Can Increasing the Hit Ratio Hurt Cache Throughput?

Ziyue Qiu, Juncheng Yang, and Mor Harchol-Balter*

Carnegie Mellon University

Abstract. Software caches are an intrinsic component of almost every computer system. Consequently, caching algorithms, particularly eviction policies, are the topic of many papers. Almost all 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 request latency. Intuitively it makes sense that higher hit ratio should lead to higher throughput (and lower request latency), since more requests are found in the cache (low access time) as opposed to the disk (high access time).
This paper challenges this intuition. We show that increasing the hit ratio can actually *hurt* the throughput (and request latency) for many caching 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 towards faster disks and more cores per CPU.

Keywords: caches, hit ratio, performance evaluation, scalability, modification analysis, queueing theory, LRU, eviction policies

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 [22], Google CliqueMap [85], and include many types of caches in between [14, 28, 94].

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 [84]. The principle of caching is very simple: Store items that are likely to be accessed soon in the cache. Store everything else on disk.

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.

* Supported by NSF-CIF-2403194, NSF-III-2322973, and NSF-CMMI-2307008.

Table 1: Table shows the algorithms we evaluated. Detailed descriptions of the algorithms are in Sec. 4. A classification of 12 additional algorithms is given in Table 2.

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 [13], RocksDB [77], LevelDB [10]	no
FIFO	Evicts the oldest item.	ATS [18]	yes
Probabilistic LRU	Only moves accessed item to the head of queue with some probability $1 - q$.	HHVM [3]	depends on q
FIFO-Reinsertion a.k.a. CLOCK	Item at end of queue gets a second chance through the queue before eviction.	RocksDB [77]	yes
Segmented LRU	Uses two LRU queues to differentiate items that have been accessed twice.	Linux [87]	no
S3-FIFO [100]	Uses a small FIFO queue to evict most new and unpopular objects.	RedPanda	yes

Examples include the caches used in big-data systems (Hadoop, HDFS [53], Alluxio [13]), deep learning systems (Pytorch [74]), and databases (RocksDB [42]).

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. The *most common* cache eviction algorithm is Least-Recently-Used (LRU) [8,11,47,70], which evicts the least recently *accessed* item in the cache. Because of its popularity, this paper will focus on LRU cache eviction. LRU is widely used because data access patterns often show locality where recently used data have a higher chance to get reused [40, 41]. Another common algorithm is First-In-First-Out (FIFO), which evicts the least recently *inserted* item, i.e., the oldest item. Many more advanced eviction algorithms also exist [17, 20, 24, 43–45, 59–61, 68, 78, 86, 89, 90, 93, 97, 104].

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 [22, 35, 36, 54, 67, 80, 81, 91, 94, 98, 99, 105]. Maximizing the hit

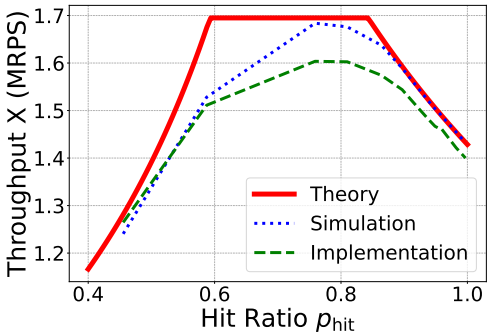


Fig. 1: Throughput under LRU (measured in millions of requests per second) increases as the hit ratio increases initially but then drops when the hit ratio gets high.

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

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

A. Queueing model for upper bounding throughput While many papers have analyzed the cache *hit ratio*, e.g., [21,25,29,31,32,37,39,48–51,62,73,79,92], 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.

Instead we develop a 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. For example, our bound for the case of LRU is given in Figure 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.

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 Figure 1 via the blue dotted line.

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 [3, 4], 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 [22], Memcached [5], Intel OCF [7], BCache [1], and RocksDB LRU Cache [77]. For LRU, the result of our implementation is shown via the green dashed line in Figure 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 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 Figures 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.

1.5 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, such models do not exist in prior work.
- 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 Figure 12.
– 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 Figure 12.

2 Background

DRAM-based software caches: As explained in Section 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. This is in contrast to SSD-based software caches where the cache access is 100 times slower and far less concurrent than in the DRAM system. This paper focuses on DRAM-based software caches. Hardware caches are also outside the scope of this paper as are network-connected caches.
Hardware 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 [6], but a modern CPU has 32-192 cores [15]. 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. Today's backend disks are implemented on SSDs, where high-end SSDs have latencies around 5 μs [57, 82], low-end SSDs show latencies of a few hundred μs [16, 88], and most commercial SSDs have latency in between [58,65,69]. These backend disks support massive concurrency [33,102], allowing requests at the disk to all be served in parallel without queueing.

3 A three-pronged approach to LRU

Because LRU is the most common caching policy, 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 Section 3.4. Takeaways are summarized in Section 3.5.

3.1 A closed-loop queueing model of LRU

As with many caching systems evaluations and benchmarks, e.g., Cachelib [22], YCSB benchmark [38], S3-FIFO [100], and FrozenHot [76], 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).

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 [55, Chapters 2,6,7] for background on modeling closed-loop queueing models.

