promotion if there is another promotion happening concurrently; Caffeine [82], Ristretto [83], RocksDB's HyperClock [84] and others decouple the promotion from cache hits; Redis [19], RocksDB's LRU cache [85], and Linux [12] use sharding to reduce the concurrency on each cache partition.

Besides the above simple solutions adopted in industry to avoid contention, many recent works have also proposed new designs to improve cache throughput and scalability. For example, Friedman et al. show that limited associativity can improve cache throughput under high concurrency [86]; MICA [87] uses sharding to improve throughput; Hyperbolic [88] and LHD [28] use random sampling to reduce contention

at cache hits; Segcache [41] uses a segment-structured storage layout with batched evictions to improve throughput; S3-FIFO [13] and SIEVE [89] use FIFO queues to achieve higher throughput.

Although all of the above works recognize that contention can lead to lower throughput, they do not provide a formal model to mathematically understand the effect of contention and increased concurrency. In particular, these works do not tie the contention to hit ratio and do not specify the "critical hit ratio" beyond which the throughput actually decreases.

3 A three-pronged approach to LRU

Because common caching policies are largely based on LRU, we devote this section to LRU's performance. Our purpose is to explain why increasing the hit ratio can lead to decreased throughput. The numbers used in the queueing model are measured from our prototype built on Meta's DRAM-based software cache, with hardware to be described in Sect. 3.4. Takeaways are summarized in Sect. 3.5.

3.1 A closed-loop queueing model of LRU

As with many caching systems evaluations and benchmarks, e.g., Cachelib [2], YCSB benchmark [90], S3-FIFO [13], and FrozenHot [91], our system is best modeled by a *closed-loop queueing model*, where new requests are triggered by the completion of previous requests. There is a fixed Multi-Programming Limit (MPL), N, denoting the number of requests that can be in the system at a time (this is dictated by the number of cores—in our case 72).

Note that not all caching systems are closed-loop. In Sect. 3.2.2, we briefly discuss open-loop systems.

3.1.1 Modeling concurrency in a closed-loop model

Each request is handled by a single core. The total number of requests in the system is thus limited by the total number of cores. Throughout, we assume that there is one CPU with 72 cores, and thus, we can process 72 requests concurrently.

We thus model our caching system via a closed-loop queueing model, where a new request is allowed to enter only when some other request completes. The *multi-programming limit (MPL)* for the system is $N = 72$. See [92, Chapters 2,6,7] and [93] for background on modeling closed-loop queueing models.

3.1.2 Modeling disk access and cache access

Contemporary SSDs expose substantial internal parallelism. If we couple this parallism together with the fact that most requests are not served at the disk, but rather the cache, it is reasonable to model the disk as having no queue. In queueing speak, we thus represent the disk as a *think station* (infinite number of simultaneous service stations)

with mean think time $\mathbf{E}[Z_{disk}] = 100\mu s$. Likewise, the cache lookup can also be executed concurrently. Thus, the cache lookup can also be modeled as a think station but with a much faster mean think time of $\mathbf{E}[Z_{cache}] = 0.51\mu s$. Note that a think station is different from a queue station in that there is no queueing at a think station—every request starts running immediately, and requests are served concurrently.

3.1.3 Modeling software global list operations

In an *LRU* cache, all cached items are stored in a single *global linked list*, where the least-recently-used item is the "tail" item and is the one to evict, while the most-recently-used item is at the "head" of the list. A request for some item d first looks for d in the cache. Either d is in the cache (called a "hit"), or it is not (a "miss"). The probability of a hit is denoted by p_{hit} and the probability of a miss is p_{miss}, where $p_{hit} + p_{miss} = 1$.

If the request is a hit, then two things need to happen:

1. The item d must be delinked from its position in the global linked list. This is the *delink operation*. The delink time is denoted by the service time random variable S_{delink}.
2. The item d needs to be attached to the head of the global linked list. This is called the *cache head update*. The head update time is denoted by the random variable S_{head}.

The actual "reading time" (the time to read a 4 KB block from DRAM) is not included in our model. The reason is that this is very small compared with cache lookup, is handled concurrently, and is the same across all algorithms.

If the request is a miss, then three things need to happen:

1. The item d needs to be found on disk.
2. The least-recently-used item, at the tail of the global list, needs to be removed. We call this the *cache tail update* and denote it by the random variable S_{tail}.
3. Item d needs to be attached to the head of the global linked list. This is the *cache head update* mentioned earlier, denoted by random variable S_{head}.

We can thus model the LRU caching system via the queueing model shown in Fig. 2.

3.1.4 Modeling device service times

Understanding how to properly model S_{head}, S_{tail}, and S_{delink} is important and non-trivial and will come up again when discussing different eviction policies. We will explain these here. We start with S_{head}, which denotes the time needed for a head update on the global linked list. This has *two components*. The first component is obvious: just the update to the global list. The second component is less obvious: communicating this update to all the other requests in the queue. Specifically, the core performing the head update needs to communicate with all the other cores in the head update queue to alert them that this head update is happening. Obviously, if there are many requests in the head update queue, then this communication will take longer.

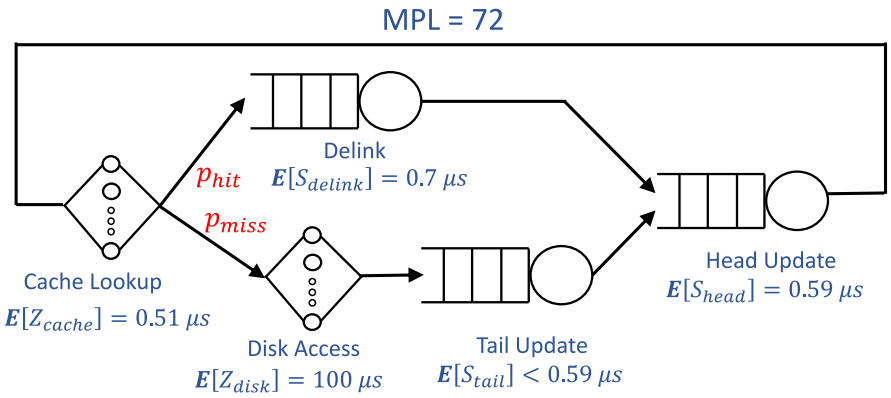


Fig. 2 Queueing model of LRU cache

In summary, S_{head} consists of a *constant* head update time and a communication time dependent on *queue length*. When the head update is the bottleneck operation, our measurements show that these two components add up to $\mathbf{E}[S_{head}] = 0.59\mu s$. When the head update is not the bottleneck operation, the queue length at the head update queue drops, thus $\mathbf{E}[S_{head}]$ decreases. However, as we'll see in the analysis of LRU (Sect. 3.2), in the case where an operation is not a bottleneck, its service time does not impact overall throughput much. Specifically, any value in $0 < \mathbf{E}[S_{head}] < 0.59\mu s$ will produce the same throughput if a head update is not the bottleneck operation. We find that S_{head} follows approximately a Bounded Pareto ($\alpha = 0.45$) distribution ranging from $0.1\mu s$ to $1.2\mu s$. However, as we'll see in Sect. 3.2, the throughput analysis of the queueing model is only influenced by the mean, $\mathbf{E}[S_{head}]$. Similarly, when the delink queue is the bottleneck, $\mathbf{E}[S_{delink}] = 0.7\mu s$. When the delink queue is not the bottleneck, its value has little impact on analysis.

Importantly, when we look at other eviction algorithms, the number for $\mathbf{E}[S_{head}]$ can change because the queue length at the head update queue will change. To measure $\mathbf{E}[S_{head}]$, we create a setting where the head update is the bottleneck operation. This means that the head update queue will be flooded. Consequently, the service time, S_{head}, is simply the inter-departure time from this flooded queue (namely the time between consecutive departures from the queue), which is easy to measure. Likewise, it is easy to measure $\mathbf{E}[S_{delink}]$, because that device can also be made into the bottleneck (by setting p_{hit} appropriately).

The case of S_{tail} is slightly different because the tail update is *never* the bottleneck operation. This makes it difficult to precisely measure $\mathbf{E}[S_{tail}]$ because we can't keep the tail update server fully utilized as we were able to do for the other servers. Fortunately, again, as shown in Sect. 3.2, the precise value will not matter.

3.2 Analysis of LRU queueing model

The goal of this section is to determine analytically how the system throughput is affected by hit ratio, p_{hit}. Key to our analysis is identifying the "bottleneck device"

(see [92, Chapters 6,7]). Looking at Fig. 2, one might assume naively that the bottleneck is the *disk access*, since that time dwarfs all other times in the network by a factor of more than 100. However, this intuition is wrong because the bottleneck device is by definition the device behind which jobs queue, and there is no queueing at the disk in our model (recall that we explained our modeling of the disk and the cache in Sect. 3.1.2). The analysis below identifies the bottleneck device and provides an upper bound on system throughput. We first illustrate our analysis assuming that mean disk latency is $\mathbf{E}[Z_{disk}] = 100\mu s$ and then generalize to other disk latencies.

The first step is to determine the *mean think time* of the system, $\mathbf{E}[Z]$, where $\mathbf{E}[Z]$ is the mean time spent on accesses that can be executed concurrently by all cores. This includes cache lookup and disk access:

$$\begin{aligned}\mathbf{E}[Z] &= \mathbf{E}[Z_{cache}] + (1 - p_{hit}) \cdot \mathbf{E}[Z_{disk}] \\ &= 0.51 + (1 - p_{hit}) \cdot 100 = 100.51 - 100 p_{hit}\end{aligned}$$

For each queue, we now compute the *device demand*, which is the expected service demand on the corresponding device, per request into the system. The device demands are:

$$\begin{aligned}D_{delink} &= p_{hit} \cdot 0.7 \\ D_{tail} &< (1 - p_{hit}) \cdot 0.59 \\ D_{head} &= 0.59\end{aligned}$$

The *total demand*, D, is the sum of the device demands:

$$D = D_{delink} + D_{tail} + D_{head}$$

Because $0 < D_{tail} < 0.59$, we have upper and lower bounds on D as follows:

$$\begin{aligned}0.7 p_{hit} + 0.59 < D &< 0.7 p_{hit} + 0.59(1 - p_{hit}) + 0.59 \\ 0.7 p_{hit} + 0.59 < D &< 0.11 p_{hit} + 1.18\end{aligned}$$

The next step is to determine the *bottleneck device*, which is the device with the highest demand. We can see that the bottleneck device is the delink device if p_{hit} is sufficiently high, specifically $p_{hit} > 0.84$. Otherwise, the bottleneck device is the head update device. We write this as:

$$D_{max} = \max(0.59, 0.7 p_{hit}) = \begin{cases} 0.59 & \text{if } p_{hit} < 0.84 \\ 0.7 p_{hit} & \text{if } p_{hit} > 0.84 \end{cases}$$

We use X to denote system throughput. From [92, Theorem 7.1], we know that X is upper-bounded by two terms, as follows:

$$X \leq \min\left(\frac{N}{D + \mathbf{E}[Z]} \quad \frac{1}{D_{max}}\right).$$

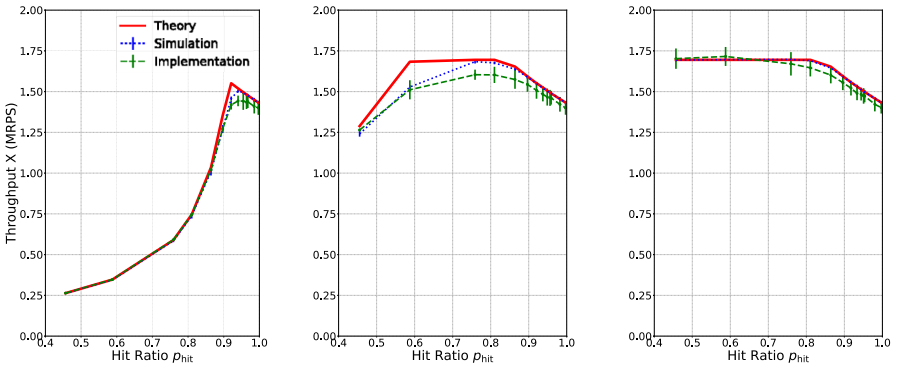


Fig. 3 Results for theory, implementation, and simulation under an LRU cache. The throughput of the LRU cache decreases at higher hit ratios. This trend becomes more pronounced as we move toward lower disk latencies—from **(a)** to **(b)** to **(c)**

Substituting in the expressions for $\mathbf{E}[Z]$, and D_{max} that we have already derived, as well as the lower bound on D and the fact that $N = MPL = 72$, we have that, for the case of $\mathbf{E}[Z_{disk}] = 100\mu s$:

$$X_{\text{LRU}} \leq \min\left(\frac{72}{101.1 - 99.3p_{hit}} \quad \frac{1}{\max(0.59, 0.7p_{hit})}\right) \tag{1}$$

Equation (1) represents an *upper bound on throughput*, shown in red in Fig. 3b. This turns out to be a very good bound on the measured throughput from our implementation. Equation (1) highlights three regimes:

- When $p_{hit} < 0.59$, the first term in (1) is the relevant bound (minimum term). In this regime, throughput (X_{LRU}) is strictly increasing with p_{hit}.
- When $0.59 < p_{hit} < 0.84$, the second term in (1) is the relevant bound, where the max term in the denominator is 0.59. In this regime, throughput is constant: $X_{\text{LRU}} = 0.59$.
- When $p_{hit} > 0.84$, the second term in (1) is again the relevant bound, but the max term in the denominator is $0.7p_{hit}$. In this regime, X_{LRU} decreases with p_{hit}.

