AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network

Daniel S. Berger, *University of Kaiserslautern;* Ramesh K. Sitaraman, *University of Massachusetts Amherst and Akamai Technologies;* Mor Harchol-Balter, *Carnegie Mellon University*

https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/berger

This paper is included in the Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17).

March 27–29, 2017 • Boston, MA, USA

ISBN 978-1-931971-37-9

AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network

Daniel S. Berger[1], Ramesh K. Sitaraman[2], and Mor Harchol-Balter[3]
[1]*University of Kaiserslautern* [2]*UMass Amherst & Akamai Technologies* [3]*Carnegie Mellon University*

Abstract

Most major content providers use content delivery networks (CDNs) to serve web and video content to their users. A CDN is a large distributed system of servers that caches and delivers content to users. The first-level cache in a CDN server is the memory-resident Hot Object Cache (HOC). A major goal of a CDN is to maximize the object hit ratio (OHR) of its HOCs. But, the small size of the HOC, the huge variance in the requested object sizes, and the diversity of request patterns make this goal challenging.

We propose AdaptSize, the first *adaptive*, *size-aware* cache admission policy for HOCs that achieves a high OHR, even when object size distributions and request characteristics vary significantly over time. At the core of AdaptSize is a novel Markov cache model that seamlessly adapts the caching parameters to the changing request patterns. Using request traces from one of the largest CDNs in the world, we show that our implementation of AdaptSize achieves significantly higher OHR than widely-used production systems: 30-48% and 47-91% higher OHR than Nginx and Varnish, respectively. AdaptSize also achieves 33-46% higher OHR than state-of-the-art research systems. Further, AdaptSize is more robust to changing request patterns than the traditional tuning approach of hill climbing and shadow queues studied in other contexts.

1 Introduction

Content delivery networks (CDNs) [18] enhance performance by caching objects in servers close to users and rapidly delivering those objects to users. A large CDN, such as that operated by Akamai [57], serves trillions of user requests a day from 170,000+ servers located in 1500+ networks in 100+ countries around the world. CDNs carry the majority of today's Internet traffic and are expected to carry almost two thirds within five years [22].

A CDN server employs two levels of caching: a small but fast in-memory cache called the Hot Object Cache (HOC) and a large second-level Disk Cache (DC). Each requested object is first looked up in the HOC. If absent, it is looked up in the DC. If also absent there, the object is fetched over the WAN from the content provider's origin.

Serving requests from the HOC is much faster and more efficient than serving from the DC. Thus, the goal of a CDN is to maximize the *object hit ratio* (OHR), which is the fraction of requests served from the HOC.

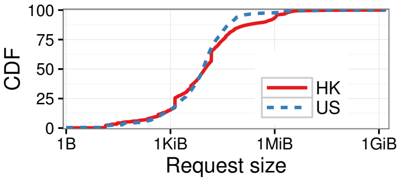


Figure 1: The cumulative distribution for object sizes in two Akamai production traces from Hong Kong and the US. Sizes vary by more than nine orders of magnitude.

In this paper, we show how to attain this goal in a robust, efficient and scalable manner (see Section 2).

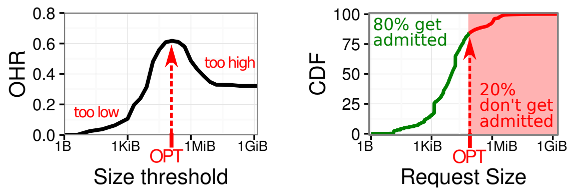
1.1 Why HOC cache management is hard

HOC cache management entails two types of decisions. First, the cache can decide whether or not to admit an object (cache admission). Second, the cache can decide which object to evict from the cache (cache eviction), if there is no space for a newly admitted object. While cache management is well studied in other contexts, HOC cache management poses the following new challenges.

1) The HOC is subject to extreme variability in request patterns and object sizes. CDNs serve multiple traffic classes using a *shared* server infrastructure. Such classes include web sites, videos, and interactive applications from thousands of content providers, each class with its own distinctive object size distributions and request patterns [57]. Figure 1 shows the object size distribution of requests served by two Akamai production servers (one in the US, the other in Hong Kong). We find that object sizes span more than nine orders of magnitude, and that the largest objects are often of the same order of magnitude as the HOC size itself. This extreme variability underscores the need for cache admission, as placing one large object in the HOC can result in the eviction of many small ones, which can severely degrade the OHR (Section 3).