Modeling disk access and cache access Our disk has enough concurrency that it can be accessed simultaneously by all 72 requests. Thus, in queueing speak, we can model 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.

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 Figure 2. We measure the latency (service time) of each operation and indicate the means in Figure 2. From a queueing theory standpoint, only the mean service times matter, and it suffices to have an upper bound on service times for those devices that are not bottleneck devices. The measurement process is non-trivial and the details are described in the full paper [75].

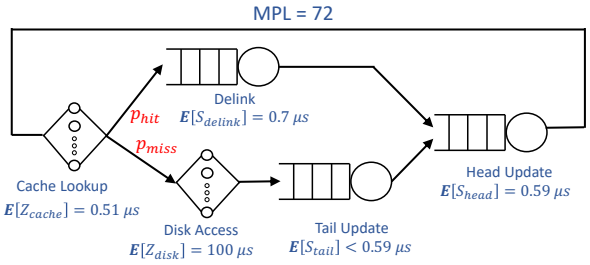


Fig. 2: Queueing model of LRU cache.

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}. 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. Our analysis of closed systems is based on [55, Chapters 6,7]) and produces an upper bound on throughput for the queueing network in Figure 2.

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:

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

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:

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

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:

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

We use X to denote system throughput. From [55, Theorem 7.1], we know that X is upper-bounded by two terms, as follows:

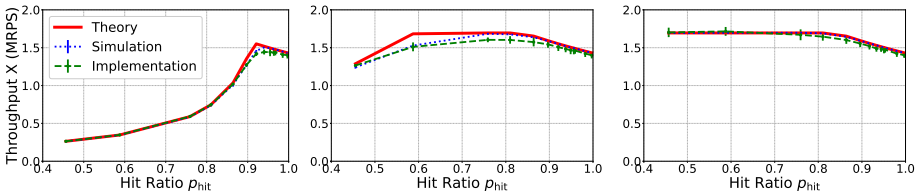
$$X \leq \min\left(\frac{N}{D + \mathbf{E}[Z]}\ ,\ \frac{1}{D_{max}}\right).$$

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 $\mathbf{E}[Z_{disk}] = 100\mu s$:

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

Equation (1) represents an *upper bound on throughput*, shown in red in Figure 3(b). This turns out to be a very good bound on the measured throughput from our implementation. When $p_{hit} < 0.59$, the first term in (1) is the relevant bound (minimum term). 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. When $p_{hit} > 0.84$, the second term in (1) is again the relevant bound, but the max term in the denominator is $0.7p_{hit}$.

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. 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 $101.69 - 99.89p_{hit}$.



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

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 towards lower disk latencies – from (a) to (b) to (c).

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 Figure 3(c).

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

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

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

Discussion of the analytic results. 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. From a more intuitive perspective, when the hit ratio is high, we see that the *delink* operation becomes the bottleneck. Hence almost all requests are queued behind the delink server in Figure 2. Thus, while it seems that we are saving time by not going to disk, we instead are wasting time by having to queue up at the delink device. Thus requests can actually take *longer*. This longer request latency translates to a drop in throughput.

In all the curves of Figure 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.

Throughout, we have looked at throughput, but we could instead have looked at *mean latency*, namely the time from when a request is submitted until it completes. Given that we have a closed-loop setting, mean request latency and throughput are inversely related (see [55, Chapters 6,7]). Hence mean request latency *increases* for higher hit ratios.

3.3 Simulation evaluation

Recall that our queueing analysis from Section 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 [83] for closed-loop models.

The results of our simulation are shown in Figure 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 (Section 3.2) and the simulation (Section 3.3), we were evaluating the queueing network in Figure 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.

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

Experimental setup: Our experiments use dual-socket servers with Intel Xeon Platinum 8360Y 36-core processors (Ubuntu 20) at 2.4GHz, running on CloudLab platform [2]. 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 [34] and social networks [98]. 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 (Section 3.1). We were able to derive an *upper bound* on throughput in our model as a function of the hit ratio (Section 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 on the delink operation, resulting in longer delays and lower throughout. We next simulated the queueing network (Section 3.3), determining the exact throughput as a function of hit ratio. Finally, in Section 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.

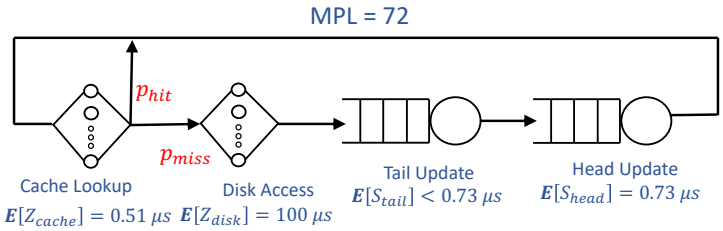


Fig. 4: Queueing model of FIFO cache.

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.

We can thus model the FIFO caching system via the closed-loop queueing model in Figure 4. It may seem strange that $\mathbf{E}[S_{head}]$ is higher in the FIFO system (Figure 4) than in the LRU system (Figure 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.