As a side note: Recall that $0 < \mathbf{E}[S_{tail}] < 0.59$. In the above analysis, we assumed that $\mathbf{E}[S_{tail}] = 0$ because we wanted an upper bound on X_{LRU}. If instead, we had used any value of $\mathbf{E}[S_{tail}]$ in the range between 0 and 0.59, the impact on our result for X would be very small ($< 0.5\%$). The reason is that changing $\mathbf{E}[S_{tail}]$ would only change D, not D_{max}. Hence only the first term in (1) would change, and we only care about this first term when p_{hit} is low, specifically $p_{hit} < 0.59$. Within this first term, $D + \mathbf{E}[Z]$ would change from $101.1 - 99.3p_{hit}$ to $101.69 - 99.89p_{hit}$.

The above analysis assumed that $\mathbf{E}[Z_{disk}] = 100\mu s$. If we redo the analysis for the case where $\mathbf{E}[Z_{disk}] = 5\mu s$, the throughput is (2), as shown in red in Fig. 3c.

$$X_{\text{LRU}} \leq \min\left(\frac{72}{6.1 - 4.3p_{hit}} \quad \frac{1}{\max(0.59, 0.7p_{hit})}\right) \tag{2}$$

If we repeat the analysis for the case where $\mathbf{E}[Z_{disk}] = 500\mu s$, we get (3) shown in red in Fig. 3a.

$$X_{\text{LRU}} \leq \min\left(\frac{72}{501.1 - 499.3p_{hit}} \quad \frac{1}{\max(0.59, 0.7p_{hit})}\right) \tag{3}$$

3.2.1 Discussion of the analytic results

Contention Summary We've seen that increasing the hit ratio leads to lower throughput when the hit ratio is high. We have shown via a queueing analysis why this happens. When the hit ratio is high, we see that the *delink* operation on the *hit path* becomes the bottleneck. Hence, almost all requests are queued behind the delink server in Fig. 2. Increasing p_{hit} only increases the demand on the delink server, hence increasing contention on the hit path. This makes the queue on the delink server even longer. This leads to higher request latency and lower throughput.

More intuition Intuitively, however, it might still be hard to understand how even having a full queue at the delink device can be problematic, given that 72 requests times $0.7\mu s$ each is only about $50\mu s$, which is still small compared to a single visit to the disk, which takes $100\mu s$. To understand what's going on, consider this *thought experiment*: Imagine that the delink queue is full. Let r denote an arbitrary request that enters the system. Request r requires a total time of $72 \cdot 0.7\mu s \approx 50\mu s$ at the full delink queue. At first, this seems preferable to sending request r to the disk, where it will take $100\mu s$. Suppose, however, we do send request r to the disk. The penalty to request r is then

$$\text{Penalty to single request } = 100\mu s - 50\mu s = 50\mu s.$$

However, consider the benefit to all the other requests. While request r is spending $100\mu s$ at the disk, the other 71 requests get 2 trips each through the delink queue. Each of those requests now waits $0.7\mu s$ less than before. So we have 71 requests each saving $2 \cdot 0.7\mu s$ (because there are 2 trips). Hence, the total savings is

$$\text{Total savings to other requests } = 71 \cdot 2 \cdot 0.7 \approx 100\mu s.$$

Hence we traded off a penalty (to a single request) of $50\mu s$ with a savings (across all the other requests) of $100\mu s$. Thus there is a *net gain* overall from having sent the one request to disk. We can now continue this reasoning in considering the benefit of sending an additional request to disk, etc.

The critical hit ratio In all the curves of Fig. 3, we find that there is some point, p^*_{hit}, after which increasing the hit ratio only hurts. This point p^*_{hit} *decreases* as the mean disk latency decreases. Thus, our message about not blindly increasing the hit ratio will become more and more valid as we move to faster disks.

Effect on mean response time Throughout, we have looked at throughput, but we could instead have looked at *mean response time*, namely the time from when a request is submitted until it completes. Given that we have a closed-loop setting,

mean response time and throughput are inversely related (see [92, Chapters 6,7]). Hence mean response time *increases* for higher hit ratios.

3.2.2 Brief discussion of closed versus open queueing models

Throughout this paper, we model our caching system as a closed queueing network. This makes sense for several reasons: First, the number of cores at the CPU (72) limits the number of requests that can be simultaneously processed (the MPL in a closed-loop system). Second, our goal in this paper is to study *throughput*, where throughput is the average rate at which requests complete. Throughput is limited by the bottleneck device in a closed-loop system. For these same reasons, many systems are best modeled by closed queueing networks, see Table 1 of [93].

Another choice is to model a caching system as an open system. Here there is a stream of exogenous arriving requests with some average arrival rate λ. The system is still limited to only serving 72 requests at a time, so any additional requests that arrive will have to queue outside the caching system. In practice, it is desirable to keep the queue short and not drop requests, which typically translates to keeping λ low. The throughput metric makes much less sense in an open system, since the average completion rate (throughput) is just the arrival rate (λ), since every arrival completes. An open queueing model hence makes much less sense for the goals of this paper.

3.3 Simulation evaluation

Recall that our queueing analysis from Sect. 3.2 only provides upper bounds on throughput. To get the exact throughput of the queueing network, we turn to simulation. We use an event-driven simulation[1] based on our measurements of the device latencies (service times). We note that our results appear to be insensitive to the particular service time distributions used. This is consistent with the findings in [93] for closed-loop models.

The results of our simulation are shown in Fig. 3 via dotted blue lines. We also show 95% confidence intervals, but they are typically too small to be visible. As expected, the results of simulation lie below the red theory lines, which represent the theoretical upper bound.

3.4 Implementation setup and results

In both the analysis (Sect. 3.2) and the simulation (Sect. 3.3), we were evaluating the queueing network in Fig. 2. We now study LRU via our implementation that is independent of any queueing network. Our implementation is a prototype based on Meta's HHVM Cache. Figure 3 shows the results of our LRU implementation via green dashed lines with 95% confidence intervals.

Experimental setup Our experiments use dual-socket servers with Intel Xeon Platinum 8360Y 36-core processors (Ubuntu 20) at 2.4GHz, running on CloudLab

[1] Open-sourced: https://github.com/ziyueqiu/CacheThputSim.git

platform [94]. To avoid NUMA impacts, our evaluations only use a single socket with hyperthreading enabled. To provide consistent results, we disable turbo-boosting and fix the per-core frequency at 3.1 GHz.

Our Intel Xeon Platinum CPU allows for 72 cores to run concurrently. This limits the number of requests that can be in the system at once to 72. Each request accesses a 4KB block of data, the common size in database block caches. We emulate three different disk speeds: $500\mu s$, $100\mu s$, and $5\mu s$. We consider p_{hit} in the range of [0.4, 1], with a step size of 0.05 in most cases, but a step size of 0.02 for higher p_{hit} values. Each experiment is run 20 times.

Workload creation For request generation, we employ 72 client threads, where each thread is assigned to a single CPU core. We use a synthetic popularity distribution following the Zipfian parameter $\theta = 0.99$, representative of cache accesses from e-Commerce websites [95] and social networks [40]. Recognizing that our goal is to assess the impact of hit ratio on throughput and that the popularity distribution only affects the hit ratio, we determine that it is sufficient to test with this well-established Zipfian distribution without the need for a broader range of models or real-world access traces. Throughput measurements are conducted after some warmup period, when the cache is full.

Results of implementation Our implementation results and simulation results are always within 5%.

3.5 Summary and takeaways

We started the section by presenting a queueing model of our LRU caching system (Sect. 3.1). We were able to derive an *upper bound* on throughput in our model as a function of the hit ratio (Sect. 3.2). Our analysis elucidated that when the hit ratio gets high, the queueing bottleneck shifts from the head update operation to the delink operation. Increasing the hit ratio beyond p^*_{hit} puts extra demand (contention) on the delink operation on the hit path, resulting in longer delays and lower throughout. We next simulated the queueing network (Sect. 3.3), determining the exact throughput as a function of hit ratio. Finally, in Sect. 3.4, we implemented our LRU caching system, yielding results within 5% of the simulation.

There are two takeaways. First, because the implementation matches the simulation, we conclude that our queueing model is an excellent representation of the real system, at least with respect to understanding system throughput as a function of hit ratio. Second, we see that a simple queueing analysis enables us to easily predict p^*_{hit}. This foreshadows a *theme of this paper*—queueing analysis alone suffices to predict the effect of hit ratio on throughput for many policies.

4 Evaluation of other policies

In this section, we will apply our three-pronged approach to the remaining algorithms in Table 1.

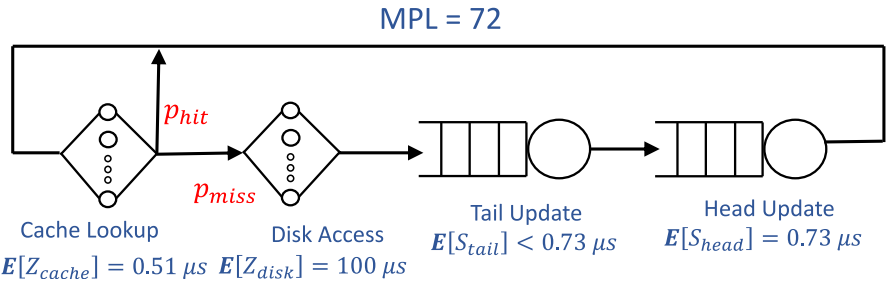


Fig. 4 Queueing model of FIFO cache

4.1 FIFO

In the *FIFO policy*, we store all cached objects into a single global linked list. The list maintains the same ordering the entire time, with new items being added to the list head and the oldest item dropping off the tail.

If a request for item d is a cache hit, *nothing* happens to the global linked list. When the request is a miss, the list must be updated. Specifically:

1. The item, d, needs to be read from disk.
2. The oldest item needs to be removed from the global list, requiring a cache tail update.
3. Item d must be attached to the list head, requiring a cache head update.

4.1.1 A queueing model of FIFO

We can thus model the FIFO caching system via the closed-loop queueing model in Fig. 4. It may seem strange that $\mathbf{E}[S_{head}]$ is higher in the FIFO system (Fig. 4) than in the LRU system (Fig. 2). To understand why this happens, recall that there are two components to $\mathbf{E}[S_{head}]$: *(i)* a constant time needed for a head update to the global linked list and *(ii)* a communication time proportional to the queue length at the head update queue. Now, in LRU there are 72 jobs in the system, split among *three* queues, but in FIFO, the 72 jobs are split among only *two* queues. Consequently the expected queue length of each of FIFO's two queues is larger than each of LRU's three queues. This in turn means that component (ii) is higher under FIFO, explaining why $\mathbf{E}[S_{head}]$ is larger under FIFO.

4.1.2 Analysis of FIFO queueing model

We now follow the same approach that we used for LRU to analyze our closed queueing network.