2) Prior academic research is largely inapplicable as it focuses on caching objects of similar sizes. While the academic literature on caching policies is extensive, it focuses on situations where objects are of the same size. Further, prior work almost exclusively focuses on *eviction policies*, i.e., *all* requested objects are *admitted* to the cache and space is only managed by eviction (Section 7). Thus, there is little emphasis on cache admission in the prior literature, even as we show that cache admission is key in our context (Section 3).

3) Production systems implement a cache admission scheme with a static threshold that performs sub-



(a) OHR versus threshold. (b) Request size distribution.

Figure 2: Experimental results with different size thresholds. (a) A OHR-vs-threshold curve shows that the Object Hit Ratio (OHR) is highly sensitive to the size threshold, and that the optimal threshold (red arrow) can significantly improve the OHR. (b) The optimal threshold admits the requested object for only 80% of the requests.

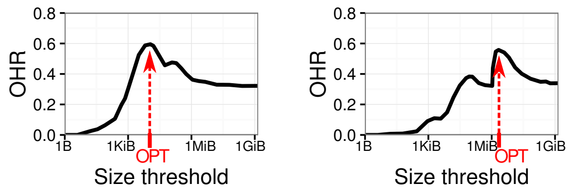
optimally for CDN workloads. In contrast to much of the academic research, production systems recognize the fact that *not all* objects can be admitted into the HOC. A common approach is to define a static size threshold and to only admit objects with size below this threshold. Figure 2a shows how OHR is affected by the size threshold for a production CDN workload. While the optimal threshold (OPT) almost doubles the OHR compared to admitting all objects, conservative thresholds that are too high lead to marginal gains, and the OHR quickly drops to zero for aggressive thresholds that are too low.

Unfortunately, the "best" threshold changes significantly over time. Figures 3a and 3b show the OHR as a function of the size threshold at two different times of the day. Note that the optimal thresholds can vary by as much as two orders of magnitude during a day. Since no prior method exists for dynamically tuning such a threshold, companies have resorted to either setting the size admission threshold conservatively high, or (more commonly) not using size-aware admission at all [67, 54, 21].

4) Simple strategies for dynamically tuning cache admission parameters do not work well. While it may seem that simple tuning approaches can be used to adapt the size threshold parameter over time, this turns out to be a non-trivial problem. This is probably why size-aware admission is not used effectively in practice. In Section 3, we consider common tuning approaches such as hill climbing with shadow caches, or using a threshold that is a fixed function of a request size percentile (e.g., the 80-th percentile as in Figure 2b). We also consider using probabilistic size-aware admission, where small sizes are "more likely" to be admitted, and large files are "less likely" to be admitted. We find that none of these approaches is sufficiently robust to traffic mix changes that occur in daily operation due in part to the CDN's global load balancer.

1.2 Our contributions

We propose AdaptSize, a lightweight and near-optimal tuning method for size-aware cache admission. AdaptSize is based on a novel statistical representation of the cache using a Markov model. This model is unique in that it incorporates all correlations between object sizes and current request rates (prior caching models assumed unit-sized objects). This model is analytically tractable, allowing us to quickly find the optimal size-aware admission policy and repeat this process at short intervals.



(a) Morning: web traffic. (b) Evening: web/video mix.

Figure 3: The optimal size threshold changes significantly over time. (a) In the morning hours, small objects (e.g., news items) are more popular, which requires a small size threshold of a few tens of KiBs. (b) In the evening hours, web traffic gets mixed with video traffic, which requires a size threshold of a few MiBs.

We have implemented AdaptSize within the Varnish production caching system[1] Varnish is known as a high-throughput system that makes extensive use of concurrency [76, 43, 42] and is used by several prominent content providers and CDNs, including Wikipedia, Facebook, and Twitter. Through evaluations of AdaptSize on production request traces from one of the world's largest CDNs, Akamai, we observe the following key features.