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:

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

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

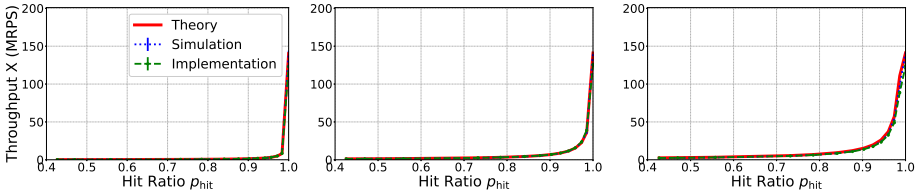
$$D_{tail} < (1 - p_{hit}) \cdot 0.73 \qquad D_{head} = (1 - p_{hit}) \cdot 0.73$$

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:

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



(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.

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

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

Given all the above terms, where $\mathbf{E}[Z_{disk}] = 100\mu s$, from [55, Theorem 7.1] our upper bound on throughput X is:

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

Equation (4) represents an *upper bound on throughput*, shown in red Figure 5(b). 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:

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

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

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

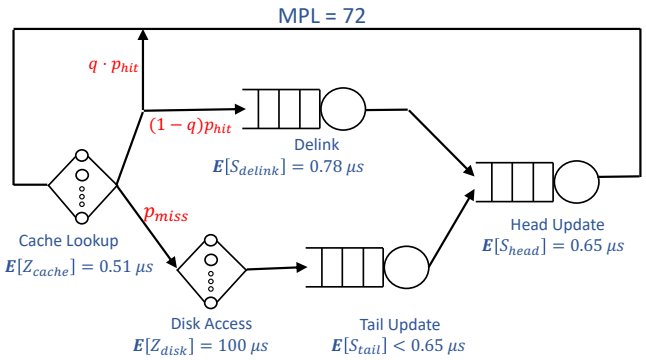
The bounds in (4), (5), and (6) are shown via the red solid lines in Figure 5.

Results of the three-pronged approach. 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. More intuitively, the bottleneck device (the head update) is now always in the *miss* path, not

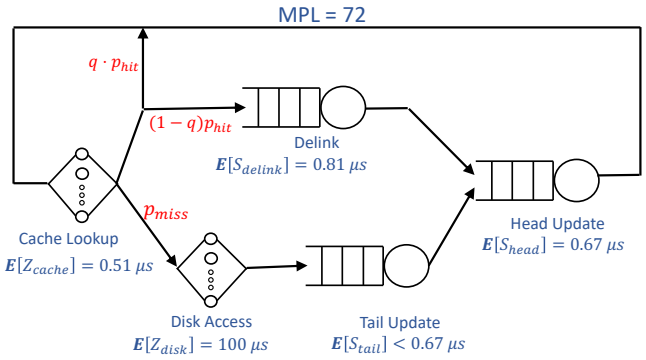
in the hit path. Therefore, increasing the hit ratio does not result in increased demand on the bottleneck device, and hence does not lead to deleterious effects on throughput. Our simulation of the queueing network follows the same process as Section 3.3, and our implementation follows the process of Section 3.4. Results of simulation are shown in blue dotted lines and results of implementation are shown via green dashed lines in Figure 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.



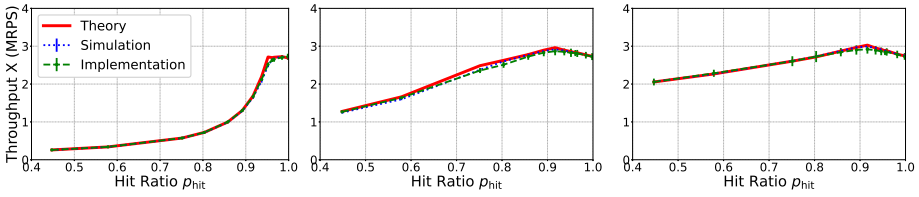
(a) Queueing model of Probabilistic LRU cache with $q = 0.5$.



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

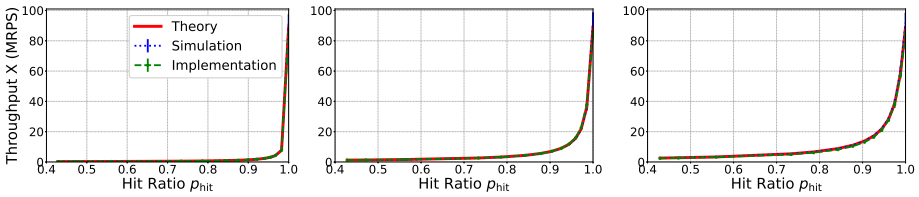
Fig. 6: Queueing model of Probalistic LRU cache.

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.



(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.

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).

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 Sections 3.1. The queueing network for Probabilistic LRU is shown in the case of $q = 0.5$ (Figure 6(a)) and $q = 1 - \frac{1}{72}$ (Figure 6(b)). 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.

We defer the analysis of the queueing model to the full paper [75].

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.

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, which becomes the bottleneck, resulting in higher queueing times and reduced throughput.