Again the *mean think time* of the system is:

$$\mathbf{E}[Z] = \mathbf{E}[Z_{cache}] + p_{miss} \cdot \mathbf{E}[Z_{disk}] = 100.51 - 100 p_{hit}$$

For each queue, we now compute the *device demand*:

$$D_{tail} < (1 - p_{hit}) \cdot 0.73 \qquad D_{head} = (1 - p_{hit}) \cdot 0.73$$

The *total demand*, D, is the sum of the device demands:

$$D = D_{tail} + D_{head}$$

Because $0 < D_{tail} < (1 - p_{hit}) \cdot 0.73$, we have upper and lower bounds on D as follows:

$$(1 - p_{hit}) \cdot 0.73 < D < (1 - p_{hit}) \cdot 0.73 + (1 - p_{hit}) \cdot 0.73$$
$$0.73 - 0.73 p_{hit} < D < 1.46 - 1.46 p_{hit}$$

The *bottleneck device* is always the head update, so:

$$D_{max} = 0.73 - 0.73 p_{hit}.$$

Given all the above terms, where $\mathbf{E}[Z_{disk}] = 100\mu s$, from [92, Theorem 7.1] our upper bound on throughput X is:

$$X_{\text{FIFO}} \leq \min\left(\frac{72}{101.24 - 100.73 p_{hit}} \quad \frac{1}{0.73 - 0.73 p_{hit}}\right) \tag{4}$$

Equation (4) represents an *upper bound on throughput*, shown in red Fig. 5b. Recall that $0 < \mathbf{E}[S_{tail}] < 0.73$. In the above analysis, we assumed that $\mathbf{E}[S_{tail}] = 0$ because we wanted an upper bound on X. If instead, we had used any value of $\mathbf{E}[S_{tail}]$ in the range $(0, 0.73)$, X would only change by $< 0.5\%$.

What's interesting about (4) is that both terms in the bound of X increase with p_{hit}. This contrasts with the expressions of throughput for LRU, where increasing p_{hit} decreased the second term in the bound of X.

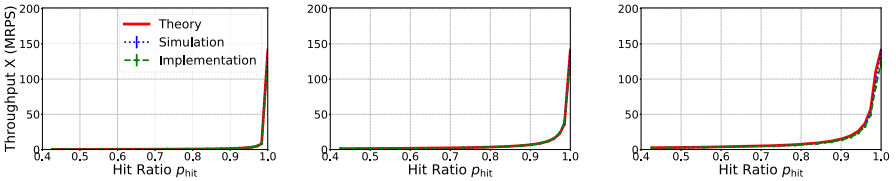
The above analysis assumed that $\mathbf{E}[Z_{disk}] = 100\mu s$. We can likewise redo the analysis for the case where $\mathbf{E}[Z_{disk}] = 5\mu s$, obtaining:

$$X_{\text{FIFO}} \leq \min\left(\frac{72}{6.24 - 5.73 p_{hit}} \quad \frac{1}{0.73 - 0.73 p_{hit}}\right) \tag{5}$$

Likewise, when $\mathbf{E}[Z_{disk}] = 500\mu s$, we obtain:

$$X_{\text{FIFO}} \leq \min\left(\frac{72}{501.24 - 500.73 p_{hit}} \quad \frac{1}{0.73 - 0.73 p_{hit}}\right) \tag{6}$$

The bounds in (4), (5), and (6) are shown via the red solid lines in Fig. 5.



(a) Disk latency 500 µs (b) Disk latency 100 µs (c) Disk latency 5 µs

Fig. 5 Results for theory, implementation, and simulation under a FIFO cache. For all three curves, the throughput of the FIFO cache always increases at higher hit ratios under different disk latencies

4.1.3 Results of the three-pronged approach

Contention Summary Our analysis for FIFO shows that increasing the hit ratio *always* leads to higher throughput, regardless of the mean disk latency. The queueing theory shows this mathematically. The bottleneck device is now the *head update* in the *miss path*, not in the hit path. Therefore, increasing the hit ratio does not result in increased demand on the bottleneck device, and hence does not lead to deleterious effects on throughput.

Our simulation of the queueing network follows the same process as Sect. 3.3, and our implementation follows the process of Sect. 3.4. Results of simulation are shown in blue dotted lines, and results of implementation are shown via green dashed lines in Fig. 5. These agree within 5%.

4.2 Probabilistic LRU

Under Probabilistic LRU there is an additional parameter q that controls how close the algorithm is to LRU (lower q) versus FIFO (higher q). As always, there is a global linked list; however, now the ordering of items in the linked list is a mixture of FIFO and LRU.

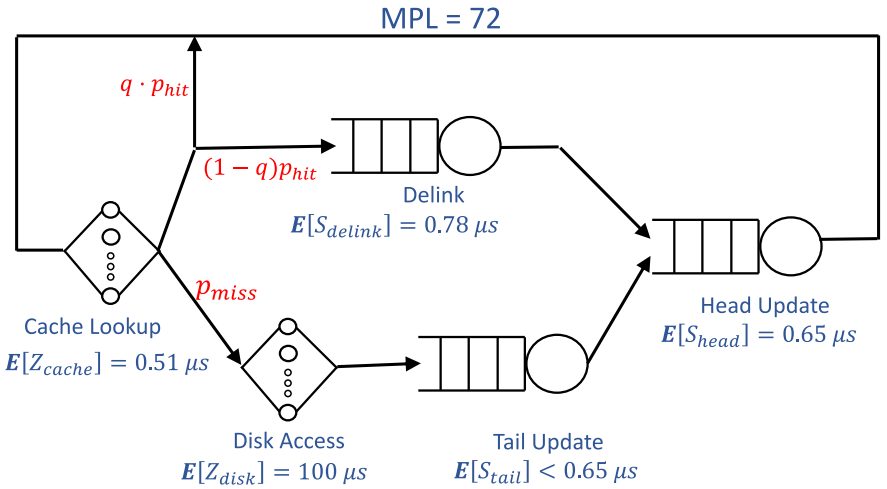
If a request for item d is a hit then, with probability q, *nothing* happens to the global linked list (as in FIFO), and, with probability $1 - q$, we follow LRU.

If the request is a miss, then three things need to happen:

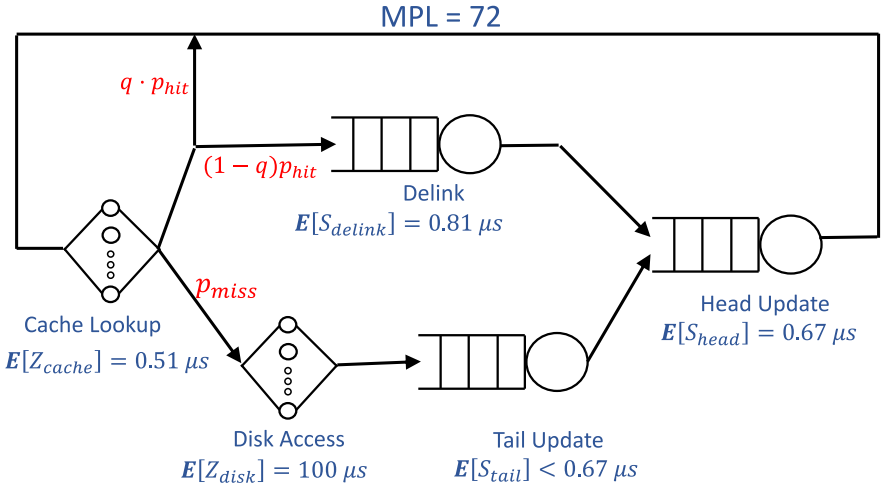
1. The item d needs to be read from disk.
2. The item at the tail of the global linked list must be removed (a tail update).
3. Item d must be attached to the global list head (a head update operation).

4.2.1 A queueing model of Probabilistic LRU

The service times for the usual operations turn out to be slightly affected by the value of q. This is because q affects the queue lengths, hence affecting the communication overhead, as explained in Sect. 3.1.3. The queueing network for Probabilistic LRU is shown in the case of $q = 0.5$ (Fig. 6a) and $q = 1 - \frac{1}{72}$ (Fig. 6b). We chose these values because it turns out that q has to be extremely high, $\geq 1 - \frac{1}{N}$, to show FIFO-like behavior. For all other values of q, our analysis (and implementation), shows LRU-like behavior. This is an interesting finding in its own right.



(a) Queueing model of Probabilistic LRU cache with q = 0 .5.



(b) Queueing model of Probabilistic LRU cache with $q = 1 - \frac{1}{72}$. Notice that the numbers change slightly.

Fig. 6 Queueing model of Probabilistic LRU cache

4.2.2 Analysis of Probabilistic LRU

We apply the usual analysis approach to Fig. 6. For both networks, the *mean think time* of the system is:

$$\mathbf{E}[Z] = \mathbf{E}[Z_{cache}] + p_{miss} \cdot \mathbf{E}[Z_{disk}] = 100.51 - 100 p_{hit}$$

For the network in Fig. 6a, we end up with the following results:

$$\begin{aligned} D_{Delink} &= 0.5 p_{hit} \cdot 0.78 \\ D_{tail} &< (1 - p_{hit}) \cdot 0.65 \\ D_{head} &= (0.5 p_{hit} + 1 - p_{hit}) \cdot 0.65 = (1 - 0.5 p_{hit}) \cdot 0.65 \\ D_{max} &= \max\left(0.39 p_{hit}, 0.65 - 0.325 p_{hit}\right) \\ D &= D_{Delink} + D_{tail} + D_{head} \end{aligned}$$

Because $0 < D_{tail} < (1 - p_{hit}) \cdot 0.65$, we have upper and lower bounds on D as follows:

$$\begin{aligned} 0.65 + 0.065 p_{hit} < D &< 0.65 + 0.065 p_{hit} + (1 - p_{hit}) \cdot 0.65 \\ 0.65 + 0.065 p_{hit} < D &< 1.3 - 0.585 p_{hit} \end{aligned}$$

Substituting the above expressions into [92, Theorem 7.1], we find that when $q = 0.5$:

$$X^{\mathbf{E}[Z_{disk}]=100\mu s}_{\text{Prob-LRU}(q=0.5)} \leq \min\left(\frac{72}{101.16 - 99.94 p_{hit}} \; \frac{1}{\max(0.39 p_{hit}, 0.65 - 0.325 p_{hit})}\right)$$

$$X^{\mathbf{E}[Z_{disk}]=5\mu s}_{\text{Prob-LRU}(q=0.5)} \leq \min\left(\frac{72}{6.16 - 4.94 p_{hit}} \; \frac{1}{\max(0.39 p_{hit}, 0.65 - 0.325 p_{hit})}\right)$$

$$X^{\mathbf{E}[Z_{disk}]=500\mu s}_{\text{Prob-LRU}(q=0.5)} \leq \min\left(\frac{72}{501.16 - 499.94 p_{hit}} \; \frac{1}{\max(0.39 p_{hit}, 0.65 - 0.325 p_{hit})}\right)$$

Likewise, we get the following upper bounds when $q = 1 - \frac{1}{72} = 0.986$:

$$X^{\mathbf{E}[Z_{disk}]=100\mu s}_{\text{Prob-LRU}(q=0.986)} \leq \min\left(\frac{72}{101.18 - 100.65 p_{hit}} \; \frac{1}{\max(0.011 p_{hit}, 0.67 - 0.656 p_{hit})}\right)$$

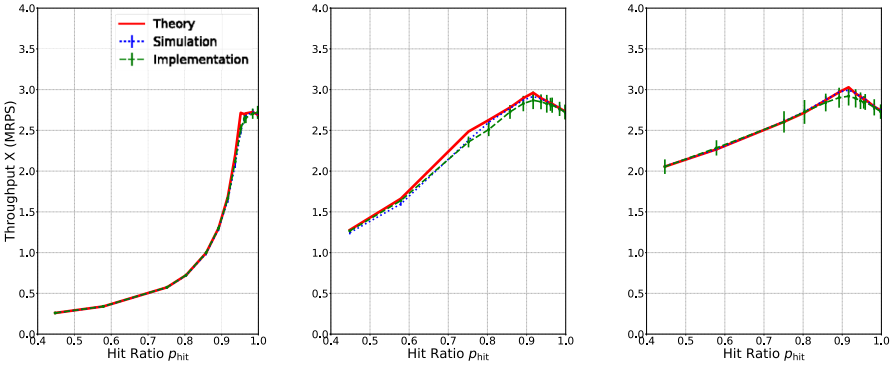
$$X^{\mathbf{E}[Z_{disk}]=5\mu s}_{\text{Prob-LRU}(q=0.986)} \leq \min\left(\frac{72}{6.18 - 5.65 p_{hit}} \; \frac{1}{\max(0.011 p_{hit}, 0.67 - 0.656 p_{hit})}\right)$$

$$X^{\mathbf{E}[Z_{disk}]=500\mu s}_{\text{Prob-LRU}(q=0.986)} \leq \min\left(\frac{72}{501.18 - 500.65 p_{hit}} \; \frac{1}{\max(0.011 p_{hit}, 0.67 - 0.656 p_{hit})}\right)$$

Note that we used the lower bound on D to upper-bound the throughput. However, as in our prior analyses, using the upper bound on D will only change the results by $< 0.5\%$.