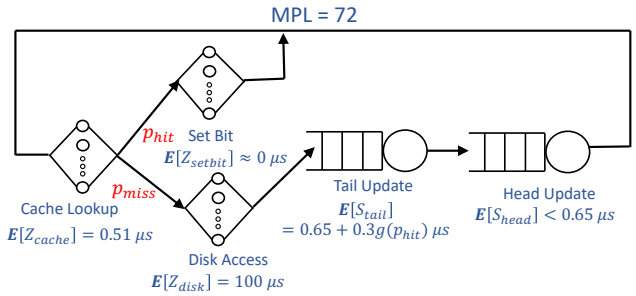
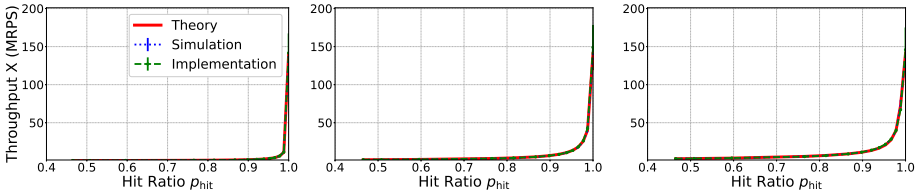


Fig. 9: Queueing model of CLOCK cache.



(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.

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 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, 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 towards 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 Figure 9.

We defer the analysis of the queueing model to the full paper [75].

Results of the three-pronged approach. Figure 10 shows the result of the theory, simulation, and implementation of CLOCK. 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 (the tail update) is in the miss path, not in the hit path.

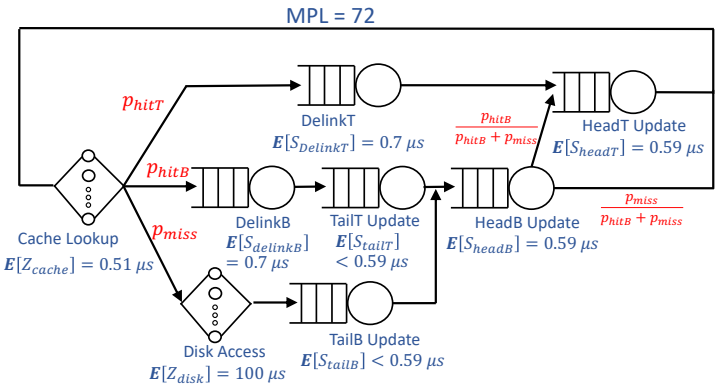


Fig. 11: Queueing model of Segmented LRU cache.

Therefore, increasing the hit ratio this does not result in increased demand 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 [56]. 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.

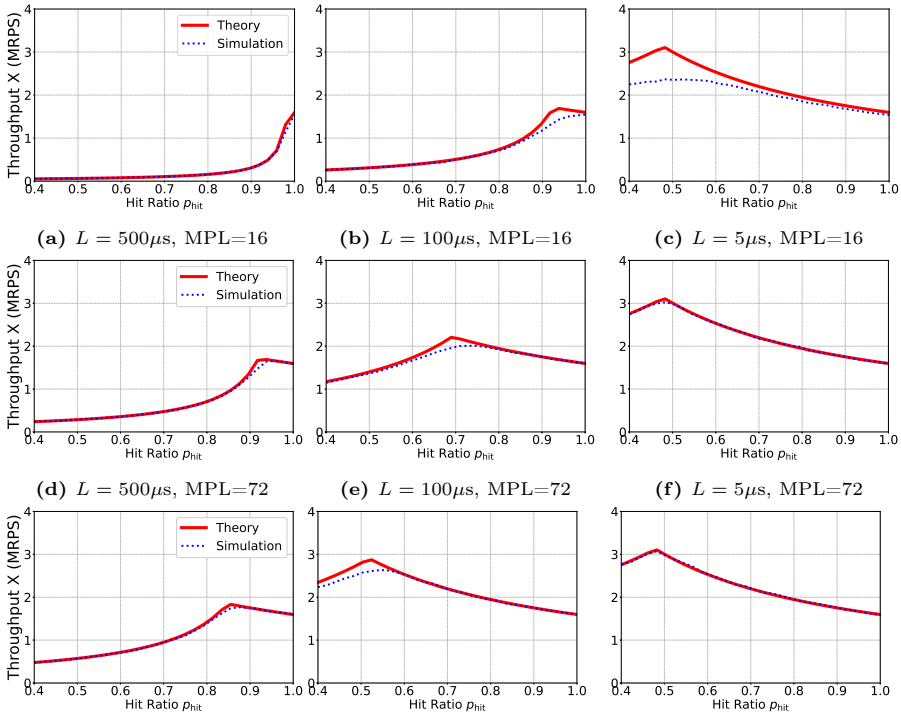
The queueing network for SLRU is shown in Figure 11. When an object d is first accessed, we do a cache lookup. There are now three cases of what might happen. If d is currently in the T list (which runs LRU), then d needs to be delinked from its current location and moved to the head of the B list (thus 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.

To do a queueing analysis of the network in Figure 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. The details are in the full paper [75].

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.

We also evaluate the effect of another trend, increasing the number of CPU cores, namely the Multi-Programming Level. The effect of *increasing the MPL* is shown in Figure 12 when looking from top (MPL = 16) to bottom (MPL = 144). We see that the point at which throughput starts to deteriorate, p^*_{hit}, moves earlier for higher MPL, as well as lower disk latency. This effect makes sense since a higher level of concurrency means that the delink operation becomes bottlenecked sooner.



(a) $L = 500\mu$s, MPL=16 **(b)** $L = 100\mu$s, MPL=16 **(c)** $L = 5\mu$s, MPL=16

(d) $L = 500\mu$s, MPL=72 **(e)** $L = 100\mu$s, MPL=72 **(f)** $L = 5\mu$s, MPL=72

(g) $L = 500\mu$s, MPL=144 **(h)** $L = 100\mu$s, MPL=144 **(i)** $L = 5\mu$s, MPL=144

Fig. 12: The throughput of a Segmented LRU cache is affected by the hit ratio, but also by the disk latency (L) and the MPL.

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. Therefore, increasing hit ratio makes more requests queue behind the delinkT server which is already bottlenecked, hence lowering the throughput. Note that this same behavior would happen if the

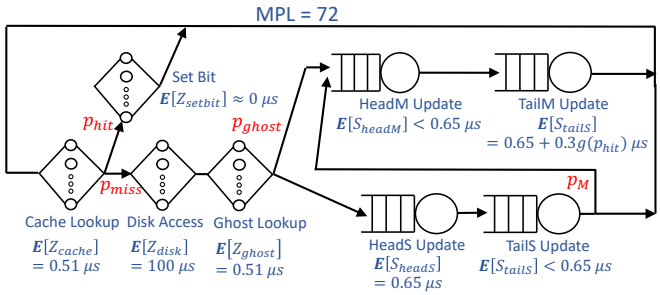


Fig. 13: Queueing model of S3-FIFO cache.

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.

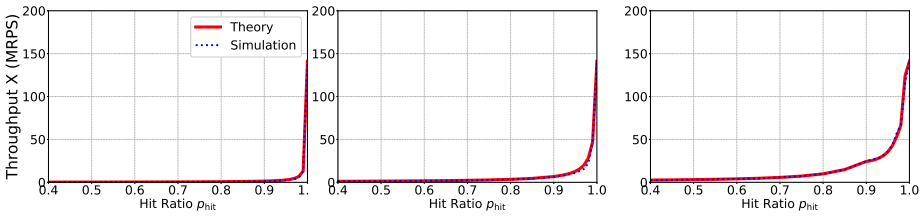
4.5 S3-FIFO

S3-FIFO [100] 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). 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.

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 $\mathbf{E}[Z_{ghost}] = 0.51\mu s$.

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.



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

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.

To do a queueing analysis of the network in Figure 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}. See the full paper for details [75].

Results of analysis and simulation. Figure 14 shows the results of analysis (red solid) and simulation (blue dotted) for S3-FIFO. 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 in the *miss* path, not in the hit path. Hence, increasing p_{hit} does not increase demand on the bottleneck device.

5 Discussion

5.1 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.

Table 2: Conjectured classification of additional algorithms.

LRU like	ARC [68], LIRS [59], TinyLFU [45], LeCaR [93], CACHEUS [78], LFU [9, 66]
FIFO like	CLOCK variants [30], SIEVE [103], QDLP [97], Hyperbolic [27], Random, LHD [20], LRB [89]

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 [30], S3-FIFO [100], QDLP [97], and SIEVE [103]. Algorithms in the second type do not have a global data structure, but rather uses random sampling to choose eviction candidates. Examples include LHD [20], LRB [89], and Hyperbolic [27]. As these algorithms do not maintain a global data structure, they are never bottlenecked by cache hits, so throughput only increases with p_{hit}.

5.2 Improving future caching systems

Future CPUs will have more cores, further increasing their concurrency. At the same time, disks are getting faster, with high-end disks having single-digit microseconds of latency. Figure 12 shows the effects of these two trends. We see that the critical hit ratio, p^*_{hit}, after which throughput drops, will shift to smaller values in the future. Therefore, fixing the problems of LRU-like algorithms, which are the predominant caching algorithms used today, will become more important.

The easiest mitigation is to reduce the number of delink operations. This can be achieved by performing the delink operation probabilistically (probabilistic LRU). As we have shown in Section 4, when q (the probability of skipping the delink queue) is very high, probabilistic LRU behaves like FIFO, where increasing the hit ratio only helps. This is the high-level idea behind a very recent paper, [76].

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

One might be thinking at this point that, in a future with higher concurrency and faster disks, one might want to 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 very recent papers (2023, 2024) which go in this direction [100, 103].

6 Prior Work

DRAM-based caches are increasingly deployed in today's system stack and are widely studied in research [12,19,23,26,27,36,46,52,61,63,70,72,78,90,93,96,101].

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 [64], p-redis [71], LRB [89], GL-Cache [96]. 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 [76], the authors fix the hit ratio at 99% and do not study throughput as a function of changing hit ratio. The second paper, NHC [95], 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.

7 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 illucidates 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 increases 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.

References

1. bcache. https://www.kernel.org/doc/Documentation/bcache.txt. Accessed Jan 20, 2024.