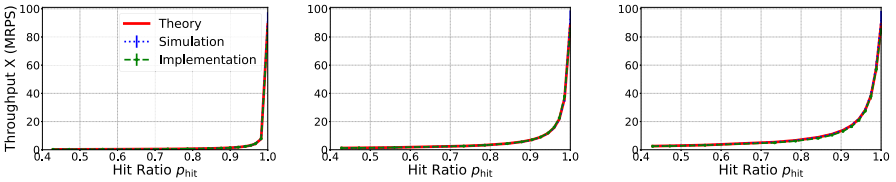
4.2.3 Results of the three-pronged approach

Figures 7 and 8 show the results of analysis (red solid lines), simulation (blue dotted lines), and implementation (green dashed lines). Again simulation and implementation agree within 5% and the analytic upper bound provides an excellent indication of the trends.



(a) Disk latency 500 µs (b) Disk latency 100 µs (c) Disk latency 5 µs

Fig. 7 Results for theory, implementation, and simulation under a Probabilistic LRU cache with $q = 0.5$. Throughput decreases at higher hit ratios under all disk latencies



(a) Disk latency 500 µs (b) Disk latency 100 µs (c) Disk latency 5 µs

Fig. 8 Results for theory, implementation, and simulation under Probabilistic LRU with $q = 1 - \frac{1}{72} = 0.986$. For all three curves, throughput increases with hit ratio

Contention Summary: The behavior of Probabilistic LRU is highly dependent on the q parameter. When *q is not very high*, we see that the throughput starts decreasing beyond hit ratio p^*_{hit}. The queueing theory shows this fact mathematically. More intuitively, given that q is not high, many requests need to go through the *delink* queue on the *hit path*, which becomes the bottleneck, resulting in higher queueing times (contention) and reduced throughput.

By contrast, when *q is very high*, we basically skip the delink operation. Hence our network behaves very similarly to FIFO. Now the bottleneck device is the *head update* on the *miss path*, and hence is not affected by increasing the hit ratio.

4.3 CLOCK

The CLOCK cache eviction policy operates a global linked list, ordered like FIFO, and however, every object that is accessed gets a “second chance" to live before eviction. As in FIFO, each object d is appended to the head of the linked list and moves down toward the tail of the list as new objects are appended to the head. Each object is given a special bit set to 0. If object d gets to the tail of the list, and its bit is still 0, then it is removed from the tail. However, if d is accessed before it gets to the tail, then d’s bit is set to 1. Note that d’s position in the list does not change. When d gets to the tail of the list, because its bit is 1, it is skipped over for eviction, provided that there are other candidates with a 0 bit. The CLOCK queueing network is shown in Fig. 9.

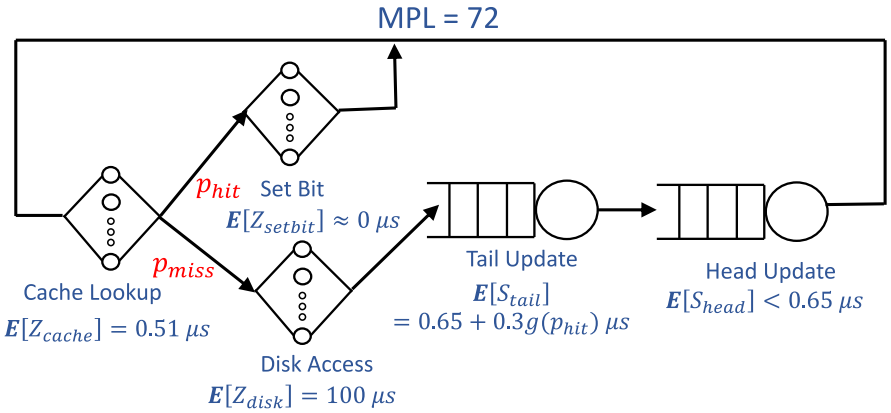


Fig. 9 Queueing model of CLOCK cache

4.3.1 A queueing model of CLOCK

Our specific implementation works as follows: Let n be the length of the linked list. When we're looking for an item to evict from the tail of the list, we start with the nth item (the last one) and check its bit. If it is 0, we evict it. If not, then we move to the $(n-1)$th item. If that is 0, we evict it. If not, then we move to the $(n-2)$th item. If that is 0, we evict it. If not, then we remove the $(n-3)$th item, regardless of its bit. Thus, the time, S_{tail}, needed for a tail update depends on how long it takes to find a 0 bit. If p_{hit} is high, the time is longer, since more items have a 1 bit. We have carefully modeled $\mathbf{E}[S_{tail}]$, and we find that $\mathbf{E}[S_{tail}]$ has two components, representing (i) the time to read the tail bit of the nth item and do a regular tail update and (ii) the additional time needed to potentially read the $(n-1)$th bit and $(n-2)$th bit and so on. Our measurements show that

$$\mathbf{E}[S_{tail}] = 0.65 + 0.3 \cdot g(p_{hit}),$$

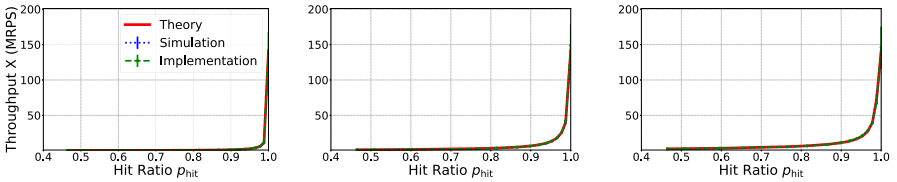
where

$$g(x) = 2.43 \cdot 10^{-5} e^{11.24x} + 0.187.$$

The queueing model for a CLOCK cache is shown in Fig. 9. As expected, this basically looks like a FIFO queueing network. The biggest difference is that the tail update queue is now the bottleneck. The other difference is that a hit in the cache requires setting the bit of the hit item to 1 instead of 0. It turns out that setting this bit takes $< 0.01\mu$s time, so we consider that to be 0.

4.3.2 Analysis of the queueing network model

Following the usual analysis approach, the *mean think time* of the system is:



(a) Disk latency 500 µs (b) Disk latency 100 µs (c) Disk latency 5 µs

Fig. 10 Results for theory, implementation and simulation under a CLOCK cache. For all three curves, the throughput of the CLOCK cache always increases with hit ratio under different disk latencies

$$\mathbf{E}[Z] = \mathbf{E}[Z_{cache}] + p_{miss} \cdot \mathbf{E}[Z_{disk}] = 100.51 - 100 p_{hit}$$

For each queue, we now compute the *device demand*:

$$\begin{aligned} D_{tail} &= (1 - p_{hit}) \cdot (0.65 + 0.3 \cdot g(p_{hit})) \\ D_{head} &< (1 - p_{hit}) \cdot 0.65 \\ D &= D_{tail} + D_{head} \\ D &> (1 - p_{hit}) \cdot (0.65 + 0.3 \cdot g(p_{hit})) \\ D &< (1 - p_{hit}) \cdot (0.65 + 0.3 \cdot g(p_{hit})) + (1 - p_{hit}) \cdot 0.65 \end{aligned}$$

The bottleneck device is the tail update, so:

$$D_{max} = (1 - p_{hit}) \cdot (0.65 + 0.3 \cdot g(p_{hit})).$$

Substituting the above expressions into [92, Theorem 7.1], we obtain an upper bound on throughput in the case where $\mathbf{E}[Z_{disk}] = 100 \mu s$:

$$\begin{aligned} X_{\text{CLOCK}} &\leq \min(A, B) \\ A &= \frac{72}{101.16 + 0.3 \cdot g(p_{hit}) - (100.65 + 0.3 \cdot g(p_{hit})) p_{hit}} \\ B &= \frac{1}{(1 - p_{hit})(0.65 + 0.3 \cdot g(p_{hit}))} \end{aligned}$$

We have derived similar bounds for the case where the disk latency is 500 μs and 5 μs. All our bounds are shown in Fig. 10.

4.3.3 Results of the three-pronged approach

Figure 10 shows the result of the theory, simulation, and implementation of CLOCK.

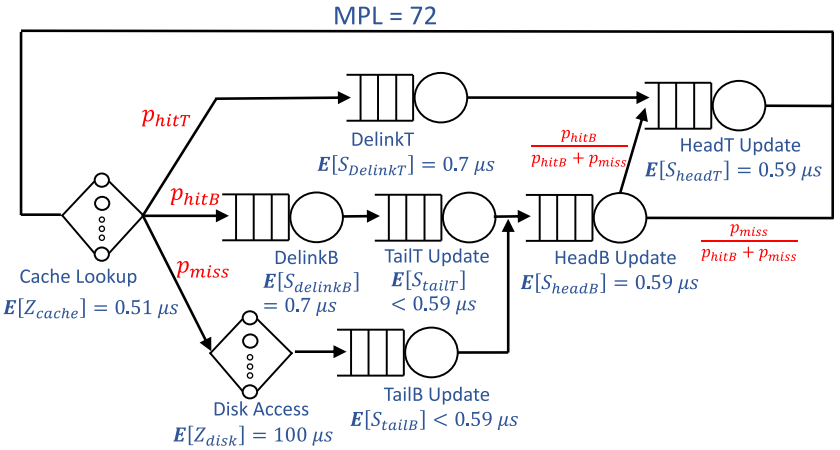


Fig. 11 Queueing model of Segmented LRU cache

Contention Summary: We see that increasing the hit ratio, p_{hit}, *always* leads to higher throughput, regardless of the mean disk latency. The queueing theory shows this fact mathematically. Intuitively, the bottleneck device is the *tail update* in the *miss path*, not in the hit path. Therefore, increasing the hit ratio this does not result in increased demand (contention) on the bottleneck device and hence does not impinge on throughput.

4.4 Segmented LRU

The Segmented LRU (SLRU) policy is one of the more advanced policies that researchers and practitioners use [96]. The high-level idea in SLRU is that the global linked list of all objects in the cache is divided into two lists: the Probationary list (denoted B for bottom) and the Protected list (T for top).

Objects initially enter the B list. If an object is never accessed after being put on the B list, it will eventually leave the cache. However, if an object on the B List is accessed, then the object will be delinked from the B list and moved onto the T list. The T list is an LRU list in the sense that it is sorted from most recently accessed to least recently accessed. When an object on the T list is accessed, the object moves to the head of the T list. When an object leaves the T list, it is moved back to the B list and the process repeats.

4.4.1 A queueing model of SLRU

The queueing network for SLRU is shown in Fig. 11. When an object d is first accessed, we do a cache lookup. There are now three cases of what might happen. If d is currently in the T list (which runs LRU), then d needs to be delinked from its current location and moved to the head of the B list (thus we have a delinkT operation followed a headT update). If d is currently in the B list, then d is delinked from the B list and moved to the head of the T list. When this happens, we need to remove the object that is at the

tail of the T list. That object is moved to the head of the B list. Finally, if d is not in the cache (a "miss"), then we need to find d in the disk. After finding d, we put it on the head of the B list and remove the object at the tail of the B list.

4.4.2 Analysis of the queueing model

To do a queueing analysis of the network in Fig. 11, we need two more things: First, we need the service times of potential bottleneck operations. These are the same as the numbers in the LRU network. Second, and more challenging, we need to understand the fraction of time that an object is found on the T list as opposed to the B list. This is obviously a function of p_{hit}.