2. Cloudlab. https://www.cloudlab.us/. Accessed Jan 20, 2024.
3. HHVM project. https://github.com/facebook/hhvm.git. Accessed Jan 20, 2024.
4. HHVM Scalable Concurrent Cache. https://github.com/facebook/hhvm/blob/master/hphp/util/concurrent-scalable-cache.h. Accessed Jan 20, 2024.
5. Memcached. memcached - a distributed memory object caching system. http://memcached.org/. Accessed Jan 20, 2024.
6. Multi-core Processor. https://en.wikipedia.org/wiki/Multi-core_processor. Accessed Jan 30, 2024.
7. Open CAS. Open Cache Acceleration Software. https://open-cas.github.io/. Accessed Jan 20, 2024.
8. Cachelib eviction policies. https://cachelib.org/docs/Cache_Library_User_Guides/eviction_policy. Accessed: 2024-04-22.
9. Least frequently used. https://en.wikipedia.org/wiki/Least_frequently_used. Accessed: 2024-11-26.
10. Leveldb. https://github.com/google/leveldb. Accessed: 2024-04-22.
11. Redis eviction policies. https://redis.io/blog/cache-eviction-strategies/. Accessed: 2024-04-22.
12. Zahaib Akhtar, Yaguang Li, Ramesh Govindan, Emir Halepovic, Shuai Hao, Yan Liu, and Subhabrata Sen. AViC: a cache for adaptive bitrate video. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT'19, pages 305–317, Orlando Florida, December 2019.
13. Alluxio. Alluxio - data orchestration for the cloud. https://www.alluxio.io/. Accessed: 2024-01-10.
14. Jussara M Almeida, Derek L Eager, and Mary K Vernon. Hybrid caching strategy for streaming media files. In *Multimedia Computing and Networking 2001*, volume 4312, pages 200–212. SPIE, 2000.
15. AMD. AMD EPYC 4th gen CPU list. https://www.amd.com/en/products/processors/server/epyc/4th-generation-9004-and-8004-series.html. Accessed: 2024-01-10.
16. Anadtech. Intel SSD DC p3700 review: The PCIE transition begins with NVMe. https://download.semiconductor.samsung.com/resources/brochure/Ultra-Low%20Latency%20with%20Samsung%20Z-NAND%20SSD.pdf. Accessed: 2024-01-10.
17. Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Warfield, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated Memory Caching for Parallel Jobs. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 267–280, 2012.
18. Apache. Apache traffic server. https://trafficserver.apache.org/. Accessed: 2024-01-10.
19. Tahir Azim, Manos Karpathiotakis, and Anastasia Ailamaki. ReCache: reactive caching for fast analytics over heterogeneous data. *Proceedings of the VLDB Endowment*, 11(3):324–337, November 2017.
20. Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX symposium on networked systems design and implementation*, NSDI'18, pages 389–403, 2018.
21. Hamza Ben-Ammar, Yassine Hadjadj-Aoul, Gerardo Rubino, and Soraya Ait-Chellouche. On the performance analysis of distributed caching systems using a customizable markov chain model. volume 130, pages 39–51, 2019.
22. Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib caching engine: Design and ex-

periences at scale. In *14th USENIX symposium on operating systems design and implementation*, OSDI'20, pages 753–768, November 2020.
23. Daniel S. Berger. Towards Lightweight and Robust Machine Learning for CDN Caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, Hotnets'18, pages 134–140, Redmond WA USA, November 2018.
24. Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 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, pages 195–212, Carlsbad, CA, October 2018.
25. Daniel S. Berger, Sebastian Henningsen, Florin Ciucu, and Jens B. Schmitt. Maximizing cache hit ratios by variance reduction. *SIGMETRICS Perform. Eval. Rev.*, 43(2):57–59, September 2015.
26. Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. AdaptSize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX symposium on networked systems design and implementation*, NSDI'17, pages 483–498, 2017.
27. Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX annual technical conference*, ATC'17, pages 499–511, Santa Clara, CA, July 2017.
28. Sem Borst, Varun Gupta, and Anwar Walid. Distributed caching algorithms for content distribution networks. In *2010 Proceedings IEEE INFOCOM*, pages 1–9. IEEE, 2010.
29. Niklas Carlsson and Derek Eager. Ephemeral content popularity at the edge and implications for on-demand caching. *IEEE Transactions on Parallel and Distributed Systems*, 28(6):1621–1634, 2017.
30. Richard W. Carr and John L. Hennessy. WSCLOCK: a simple and effective algorithm for virtual memory management. In *Proceedings of the eighth ACM symposium on Operating systems principles*, SOSP'81, pages 87–95, New York, NY, USA, December 1981.
31. Giuliano Casale and Nicolas Gast. Performance analysis methods for list-based caches with non-uniform access. *IEEE/ACM Transactions on Networking*, 29(2):651–664, 2021.
32. Hao Che, Ye Tung, and Zhijun Wang. Hierarchical web caching systems: modeling, design and experimental results. *IEEE Journal on Selected Areas in Communications*, 20(7):1305–1314, 2002.
33. Feng Chen, Binbing Hou, and Rubao Lee. Internal parallelism of flash memory-based solid-state drives. *ACM Trans. Storage*, 12(3), June 2016.
34. Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. HotRing: A Hotspot-Aware In-Memory Key-Value Store. In *18th USENIX Conference on File and Storage Technologies*, FAST'20, pages 239–252, 2020.
35. Kerhong Chen, Richard B Bunt, and Derek L Eager. Write caching in distributed file systems. In *Proceedings of 15th International Conference on Distributed Computing Systems*, pages 457–466. IEEE, 1995.
36. Audrey Cheng, David Chu, Terrance Li, Jason Chan, Natacha Crooks, Joseph M. Hellerstein, Ion Stoica, and Xiangyao Yu. Take Out the TraChe: Maximizing (Tra)nsactional Ca(che) Hit Rate. OSDI23, pages 419–439, 2023.
37. Nicaise Choungmo Fofack, Philippe Nain, Giovanni Neglia, and Don Towsley. Performance evaluation of hierarchical TTL-based cache networks. *Computer Networks*, 65:212–231, 2014.

38. Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, SoCC'10, page 143, Indianapolis, Indiana, USA, 2010.
39. Asit Dan and Don Towsley. 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, page 143–152, New York, NY, USA, 1990.
40. Peter J Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.
41. P.J. Denning. Working Sets Past and Present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.
42. Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Transactions on Storage*, 17(4):26:1–26:32, October 2021.
43. Donghee Lee, Jongmoo Choi, Jong-Hun Kim, S.H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1361, December 2001.
44. Gil Einziger, Ohad Eytan, Roy Friedman, and Benjamin Manes. Lightweight robust size aware cache management. *ACM Transactions on Storage*, 18(3), August 2022.
45. Gil Einziger, Roy Friedman, and Ben Manes. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Transactions on Storage*, 13(4):1–31, December 2017.
46. Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX symposium on networked systems design and implementation*, NSDI'19, pages 65–78, Boston, MA, February 2019.
47. Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen Kat. It's time to revisit LRU vs. FIFO. In *12th USENIX workshop on hot topics in storage and file systems*, hotStorage'20, July 2020.
48. Christine Fricker, Philippe Robert, and James Roberts. A versatile and accurate approximation for LRU cache performance. In *2012 24th International Teletraffic Congress (ITC 24)*, pages 1–8, 2012.
49. Michele Garetto, Emilio Leonardi, and Valentina Martina. A unified approach to the performance analysis of caching systems. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 1(3), May 2016.
50. Nicolas Gast and Benny Van Houdt. 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, page 123–136, 2015.
51. Nicolas Gast and Benny Van Houdt. TTL approximations of the cache replacement algorithms LRU(m) and h-LRU. *Performance Evaluation*, 117:33–57, 2017.
52. Binny S Gill and Luis Angel D Bathen. AMP: Adaptive multi-stream prefetching in a shared cache. In *FAST*, volume 7, pages 185–198, 2007.
53. Hadoop. HDFS architecture guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. Accessed: 2024-01-10.
54. Ubaid Ullah Hafeez, Muhammad Wajahat, and Anshul Gandhi. Elmem: Towards an elastic memcached system. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 278–289. IEEE, 2018.

55. Mor Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.
56. Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP'13, pages 167–181, New York, NY, USA, November 2013.
57. 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: 2024-01-10.
58. Intel. Intel solid-state drive DC s3700. https://download.intel.com/newsroom/kits/ssd/pdfs/Intel_SSD_DC_S3700_Product_Specification.pdf. Accessed: 2024-01-10.
59. Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *ACM SIGMETRICS Performance Evaluation Review*, volume 30 of *SIGMETRICS'02*, pages 31–42, June 2002.
60. Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB'94, pages 439–450, San Francisco, CA, USA, September 1994.
61. Vadim Kirilin, Aditya Sundarrajan, Sergey Gorinsky, and Ramesh K. Sitaraman. RL-Cache: Learning-Based Cache Admission for Content Delivery. In *Proceedings of the 2019 Workshop on Network Meets AI & ML - NetAI'19*, NetAI'19, pages 57–63, Beijing, China, 2019.
62. Jim Kurose. Information-centric networking: The evolution from circuits to packets to content. *Computer Networks*, 66:112–120, 2014.
63. Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan, and Yuanyuan Zhou. C-Miner: mining block correlations in storage systems. In *Proceedings of the 3rd USENIX conference on File and storage technologies*, FAST'04, page 13, USA, March 2004.
64. Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable load balancing for Large-Scale storage systems with distributed caching. In *17th USENIX conference on file and storage technologies*, FAST'19, pages 143–157, Boston, MA, February 2019.
65. Louwrentius. Understanding storage performance - IOPS and latency. https://louwrentius.com/understanding-storage-performance-iops-and-latency.html. Accessed: 2024-01-10.
66. Dhruv Matani, Ketan Shah, and Anirban Mitra. An o (1) algorithm for implementing the lfu cache eviction scheme. *arXiv preprint arXiv:2110.11602*, 2021.
67. Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th symposium on operating systems principles*, SOSP'21, pages 243–262, New York, NY, USA, 2021.
68. Nimrod Megiddo and Dharmendra S Modha. ARC: A self-tuning, low overhead replacement cache. In *2nd USENIX conference on file and storage technologies*, FAST'03, 2003.