We used our simulation to estimate this function and matched it to the following:

$$\begin{aligned}\mathbf{P}\{\text{Object in T list}\} &= \ell(p_{hit}) \equiv -0.1144 \cdot p_{hit}^2 + 1.009 \cdot p_{hit}\\ \mathbf{P}\{\text{Object in B list}\} &= f(p_{hit}) \equiv p_{hit} - \ell(p_{hit})\end{aligned}$$

We now analyze the closed queueing network in Fig. 11.
The *mean think time* of the system is:

$$\mathbf{E}[Z] = \mathbf{E}[Z_{cache}] + p_{miss} \cdot \mathbf{E}[Z_{disk}] = 100.51 - 100 p_{hit}$$

For each queue, we now compute the *device demand*:

$$\begin{aligned} D_{DelinkT} &= \ell(p_{hit}) \cdot 0.7 & D_{tailT} &< f(p_{hit}) \cdot 0.59\\ D_{DelinkB} &= f(p_{hit}) \cdot 0.7 & D_{tailB} &< (1 - p_{hit}) \cdot 0.59\\ D_{headT} &= p_{hit} \cdot 0.59 & D_{headB} &= (1 - \ell(p_{hit})) \cdot 0.59\end{aligned}$$

where

$$D_{max} = \max\left(0.7 \cdot \ell(p_{hit}), 0.59 p_{hit}, 0.59(1 - \ell(p_{hit}))\right).$$

The total demand, D, is the sum of all device demands:

$$D = D_{DelinkT} + D_{DelinkB} + D_{tailT} + D_{tailB} + D_{headT} + D_{headB}$$

$$\begin{aligned} D &> 1.29 p_{hit} + 0.59 - 0.59 \cdot \ell(p_{hit})\\ D &< 1.29 p_{hit} + 1.18 - 1.18 \cdot \ell(p_{hit})\end{aligned}$$

Substituting the above expressions into [92, Theorem 7.1], we obtain an upper bound on throughput in the case where $\mathbf{E}[Z_{disk}] = 100\mu s$:

$$\begin{aligned} X_{\text{SLRU}} &\le \min(A, B)\\ A &= \frac{72}{101.1 - 88.71 p_{hit} - 0.59\ell(p_{hit})}\\ B &= \frac{1}{\max\left(0.7\ell(p_{hit}), 0.59(1 - \ell(p_{hit}))\right)}\end{aligned}$$

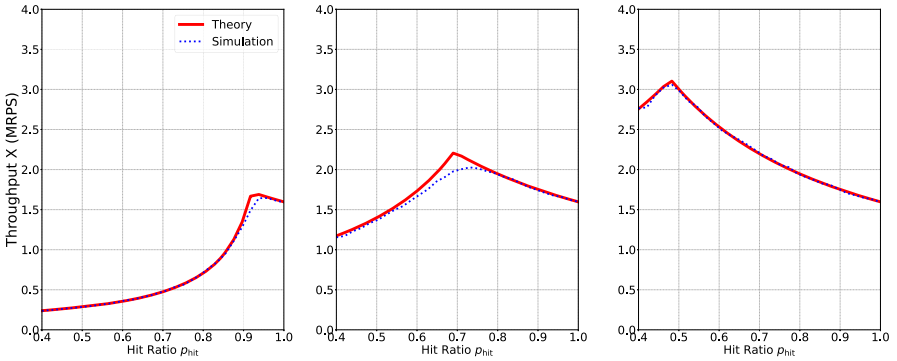


Fig. 12 Results for theory and simulation of a Segmented LRU cache. The throughput of the SLRU cache decreases at higher hit ratios. This trend becomes more pronounced as we move toward lower disk latencies—from (a) to (b) to (c)

We have derived similar expressions for the case of $\mathbf{E}[Z_{disk}] = 500\mu s$ and $\mathbf{E}[Z_{disk}] = 5\mu s$, which are shown in Fig. 12.

4.4.3 Results of analysis and simulation

Figure 12 shows the results of analysis (red solid lines) and simulation (blue dotted lines) for SLRU. We have not implemented SLRU.

Contention Summary Like LRU, SLRU also experiences reduced throughput with high hit ratios. The queueing theory shows this mathematically. More intuitively, the bottleneck here is often the *delinkT* operation on the *hit path*. Therefore, increasing hit ratio makes more requests queue behind the delinkT server which is already bottlenecked, adding contention, and hence lowering the throughput. Note that this same behavior would happen if the DelinkB operation were the bottleneck. Thus using a more complex policy, like SLRU, which has more queues, does not alleviate the bottleneck on the hit path.

4.5 S3-FIFO

S3-FIFO [13] is a new algorithm that claims to have state-of-the-art hit ratio. We have not implemented this policy, but will study it via analysis and simulation. S3-FIFO divides the global list of all elements in the cache into two global FIFO-ordered lists. The first list is called the Small List, S, because it has a fixed short length, while the second list is called the Main List, M. Typically, the S-List contains 10% of the items while the M-List contains the rest (we used these numbers in our model).

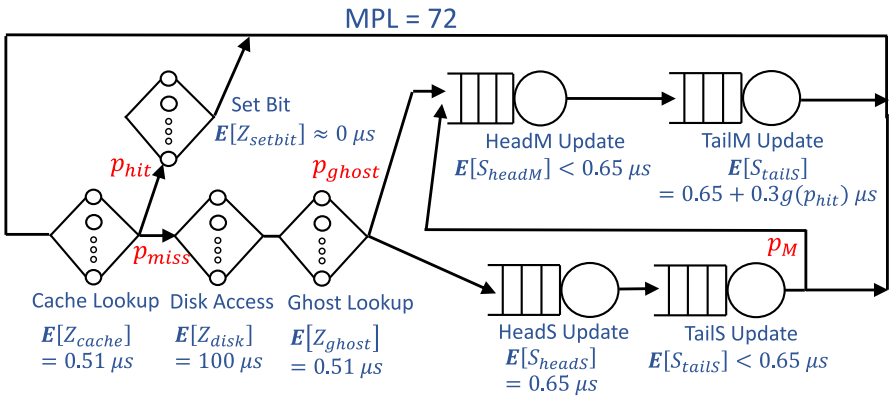


Fig. 13 Queueing model of S3-FIFO cache

4.5.1 A queueing model of S3-FIFO

S3-FIFO is very similar to CLOCK in that only a miss causes work to be done, while a hit only involves setting a bit. Under S3-FIFO, there is also a Ghost element which determines whether the requested item should be stored in the S-List or the M-List. The Ghost's decision is generally based on whether the object was accessed within the last x misses of the cache, where x is the number of items in the M-List. Ghost lookup is similar in speed to a cache lookup, so $\mathbf{E}\left[Z_{ghost}\right] = 0.51\mu s$.

As in CLOCK, every object in the cache has a special bit. When the object first enters the cache, the object is assigned a bit of 0. If the object is accessed while it is in the cache, then its bit is changed from 0 to 1.

When an object d is requested, if d is in the cache (regardless of which list), we get d and set d's bit to 1 (if it's not already 1), completing the request. If, on the other hand, d is not in the cache (a "miss"), then we need to get d from disk. We now use the Ghost to figure out in which list to store d. If d was "missed recently" (meaning it was missed sometime within the last x misses of the cache), then we store d in the M-List. Otherwise, we store d in the S-List. Since both are FIFO lists, it is advantageous to d to be stored in the M-List, which is a lot longer. Either way, we still need a Head Update (to append d) and also a Tail Update (to remove the object at the tail), thus keeping both lists at their original size. Before the object at the tail of the S-List is thrown out, it has the opportunity to move to the M-List. This happens if and only if its bit is 1. Otherwise it is tossed. Objects in the M-List never move to the S-List.

4.5.2 Analyzing the queueing model

To do a queueing analysis of the network in Fig. 13, we need three more things: First, we need to know the service times for all potential bottleneck operations. These are the same as the numbers in the CLOCK network. Second, and more challenging, we need to understand the fraction of requests that the Ghost sends to the M list (denoted p_{ghost}) as opposed to the S list. Third, we need to know the fraction of items, p_M, at

the tail of the S list that have a 1 bit associated with them as opposed to a 0 bit. We used our simulation to estimate both p_{ghost} and p_M as functions of p_{hit}.

Our results are expressible in terms of the χ^2 function, which is denoted by $h(x; a, b, c)$, where $\Gamma(\cdot)$ is the Gamma factorial function. We were able to match our measurements of p_{ghost} and p_M to the expressions below, which involve both p_{hit} and the χ^2 function with particular parameters:

$$h(x; a, b, c) = \frac{1}{2^{\frac{a}{2}}\Gamma\left(\frac{a}{2}\right)c^a}\left(\frac{x-b}{c}\right)^{\frac{a}{2}-1} e^{-\frac{x-b}{2c}}$$

$$p_{ghost} = \frac{h(65 \cdot (1 - p_{hit}); 4.4912, 1.1394, 3.595)}{1 - p_{hit}}$$

$$p_M = \frac{h(400 \cdot (1 - p_{hit}); 2.2870, 4.5309, 26.5874)}{1 - p_{hit}}$$

The *mean think time* of the system is:

$$\begin{aligned}\mathbf{E}[Z] &= \mathbf{E}[Z_{cache}] + p_{miss} \cdot (\mathbf{E}[Z_{disk}] + \mathbf{E}\left[Z_{ghost}\right]) \\ &= 0.51 + p_{miss} \cdot 100.51 = 101.02 - 100.51 p_{hit}\end{aligned}$$

For each queue, we now compute the *device demand*:

Let $q_{ghost} = 1 - p_{ghost}$. Then,

$$\begin{aligned}D_{tailS} < D_{headS} &= p_{miss} \cdot (1 - p_{ghost}) \cdot 0.65 \\ D_{headM} &< (p_{miss} \cdot q_{ghost} \cdot p_M + p_{miss} \cdot p_{ghost}) \cdot 0.65 \\ D_{tailM} &= (p_{miss} \cdot q_{ghost} \cdot p_M + p_{miss} \cdot p_{ghost}) \cdot g(p_{hit})\end{aligned}$$

Because $D_{tailM} \geq D_{headM}$, the bottleneck device can be the tail update on the Main List or the head update on the Small List, based on p_M, so:

$$D_{max} = \max\left(D_{tailM}, D_{headS}\right).$$

The *total demand*, D, is the sum of the device demands:

$$D = D_{headS} + D_{tailS} + D_{headM} + D_{tailM}$$

Because D now has four different components, it becomes messy to write out the explicit upper and lower bounds on D. Nonetheless, we have computed upper bounds on throughput, X, via the usual formula from [92, Theorem 7.1], and our results are shown in Fig. 14.

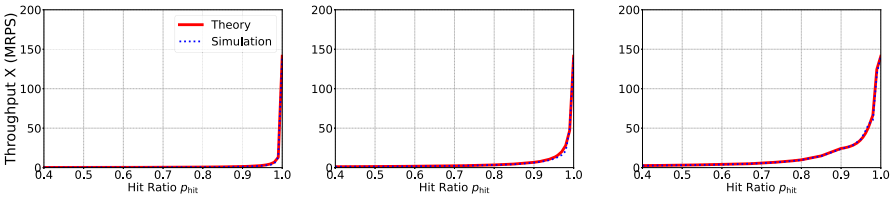


Fig. 14 Results for theory and simulation under an S3-FIFO cache. Throughput of the S3-FIFO cache always increases with the hit ratio, for all disk latencies

4.5.3 Results of analysis and simulation

Figure 14 shows the results of analysis (red solid) and simulation (blue dotted) for S3-FIFO.

Contention Summary: We see that increasing the hit ratio, p_{hit}, always leads to higher throughput, regardless of the mean disk latency. This holds because the bottleneck device (no matter whether it is the *headS update* or *tailM update*) is on the *miss path*, not in the hit path. Hence, increasing p_{hit} does not increase demand on the bottleneck device.

5 Future trends

There are two trends which will become more prominent in the future:

1. *Back-end storage (the disk) will become faster.* Specifically, backend storage is being built on SSDs, which are getting faster, with latencies in the highest-end SSDs approaching 5μs [73, 74].
2. *Caches will have increasing concurrency.* Specifically, the number of cores per CPU is increasing, which in turn means that the Multi-Programming Level (MPL) is increasing.

Figure 15 shows the effect of both these trends. Horizontally, we see disk speeds which improve from left (500μs) to right (5μs). Vertically, we see concurrency which improves from top (MPL = 16) to bottom (MPL = 144). We see that the point at which throughput starts to deteriorate, p^*_{hit}, moves earlier as we move either right (faster disks) or down (higher concurrency). Observe that in the top left corner, $p^*_{hit} \approx 1$, meaning that we never see throughput deteriorate. By contrast, in the bottom right corner, $p^*_{hit} < 0.5$, meaning that throughput deteriorates even at low hit ratios.

To understand why both future trends have the effect of lowering p^*_{hit}, it helps to return to our original throughput bound,

$$X \leq \min\left(\frac{N}{D + \mathbf{E}[Z]} \quad \frac{1}{D_{max}}\right).$$

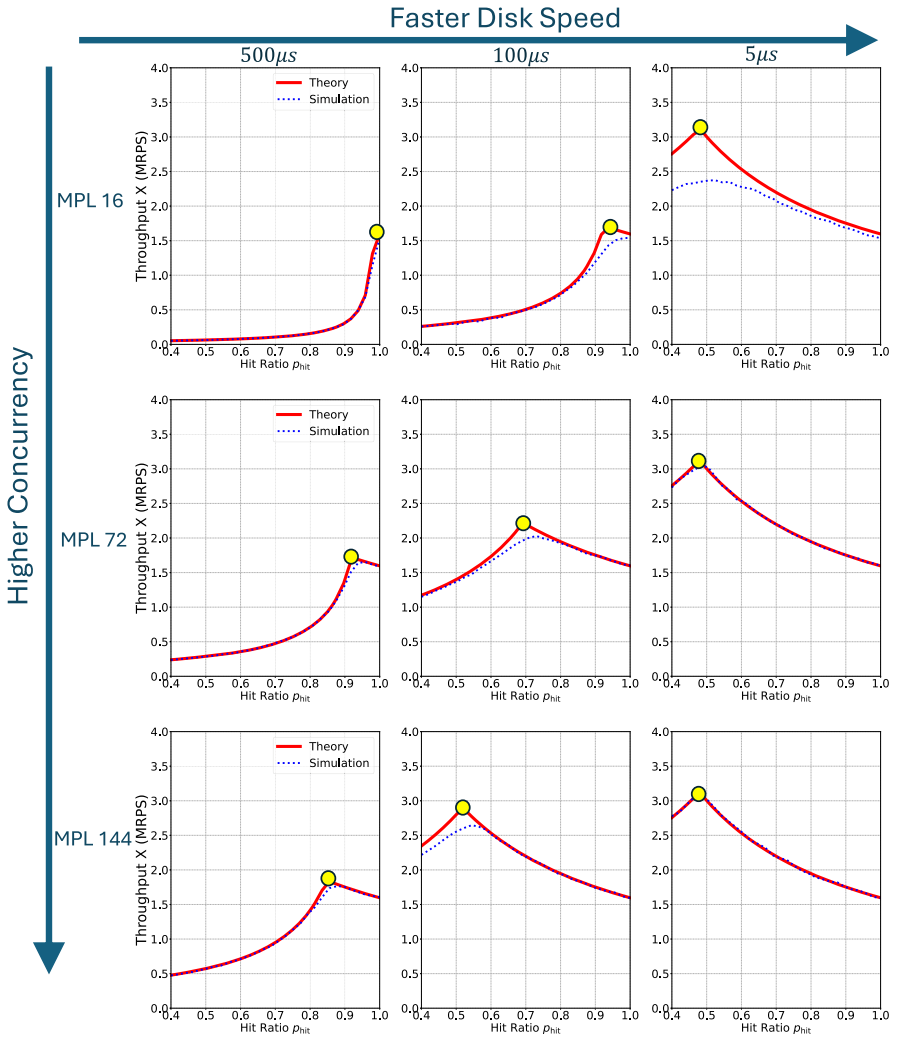


Fig. 15 The effect of two future trends: (i) Faster disk speeds and (ii) higher concurrency. Results are for the case of Segmented LRU. The yellow circles highlight the location of p^*_{hit}. As shown p^*_{hit} decreases both as disks get faster and as the concurrency level increases

Observe that increasing the MPL makes the left term higher. Likewise, increasing the disk speed results in lower total think time, $\mathbf{E}[Z]$, which also makes the left term higher. The consequence of the left term getting higher is that the *right term*, $\frac{1}{D_{max}}$ becomes the dominant term sooner, meaning that our bottleneck device becomes relevant earlier.

Table 2 Conjectured classification of additional algorithms

LRU-like	ARC [23], LIRS [24], TinyLFU [25]
	LeCaR [29], CACHEUS [30], LFU
FIFO-like	CLOCK variants [97], SIEVE [89], QDLP [32],
	Hyperbolic [88], Random, LHD [28], LRB [27]

6 Discussion of caching algorithms

In Sect. 5, we saw that in future systems increasing the hit ratio is likely to cause throughput to deteriorate even sooner. In this section, we first discuss which caching algorithms are likely to be affected by this trend and then discuss what we can do about it.

6.1 A classification of eviction algorithms

Throughout our paper, we find that common eviction algorithms can be classified into two categories: **LRU-like** algorithms and **FIFO-like** algorithms. The throughput of LRU-like algorithms (LRU, Probabilistic LRU with lower q, and SLRU) typically drops when the hit ratio becomes high. By contrast, the throughput of FIFO-like algorithms (FIFO, Probabilistic LRU with very high q, CLOCK, and S3-FIFO) always increases with the hit ratio.

In Table 2, we conjecture the behavior of other algorithms based on what we've learned. The LRU-like algorithms all have the common feature that they perform a delink operation upon a cache hit. This delink operation becomes the bottleneck, meaning that increasing the hit ratio will increase the queueing time at the delink queue, which will lead to lower throughput. The FIFO-like algorithms have the common feature that they do not update the global data structure upon cache hits. Consequently, increasing p_{hit} will not increase the queue length at the bottleneck device and therefore will not lead to lower throughput.

These FIFO-like algorithms can be further subdivided into two types. Algorithms in the first type use FIFO as the basic building block, e.g., CLOCK variants [97], S3-FIFO [13], QDLP [32], and SIEVE [89]. Algorithms in the second type do not have a global data structure, but rather uses random sampling to choose eviction candidates. Examples include LHD [28], LRB [27], and Hyperbolic [88]. As these algorithms do not maintain a global data structure, they are never bottlenecked by cache hits, so throughput only increases with p_{hit}.

6.2 Improving future caching systems

As we've explained, in the last few years, it has become very clear that LRU caching systems can suffer from high contention and low throughput. In an effort to reduce contention, some modern production systems have opted to use variants of LRU (see Sect. 2.3).

What has not been studied is how throughput decreases at higher hit ratios and what the "critical hit ratio" is across different caching policies. The decrease in throughput under high hit ratios is actually a relatively new phenomenon. Until recently, caching systems lived in the upper left hand corner of Fig. 15, so we didn't have to worry about throughput deteriorating. But as we move to a higher number of cores per CPU and lower disk times, it is evident the LRU-like caching algorithms will start to experience more problems as the hit ratio increases. Given that LRU is the basis of many caching algorithms used today, we need to figure out how we can adapt LRU-like algorithms so that their throughput *doesn't deteriorate with increasing hit ratio*.

The easiest mitigation is to reduce contention on the hit path by reducing the number of delink operations. This can be achieved by performing the delink operation probabilistically (Probabilistic LRU). As we have shown in Sect. 4, when q (the probability that skip the update on the hit path) is very high, Probabilistic LRU behaves like FIFO, where increasing the hit ratio only helps. This is the high-level idea behind a recent paper, [91].

Another approach is to send some of the requests to the disk directly, bypassing the cache, at times when cache load is high. We simulated this solution and found that it is promising: throughput stays constant after the critical p^*_{hit} point, rather than dropping.

One might be thinking at this point that maybe one should just forgo LRU altogether, in favor of FIFO. However there are reasons why cache designers prefer LRU to FIFO. Specifically FIFO is less efficient in its use of cache space—a larger cache is needed to obtain the same hit ratio under FIFO as under LRU. Thus, what we believe will be used in the future is some *combination* of FIFO and LRU which looks a lot like FIFO, but still maintains the efficiency of LRU. There are a couple recent systems papers which go in this direction [13, 89].

7 Prior work

DRAM-based caches are increasingly deployed in today's system stack and are widely studied in research [17, 29–31, 36, 43, 88, 98–107].

Most of the above works use the *hit ratio* as a proxy for system performance. Those that do look at throughput only consider it a single thread (i.e., MPL=1), e.g., DistCache [108], p-redis [109], LRB [27], GL-Cache [105]. By contrast, our paper models modern systems with MPL = 72 concurrent threads.

Two papers deserve a bit more attention because, like our paper, both are aimed at increasing throughput and both assume a high number of threads. In the first paper, FrozenHot [91], the authors fix the hit ratio at 99% and do not study throughput as a function of changing hit ratio. The second paper, NHC [110], is quite different from

ours. This paper uses NVM and Optane caches, which have much lower concurrency. Their bottlenecks are thus far different from ours; in particular the cache operations are not modeled whatsoever. Finally, neither the FrozenHot nor NHC papers have any analytic modeling component.

Many previous works and industry systems have recognized that LRU cannot achieve high throughput at high concurrency and have designed heuristics to address this problem. For example, Cachelib [20] delays the promotion upon cache hits; Caffeine [82] and Ristretto [83] use batched promotion and avoid lock contention during cache hits. Similarly, many recent academic works have also examined this problem, using sharding [87], FIFO or log-structured design [13, 41, 89], and random sampling [28, 111] to reduce lock contention at cache hits or avoid promotion altogether. Although these works have recognized that LRU has low throughput under high concurrency, our paper is the first to provide a formal model for mathematically examining the effect of contention on throughput.

8 Conclusion

This paper examines the effect of the cache hit ratio on the request throughput of DRAM-based software caching systems. We use a *three-pronged approach*. First we create a queueing model of the particular eviction policy, which we evaluate analytically to obtain an upper bound on the throughput as a function of the hit ratio. We next use simulation to exactly evaluate the queueing network. Finally we study the eviction policy in implementation.

For all policies studied, our simulation and implementation agree within 5%, indicating that our queueing network models are quite accurate. Another finding is that our analysis, while only providing an upper bound, is an excellent indicator of p^*_{hit}, the critical hit ratio at which throughput starts decreasing. The takeaway is that, for the evaluation of future eviction policies, it suffices to use analytic upper bounds to understand whether increasing hit ratio will help.

Our queueing analysis elucidates why throughput drops with higher hit ratio for LRU-like algorithms. In a nutshell, LRU-like algorithms require updating the global linked list upon a cache hit. When the hit ratio becomes very high, this update operation becomes the system bottleneck, meaning that many requests queue up there. As a consequence, increasing the hit ratio only adds contention, increasing the queueing time at this bottleneck, which reduces system throughput. Based on this intuition, we conjecture that many algorithms that we have not yet studied (ARC, LIRS, LFU, CACHEUS, LeCAR) should also exhibit this perverse behavior.

We also study two major trends in computing and storage: newer CPUs have an increasing number of cores, while disk latencies are steadily decreasing. We show that both these two trends imply that LRU-like algorithms will only behave worse in the future; i.e., throughput will start to decrease at lower and lower values of the hit ratio. We offer some suggestions to remedy this problem.

We mention again that our results apply to DRAM-based software caches (the "cache") and SSD-backed storage (the "disk").

Acknowledgements This work was supported by NSF-CIF-2403194, NSF-III-2322973, and NSF-CMMI-2307008.

Funding Open Access funding provided by Carnegie Mellon University This work was supported by NSF-CIF-2403194, NSF-III-2322973, and NSF-CMMI-2307008

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Qiu, Z., Yang, J., Harchol-Balter, M.: Can increasing the hit ratio hurt cache throughput? In: 17th EAI International Conference on Performance Evaluation Methodologies and Tools (ValueTools 2024) (2024). https://doi.org/10.48550/arXiv.2404.16219
2. Berg, B., Berger, D.S., McAllister, S., Grosof, I., Gunasekar, S., Lu, J., Uhlar, M., Carrig, J., Beckmann, N., Harchol-Balter, M., Ganger, G.R.: The CacheLib caching engine: Design and experiences at scale. In: 14th USENIX Symposium on Operating Systems Design and Implementation. OSDI'20, pp. 753–768 (2020). https://www.usenix.org/conference/osdi20/presentation/berg
3. Singhvi, A., Akella, A., Anderson, M., Cauble, R., Deshmukh, H., Gibson, D., Martin, M.M.K., Strominger, A., Wenisch, T.F., Vahdat, A.: CliqueMap: productionizing an RMA-based distributed caching system. In: Proceedings of the 2021 ACM SIGCOMM 2021 Conference. SIGCOMM'21, pp. 93–105, Virtual Event USA (2021). https://doi.org/10.1145/3452296.3472934
4. Borst, S., Gupta, V., Walid, A.: Distributed caching algorithms for content distribution networks. In: 2010 Proceedings IEEE INFOCOM, pp. 1–9. IEEE (2010)
5. Willick, D.L., Eager, D.L., Bunt, R.B.: Disk cache replacement policies for network fileservers. In: [1993] Proceedings. The 13th International Conference on Distributed Computing Systems, pp. 2–11. IEEE (1993)
6. Almeida, J.M., Eager, D.L., Vernon, M.K.: Hybrid caching strategy for streaming media files. In: Multimedia Computing and Networking 2001, vol. 4312, pp. 200–212. SPIE (2000)
7. Scott, C.: Latency Numbers Every Programmer Should Know. https://colin-scott.github.io/personal_website/research/interactive_latency.html. Accessed 10 Jan 2024
8. Alluxio: Alluxio—Data orchestration for the cloud. https://www.alluxio.io/. Accessed 10 Jan 2024
9. LevelDB. https://github.com/google/leveldb. Accessed 22 Apr 2024
10. Apache: Apache Traffic Server. https://trafficserver.apache.org/. Accessed 10 Jan 2024
11. HHVM project. https://github.com/facebook/hhvm.git. Accessed 20 Jan 2024
12. sobyte: Linux Kernel Page Replacement algorithms. https://www.sobyte.net/post/2022-01/linux-multi-lru/. Accessed 10 Jan 2024
13. Yang, J., Zhang, Y., Qiu, Z., Yue, Y., Vinayak, R.: FIFO queues are all you need for cache eviction. In: Proceedings of the 29th Symposium on Operating Systems Principles. SOSP'23, pp. 130–149, New York, NY, USA (2023). https://doi.org/10.1145/3600006.3613147
14. Hadoop: HDFS architecture guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. Accessed 10 Jan 2024
15. Pytorch: Pytorch. https://pytorch.org/. Accessed 10 Jan 2024
16. Dong, S., Kryczka, A., Jin, Y., Stumm, M.: RocksDB: evolution of development priorities in a key-value store serving large-scale applications. ACM Trans. Storage **17**(4), 26–12632 (2021). https://doi.org/10.1145/3483840
17. O'Neil, E.J., O'Neil, P.E., Weikum, G.: The LRU-K page replacement algorithm for database disk buffering. ACM SIGMOD Rec. **22**(2), 297–306 (1993). https://doi.org/10.1145/170036.170081