69. 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: 2024-01-10.
70. Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Record*, 22(2):297–306, June 1993.
71. Cheng Pan, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. pRedis: Penalty and Locality Aware Memory Allocation in Redis. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC'19, pages 193–205, Santa Cruz CA USA, November 2019.
72. David Plonka and Paul Barford. Context-aware clustering of DNS query traffic. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement conference - IMC '08*, page 217, Vouliagmeni, Greece, 2008.
73. Ioannis Psaras, Richard G Clegg, Raul Landa, Wei Koong Chai, and George Pavlou. 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*, pages 78–91. Springer, 2011.
74. Pytorch. Pytorch. https://pytorch.org/. Accessed: 2024-01-10.
75. Ziyue Qiu, Juncheng Yang, and Mor Harchol-Balter. Can increasing the hit ratio hurt cache throughput? *arXiv preprint arXiv:2404.16219*, 2024.
76. Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. FrozenHot Cache: Rethinking Cache Management for Modern Hardware. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys'23, pages 557–573, New York, NY, USA, May 2023.
77. RocksDB. Block cache. https://github.com/EighteenZi/rocksdb_wiki/blob/master/Block-Cache.md. Accessed: 2024-01-10.
78. Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning Cache Replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies*, FAST'21, pages 341–354, February 2021.
79. Elisha J. Rosensweig, Jim Kurose, and Don Towsley. Approximate models for general cache networks. In *2010 Proceedings IEEE INFOCOM*, pages 1–9, 2010.
80. Anirudh Sabnis and Ramesh K. Sitaraman. TRAGEN: a synthetic trace generator for realistic cache simulations. In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC'21, pages 366–379, Virtual Event, November 2021.
81. Anirudh Sabnis and Ramesh K. Sitaraman. JEDI: model-driven trace generation for cache simulations. In *Proceedings of the 22nd ACM Internet Measurement Conference*, IMC'22, pages 679–693, New York, NY, USA, October 2022.
82. 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: 2024-01-10.
83. Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: a cautionary tale. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation-Volume 3*, Volume3, pages 18–18, 2006.
84. Colin Scott. Latency numbers every programmer should know. https://colin-scott.github.io/personal_website/research/interactive_latency.html. Accessed: 2024-01-10.
85. Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch,

and Amin Vahdat. CliqueMap: productionizing an RMA-based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM'21, pages 93–105, Virtual Event USA, August 2021.
86. Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. EELRU: simple and effective adaptive page replacement. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):122–133, May 1999.
87. sobyte. Linux kernel page replacement algorithms. https://www.sobyte.net/post/2022-01/linux-multi-lru/. Accessed: 2024-01-10.
88. Solidigm. Solidigm p44 pro series – exceptional performanc. https://www.bhphotovideo.com/lit_files/1022190.pdf. Accessed: 2024-01-10.
89. Zhenyu Song, Daniel S Berger, Kai Li, Anees Shaikh, Wyatt Lloyd, Soudeh Ghorbani, Changhoon Kim, Aditya Akella, Arvind Krishnamurthy, Emmett Witchel, and others. Learning relaxed belady for content distribution network caching. In *17th USENIX symposium on networked systems design and implementation*, NSDI'20, pages 529–544, 2020.
90. Zhenyu Song, Kevin Chen, Nikhil Sarda, Deniz Altınbüken, Eugene Brevdo, Jimmy Coleman, Xiao Ju, Pawel Jurczyk, Richard Schooler, and Ramki Gummadi. HALP: Heuristic Aided Learned Preference Eviction Policy for YouTube Content Delivery Network. In *20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'23, pages 1149–1163, 2023.
91. Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K. Sitaraman. 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, pages 55–67, Incheon Republic of Korea, November 2017.
92. Saran Tarnoi, Vorapong Suppakitpaisarn, Wuttipong Kumwilaisak, and Yusheng Ji. Performance analysis of probabilistic caching scheme using markov chains. In *2015 IEEE 40th Conference on Local Computer Networks (LCN)*, pages 46–54, 2015.
93. Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ML-based LeCaR. In *10th USENIX workshop on hot topics in storage and file systems*, hotStorage'18, Boston, MA, July 2018.
94. Darryl L Willick, Derek L Eager, and Richard B Bunt. Disk cache replacement policies for network fileservers. In *[1993] Proceedings. The 13th International Conference on Distributed Computing Systems*, pages 2–11. IEEE, 1993.
95. Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 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, pages 307–323, February 2021.
96. Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. GL-Cache: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies*, FAST'23, pages 115–134, 2023.
97. Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and K. V. Rashmi. FIFO can be Better than LRU: the Power of Lazy Promotion and Quick Demotion. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS'23, pages 70–79, New York, NY, USA, June 2023.
98. Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX symposium on operating systems design and implementation*, OSDI'20, pages 191–208, November 2020.

99. Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'21, pages 503–518, April 2021.
100. Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 130–149, 2023.
101. Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. CacheSack: Admission Optimization for Google Datacenter Flash Caches. In *2022 USENIX Annual Technical Conference*, ATC'22, pages 1021–1036, Carlsbad, CA, July 2022.
102. Xiangqun Zhang, Shuyi Pei, Jongmoo Choi, and Bryan S. Kim. Excessive SSD-internal parallelism considered harmful. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '23, page 65–72, New York, NY, USA, 2023.
103. Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and K. V. Rashmi. 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.
104. Chen Zhong, Xingsheng Zhao, and Song Jiang. LIRS2: an improved LIRS replacement algorithm. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, SYSTOR'21, pages 1–12, Haifa Israel, June 2021.
105. Timothy Zhu, Anshul Gandhi, Mor Harchol-Balter, and Michael A Kozuch. Saving cash by using less cache. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*, 2012.