18. Eytan, O., Harnik, D., Ofer, E., Friedman, R., Kat, R.: It's time to revisit LRU vs. FIFO. In: 12th USENIX Workshop on Hot Topics in Storage and File Systems. hotStorage'20 (2020). https://www.usenix.org/conference/hotstorage20/presentation/eytan
19. Redis eviction policies. https://redis.io/blog/cache-eviction-strategies/. Accessed 22 Apr 2024
20. CacheLib eviction policies. https://cachelib.org/docs/Cache_Library_User_Guides/eviction_policy. Accessed 22 Apr 2024
21. Denning, P.J.: The working set model for program behavior. Commun. ACM **11**(5), 323–333 (1968)
22. Denning, P.J.: Working Sets Past and Present. IEEE Trans. Softw. Eng. SE-6(1), 64–84 (1980) https://doi.org/10.1109/TSE.1980.230464
23. Megiddo, N., Modha, D.S.: ARC: A self-tuning, low overhead replacement cache. In: 2nd USENIX Conference on File and Storage Technologies. FAST'03 (2003)
24. Jiang, S., Zhang, X.: LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In: ACM SIGMETRICS Performance Evaluation Review. SIGMETRICS'02, vol. 30, pp. 31–42 (2002). https://doi.org/10.1145/511399.511340
25. Einziger, G., Friedman, R., Manes, B.: TinyLFU: a highly efficient cache admission policy. ACM Trans. Storage **13**(4), 1–31 (2017). https://doi.org/10.1145/3149371
26. Johnson, T., Shasha, D.: 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In: Proceedings of the 20th International Conference on Very Large Data Bases. VLDB'94, pp. 439–450, San Francisco, CA, USA (1994)
27. Song, Z., Berger, D.S., Li, K., Shaikh, A., Lloyd, W., Ghorbani, S., Kim, C., Akella, A., Krishnamurthy, A., Witchel, E., et al.: Learning relaxed belady for content distribution network caching. In: 17th USENIX Symposium on Networked Systems Design and Implementation. NSDI'20, pp. 529–544 (2020)
28. Beckmann, N., Chen, H., Cidon, A.: LHD: Improving cache hit rate by maximizing hit density. In: 15th USENIX Symposium on Networked Systems Design and Implementation. NSDI'18, pp. 389–403 (2018)
29. Vietri, G., Rodriguez, L.V., Martinez, W.A., Lyons, S., Liu, J., Rangaswami, R., Zhao, M., Narasimhan, G.: Driving cache replacement with ML-based LeCaR. In: 10th USENIX Workshop on Hot Topics in Storage and File Systems. hotStorage'18, Boston, MA (2018). https://www.usenix.org/conference/hotstorage18/presentation/vietri
30. Rodriguez, L.V., Yusuf, F., Lyons, S., Paz, E., Rangaswami, R., Liu, J., Zhao, M., Narasimhan, G.: Learning Cache Replacement with CACHEUS. In: 19th USENIX Conference on File and Storage Technologies. FAST'21, pp. 341–354 (2021). https://www.usenix.org/conference/fast21/presentation/rodriguez
31. Song, Z., Chen, K., Sarda, Altınbüken, D., Brevdo, E., Coleman, J., Ju, X., Jurczyk, P., Schooler, R., Gummadi, R.: HALP: Heuristic aided learned preference eviction policy for YouTube Content Delivery Network. In: 20th USENIX Symposium on Networked Systems Design and Implementation. NSDI'23, pp. 1149–1163 (2023). https://www.usenix.org/conference/nsdi23/presentation/song-zhenyu
32. Yang, J., Qiu, Z., Zhang, Y., Yue, Y., Rashmi, K.V.: FIFO can be Better than LRU: the Power of Lazy Promotion and Quick Demotion. In: Proceedings of the 19th Workshop on Hot Topics in Operating Systems. HOTOS'23, pp. 70–79, New York, NY, USA (2023). https://doi.org/10.1145/3593856.3595887
33. Einziger, G., Eytan, O., Friedman, R., Manes, B.: Lightweight robust size aware cache management. ACM Trans. Storage **18**(3) (2022) https://doi.org/10.1145/3507920
34. Zhong, C., Zhao, X., Jiang, S.: LIRS2: an improved LIRS replacement algorithm. In: Proceedings of the 14th ACM International Conference on Systems And Storage. SYSTOR'21, pp. 1–12, Haifa Israel (2021). https://doi.org/10.1145/3456727.3463772
35. Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Noh, S.H., Sang Lyul Min, Yookun Cho, Chong Sang Kim: LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. IEEE Trans. Comput. 50(12), 1352–1361 (2001) https://doi.org/10.1109/TC.2001.970573
36. Kirilin, V., Sundarrajan, A., Gorinsky, S., Sitaraman, R.K.: RL-Cache: Learning-Based Cache Admission for Content Delivery. In: Proceedings of the 2019 Workshop on Network Meets AI & ML - NetAI'19. NetAI'19, pp. 57–63, Beijing, China (2019). https://doi.org/10.1145/3341216.3342214
37. Smaragdakis, Y., Kaplan, S., Wilson, P.: EELRU: simple and effective adaptive page replacement. ACM SIGMETRICS Perform. Eval. Rev. **27**(1), 122–133 (1999). https://doi.org/10.1145/301464.301486

38. Ananthanarayanan, G., Ghodsi, A., Warfield, A., Borthakur, D., Kandula, S., Shenker, S., Stoica, I.: Pacman: Coordinated Memory Caching for Parallel Jobs. In: 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), pp. 267–280 (2012)
39. Berger, D.S., Berg, B., Zhu, T., Sen, S., Harchol-Balter, M.: RobinHood: Tail latency aware caching – dynamic reallocation from Cache-Rich to Cache-Poor. In: 13th USENIX Symposium on Operating Systems Design and Implementation. OSDI'18, pp. 195–212, Carlsbad, CA (2018). https://www.usenix.org/conference/osdi18/presentation/berger
40. Yang, J., Yue, Y., Rashmi, K.V.: A large scale analysis of hundreds of in-memory cache clusters at Twitter. In: 14th USENIX Symposium on Operating Systems Design and Implementation. OSDI'20, pp. 191–208 (2020). https://www.usenix.org/conference/osdi20/presentation/yang
41. Yang, J., Yue, Y., Vinayak, R.: Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In: 18th USENIX Symposium on Networked Systems Design and Implementation. NSDI'21, pp. 503–518 (2021). https://www.usenix.org/conference/nsdi21/presentation/yang-juncheng
42. Sabnis, A., Sitaraman, R.K.: TRAGEN: a synthetic trace generator for realistic cache simulations. In: Proceedings of the 21st ACM Internet Measurement Conference. IMC'21, pp. 366–379, Virtual Event (2021). https://doi.org/10.1145/3487552.3487845
43. Cheng, A., Chu, D., Li, T., Chan, J., Crooks, N., Hellerstein, J.M., Stoica, I., Yu, X.: Take Out the TraChe: Maximizing (Tra)nsactional Ca(che) Hit Rate. OSDI23, pp. 419–439 (2023). https://www.usenix.org/conference/osdi23/presentation/cheng
44. McAllister, S., Berg, B., Tutuncu-Macias, J., Yang, J., Gunasekar, S., Lu, J., Berger, D.S., Beckmann, N., Ganger, G.R.: Kangaroo: Caching billions of tiny objects on flash. In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. SOSP'21, pp. 243–262, New York, NY, USA (2021). https://doi.org/10.1145/3477132.3483568
45. Sundarrajan, A., Feng, M., Kasbekar, M., Sitaraman, R.K.: Footprint Descriptors: Theory and Practice of Cache Provisioning in a Global CDN. In: Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies. CoNEXT'17, pp. 55–67, Incheon Republic of Korea (2017). https://doi.org/10.1145/3143361.3143368
46. Sabnis, A., Sitaraman, R.K.: JEDI: model-driven trace generation for cache simulations. In: Proceedings of the 22nd ACM Internet Measurement Conference. IMC'22, pp. 679–693, New York, NY, USA (2022). https://doi.org/10.1145/3517745.3561466
47. Chen, K., Bunt, R.B., Eager, D.L.: Write caching in distributed file systems. In: Proceedings of 15th International Conference on Distributed Computing Systems, pp. 457–466. IEEE (1995)
48. Zhu, T., Gandhi, A., Harchol-Balter, M., Kozuch, M.A.: Saving cash by using less cache. In: 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12) (2012)
49. Hafeez, U.U., Wajahat, M., Gandhi, A.: Elmem: Towards an elastic memcached system. In: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), pp. 278–289. IEEE (2018)
50. Berger, D.S., Henningsen, S., Ciucu, F., Schmitt, J.B.: Maximizing cache hit ratios by variance reduction. SIGMETRICS Perform. Eval. Rev. **43**(2), 57–59 (2015). https://doi.org/10.1145/2825236.2825259
51. Casale, G., Gast, N.: Performance analysis methods for list-based caches with non-uniform access. IEEE/ACM Trans. Netw. **29**(2), 651–664 (2021). https://doi.org/10.1109/TNET.2020.3042869
52. Gast, N., Van Houdt, B.: Transient and steady-state regime of a family of list-based cache replacement algorithms. In: Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems. SIGMETRICS '15, pp. 123–136 (2015). https://doi.org/10.1145/2745844.2745850
53. Carlsson, N., Eager, D.: Ephemeral content popularity at the edge and implications for on-demand caching. IEEE Trans. Parallel Distrib. Syst. **28**(6), 1621–1634 (2017). https://doi.org/10.1109/TPDS.2016.2614805
54. Ben-Ammar, H., Hadjadj-Aoul, Y., Rubino, G., Ait-Chellouche, S.: On the performance analysis of distributed caching systems using a customizable Markov chain model **130**, 39–51 (2019). https://doi.org/10.1016/j.jnca.2019.01.011
55. Kurose, J.: Information-centric networking: the evolution from circuits to packets to content. Comput. Netw. **66**, 112–120 (2014). https://doi.org/10.1016/j.comnet.2014.04.002. (**Leonard Kleinrock Tribute Issue: A Collection of Papers by his Students**)

56. Dan, A., Towsley, D.: An approximate analysis of the LRU and FIFO buffer replacement schemes. In: Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. SIGMETRICS '90, pp. 143–152. Association for Computing Machinery, New York, NY, USA (1990). https://doi.org/10.1145/98457.98525 . https://doi.org/10.1145/98457.98525
57. Rosensweig, E.J., Kurose, J., Towsley, D.: Approximate models for general cache networks. In: 2010 Proceedings IEEE INFOCOM, pp. 1–9 (2010). https://doi.org/10.1109/INFCOM.2010.5461936
58. Gast, N., Van Houdt, B.: TTL approximations of the cache replacement algorithms LRU(m) and h-LRU. Perform. Eval. **117**, 33–57 (2017). https://doi.org/10.1016/j.peva.2017.09.002
59. Che, H., Tung, Y., Wang, Z.: Hierarchical web caching systems: modeling, design and experimental results. IEEE J. Sel. Areas Commun. **20**(7), 1305–1314 (2002). https://doi.org/10.1109/JSAC.2002.801752
60. Fricker, C., Robert, P., Roberts, J.: A versatile and accurate approximation for LRU cache performance. In: 2012 24th International Teletraffic Congress (ITC 24), pp. 1–8 (2012)
61. Choungmo Fofack, N., Nain, P., Neglia, G., Towsley, D.: Performance evaluation of hierarchical TTL-based cache networks. Comput. Netw. **65**, 212–231 (2014). https://doi.org/10.1016/j.comnet.2014.03.006
62. Psaras, I., Clegg, R.G., Landa, R., Chai, W.K., Pavlou, G.: Modelling and evaluation of CCN-caching trees. In: NETWORKING 2011: 10th International IFIP TC 6 Networking Conference, Valencia, Spain, May 9-13, 2011, Proceedings, Part I 10, pp. 78–91. Springer (2011)
63. Tarnoi, S., Suppakitpaisarn, V., Kumwilaisak, W., Ji, Y.: Performance analysis of probabilistic caching scheme using markov chains. In: 2015 IEEE 40th Conference on Local Computer Networks (LCN), pp. 46–54 (2015). https://doi.org/10.1109/LCN.2015.7366282
64. Garetto, M., Leonardi, E., Martina, V.: A unified approach to the performance analysis of caching systems. ACM Trans. Model. Perform. Eval. Comput. Syst. **1**(3) (2016) https://doi.org/10.1145/2896380
65. HHVM Scalable Concurrent Cache. https://github.com/facebook/hhvm/blob/master/hphp/util/concurrent-scalable-cache.h. Accessed 20 Jan 2024
66. Memcached. memcached—a distributed memory object caching system. http://memcached.org/. Accessed 20 Jan 2024
67. Open CAS. Open Cache Acceleration Software. https://open-cas.github.io/. Accessed 20 Jan 2024
68. bcache. https://www.kernel.org/doc/Documentation/bcache.txt. Accessed 20 Jan 2024
69. Wu, C.-J., Jaleel, A., Hasenplaugh, W., Martonosi, M., Steely, S.C., Emer, J.: SHiP: signature-based hit predictor for high performance caching. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-44 '11. MICRO-44'11, p. 430, Porto Alegre, Brazil (2011). https://doi.org/10.1145/2155620.2155671
70. Jaleel, A., Theobald, K.B., Steely, S.C., Emer, J.: High performance cache replacement using re-reference interval prediction (RRIP). In: Proceedings of the 37th Annual International Symposium on Computer Architecture. ISCA'10, pp. 60–71, New York, NY, USA (2010). https://doi.org/10.1145/1815961.1815971
71. Multi-core Processor. https://en.wikipedia.org/wiki/Multi-core_processor. Accessed 20 Jan 2024
72. AMD: AMD EPYC 4th gen CPU list. https://www.amd.com/en/products/processors/server/epyc/4th-generation-9004-and-8004-series.html. Accessed 10 Jan 2024
73. Samsung: Ultra-Low Latency with Samsung Z-NAND SSD. https://download.semiconductor.samsung.com/resources/brochure/Ultra-Low%20Latency%20with%20Samsung%20Z-NAND%20SSD.pdf. Accessed 10 Jan 2024
74. Intel: Accelerate Data Center Storage and Memory Performance. https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/data-center-ssds/optane-ssd-p5800x-p5801x-brief.html. Accessed 10 Jan 2024
75. Solidigm: Solidigm P44 Pro Series – Exceptional Performanc. https://www.bhphotovideo.com/lit_files/1022190.pdf. Accessed 10 Jan 2024
76. Anadtech: Intel SSD DC P3700 Review: The PCIE transition begins with NVMe. https://download.semiconductor.samsung.com/resources/brochure/Ultra-LowLatencywithSamsungZ-NANDSSD.pdf. Accessed 10 Jan 2024
77. Intel: Intel Solid-State Drive DC S3700. https://download.intel.com/newsroom/kits/ssd/pdfs/Intel_SSD_DC_S3700_Product_Specification.pdf. Accessed 10 Jan 2024
78. Micron: Micron 7450 SSD With NVMe. https://media-www.micron.com/-/media/client/global/documents/products/product-flyer/7450_nvme_ssd_product_brief.pdf. Accessed 10 Jan 2024

79. Louwrentius: Understanding Storage Performance - IOPS and Latency. https://louwrentius.com/understanding-storage-performance-iops-and-latency.html. Accessed 10 Jan 2024
80. Zhang, X., Pei, S., Choi, J., Kim, B.S.: Excessive SSD-internal parallelism considered harmful. In: Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems. HotStorage '23, pp. 65–72. Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3599691.3603412 . https://doi.org/10.1145/3599691.3603412
81. Chen, F., Hou, B., Lee, R.: Internal parallelism of flash memory-based solid-state drives. ACM Trans. Storage **12**(3) (2016) https://doi.org/10.1145/2818376
82. Manes, B.: Caffeine: A high performance caching library for Java (2014–2025). https://github.com/ben-manes/caffeine
83. Dgraph Labs: Ristretto: A high performance memory-bound Go cache (2019–2025). https://github.com/dgraph-io/ristretto
84. RocksDB: RocksDB Block Cache - HyperCLOCKCache. https://github.com/facebook/rocksdb/wiki/Block-Cache#hyperclockcache. Accessed 09 Mar 2026
85. RocksDB: RocksDB Block Cache - LRU Cache. https://github.com/facebook/rocksdb/wiki/Block-Cache#lru-cache. Accessed 09 Mar 2026
86. Adas, D., Einziger, G., Friedman, R.: Limited associativity makes concurrent software caches a breeze. In: Proceedings of the 23rd International Conference on Distributed Computing and Networking, pp. 87–96. ACM (2022). https://doi.org/10.1145/3491003.3491013
87. Lim, H., Han, D., Andersen, D.G., Kaminsky, M.: MICA: A holistic approach to fast In-Memory Key-Value storage. In: 11th USENIX Symposium on Networked Systems Design and Implementation. NSDI'14, pp. 429–444, Seattle, WA (2014). https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim
88. Blankstein, A., Sen, S., Freedman, M.J.: Hyperbolic caching: Flexible caching for web applications. In: 2017 USENIX Annual Technical Conference. ATC'17, pp. 499–511, Santa Clara, CA (2017). https://www.usenix.org/conference/atc17/technical-sessions/presentation/blankstein
89. Zhang, Y., Yang, J., Yue, Y., Vigfusson, Y., Rashmi, K.V.: SIEVE is simpler than LRU: an efficient turn-key eviction algorithm for web caches. In: 21th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2024) (2024)
90. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing - SoCC '10. SoCC'10, p. 143, Indianapolis, Indiana, USA (2010). https://doi.org/10.1145/1807128.1807152
91. Qiu, Z., Yang, J., Zhang, J., Li, C., Ma, X., Chen, Q., Yang, M., Xu, Y.: FrozenHot Cache: Rethinking Cache Management for Modern Hardware. In: Proceedings of the Eighteenth European Conference on Computer Systems. EuroSys'23, pp. 557–573, New York, NY, USA (2023). https://doi.org/10.1145/3552326.3587446
92. Harchol-Balter, M.: Performance Modeling and Design of Computer Systems: Queueing Theory in Action. Cambridge University Press, New York (2013)
93. Schroeder, B., Wierman, A., Harchol-Balter, M.: Open versus closed: a cautionary tale. In: Proceedings of the 3rd Conference on Networked Systems Design & Implementation-Volume 3. Volume3, pp. 18–18 (2006)
94. Cloudlab. https://www.cloudlab.us/. Accessed 20 Jan 2024
95. Chen, J., Chen, L., Wang, S., Zhu, G., Sun, Y., Liu, H., Li, F.: HotRing: A Hotspot-Aware In-Memory Key-Value Store. In: 18th USENIX Conference on File and Storage Technologies. FAST'20, pp. 239–252 (2020). https://www.usenix.org/conference/fast20/presentation/chen-jiqiang
96. Huang, Q., Birman, K., Renesse, R., Lloyd, W., Kumar, S., Li, H.C.: An analysis of Facebook photo caching. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. SOSP'13, pp. 167–181, New York, NY, USA (2013). https://doi.org/10.1145/2517349.2522722
97. Carr, R.W., Hennessy, J.L.: WSCLOCK: a simple and effective algorithm for virtual memory management. In: Proceedings of the Eighth ACM Symposium on Operating Systems Principles. SOSP'81, pp. 87–95, New York, NY, USA (1981). https://doi.org/10.1145/800216.806596
98. Berger, D.S., Sitaraman, R.K., Harchol-Balter, M.: AdaptSize: Orchestrating the hot object memory cache in a content delivery network. In: 14th USENIX Symposium on Networked Systems Design and Implementation. NSDI'17, pp. 483–498 (2017)
99. Yang, T.-W., Pollen, S., Uysal, M., Merchant, A., Wolfmeister, H.: CacheSack: Admission Optimization for Google Datacenter Flash Caches. In: 2022 USENIX Annual Technical Conference. ATC'22,

pp. 1021–1036, Carlsbad, CA (2022). https://www.usenix.org/conference/atc22/presentation/yang-tzu-wei
100. Berger, D.S.: Towards Lightweight and Robust Machine Learning for CDN Caching. In: Proceedings of the 17th ACM Workshop on Hot Topics In Networks. Hotnets'18, pp. 134–140, Redmond WA USA (2018). https://doi.org/10.1145/3286062.3286082
101. Eisenman, A., Cidon, A., Pergament, E., Haimovich, O., Stutsman, R., Alizadeh, M., Katti, S.: Flashield: a hybrid key-value cache that controls flash write amplification. In: 16th USENIX Symposium on Networked Systems Design and Implementation. NSDI'19, pp. 65–78, Boston, MA (2019). https://www.usenix.org/conference/nsdi19/presentation/eisenman
102. Gill, B.S., Bathen, L.A.D.: AMP: Adaptive multi-stream prefetching in a shared cache. In: FAST, vol. 7, pp. 185–198 (2007)
103. Li, Z., Chen, Z., Srinivasan, S.M., Zhou, Y.: C-Miner: mining block correlations in storage systems. In: Proceedings of the 3rd USENIX Conference on File and Storage Technologies. FAST'04, p. 13, USA (2004)
104. Plonka, D., Barford, P.: Context-aware clustering of DNS query traffic. In: Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement Conference - IMC '08, Vouliagmeni, Greece, p. 217 (2008). https://doi.org/10.1145/1452520.1452547
105. Yang, J., Mao, Z., Yue, Y., Rashmi, K.V.: GL-Cache: Group-level learning for efficient and high-performance caching. In: 21st USENIX Conference on File and Storage Technologies (FAST 23). FAST'23, pp. 115–134 (2023). https://www.usenix.org/conference/fast23/presentation/yang-juncheng
106. Azim, T., Karpathiotakis, M., Ailamaki, A.: ReCache: reactive caching for fast analytics over heterogeneous data. Proceedings of the VLDB Endowment 11(3), 324–337 (2017) https://doi.org/10.14778/3157794.3157801
107. Akhtar, Z., Li, Y., Govindan, R., Halepovic, E., Hao, S., Liu, Y., Sen, S.: AViC: a cache for adaptive bitrate video. In: Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies. CoNEXT'19, pp. 305–317, Orlando Florida (2019). https://dl.acm.org/doi/10.1145/3359989.3365423
108. Liu, Z., Bai, Z., Liu, Z., Li, X., Kim, C., Braverman, V., Jin, X., Stoica, I.: DistCache: Provable load balancing for Large-Scale storage systems with distributed caching. In: 17th USENIX Conference on File and Storage Technologies. FAST'19, pp. 143–157, Boston, MA (2019). https://www.usenix.org/conference/fast19/presentation/liu
109. Pan, C., Luo, Y., Wang, X., Wang, Z.: pRedis: Penalty and Locality Aware Memory Allocation in Redis. In: Proceedings of the ACM Symposium on Cloud Computing. SOCC'19, pp. 193–205, Santa Cruz CA USA (2019). https://doi.org/10.1145/3357223.3362729
110. Wu, K., Guo, Z., Hu, G., Tu, K., Alagappan, R., Sen, R., Park, K., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus. In: 19th USENIX Conference on File and Storage Technologies. FAST'21, pp. 307–323 (2021)
111. Song, Z., Berger, D.S., Li, K., Shaikh, A., Lloyd, W., Ghorbani, S., Kim, C., Akella, A., Krishnamurthy, A., Witchel, E., et al.: Learning relaxed belady for content distribution network caching. In: 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), pp. 529–544 (2020)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.