1. AdaptSize improves the OHR by 47-91% over an unmodified Varnish system, and by 30-48% over an offline-tuned version of the Nginx caching system (Figure 4 and Section 6.1). Varnish and Nginx are used by almost 80% of the top 5000 websites, which we determined by crawling Alexa's top sites list [74].
2. In addition to improving upon production systems, AdaptSize also improves the OHR by 33-46% over state-of-the-art research caching systems (Figure 5 and Section 6.2).
3. Compared to other tuning methods, such as the classical hill climbing technique using shadow queues, AdaptSize improves the OHR on average by 15-20% and in some cases by more than 100% (Figure 6 and Section 6.3). In particular, we found classical tuning methods can get "stuck" at poor local optima that are avoided by AdaptSize's model-based optimization.
4. We compare AdaptSize with SIZE-OPT, which tunes the size threshold parameter using a priori knowledge of the next one million requests. AdaptSize stays within 90% of the OHR of SIZE-OPT in the median across all experiments and is never worse than 80% of the OHR of SIZE-OPT (Sections 6.1

[1] The source code of AdaptSize and installation instructions are available at `https://github.com/dasebe/AdaptSize` .

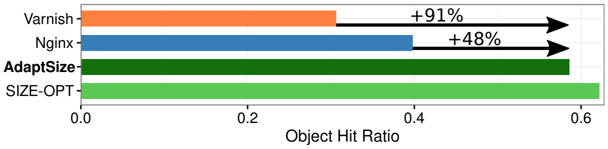


Figure 4: Comparison of AdaptSize's implementation to the Varnish and Nginx production systems. We also show the SIZE-OPT policy which has future knowledge and uses this to set the optimal size threshold at every moment in time. AdaptSize improves the OHR by 48-91% over the production systems and also achieves 95% of the OHR of SIZE-OPT. These results are for the US trace and a typical HOC size (details in Section 5.1).

and 6.3) – even when subjected to extreme changes in the request traffic.

5. In addition to improving the OHR, AdaptSize also reduces request latencies by 43% in the median, and by more than 30% at the 90-th and 99.9-th percentile. AdaptSize is able to maintain the high throughput of Varnish without adding (concurrent) synchronization overheads, and reduces the disk utilization of the second-level cache by 20% (Section 6.4).

Roadmap. The rest of this paper is structured as follows. Sections 2 and 3 discuss our goals and our rationale in designing AdaptSize. Section 4 details AdaptSize's design and implementation. Sections 5 and 6 describe our setup and experimental results, respectively. Section 7 reviews related work. We conclude in Section 8.

2 HOC Design Goals

In designing AdaptSize, we seek to maximize the OHR, while maintaining a robust and scalable HOC and avoiding adverse side-effects on second-level caches.

Maximizing the OHR. The HOC's primary design objective is user performance, which it optimizes by providing fast responses for as many requests as possible. A natural way to measure this objective is the *object hit ratio* (OHR), which gives equal weight to all user requests.

While our evaluations are based on production CDN servers without SSDs, HOCs are also key to hybrid CDN servers that typically have both hard disks and SSDs. This is because HOCs are more CPU efficient and can serve traffic at higher rates. Further, HOCs offload requests from the SSDs that are often i/o bound. HOCs are used in SSD-based servers at Akamai, and also at Fastly [54] and Wikipedia [67]. These production deployments seek to maximize the OHR of the HOC, which is the main performance metric used throughout this paper.

There are other cache performance metrics that are less relevant to the HOC. For example, the much larger DC focuses on the byte hit rate (BHR) that is the fraction of *bytes* that are served from the cache [71]. The HOC has little impact on the BHR as it is typically three orders of magnitude smaller than the DC.

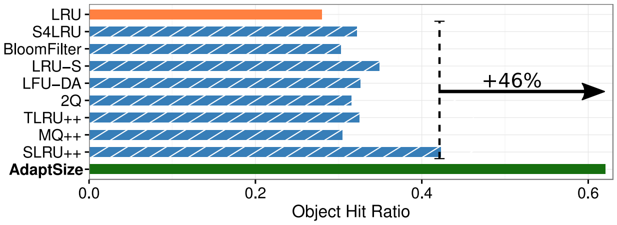


Figure 5: Comparison of AdaptSize to state-of-the-art research caching systems. Most of these use sophisticated admission and eviction policies that combine recency and frequency (striped blue bars). AdaptSize improves the OHR by 46% over the next best system. Policies annotated by "++" are actually optimistic, because we offline-tuned their parameters to the trace. These results are for the US trace and a HOC size 1.2 GiB.

Robustness against changing request patterns. A HOC is subjected to a variety of traffic changes each day. For example, web content popularity changes during the day (e.g., news in the morning vs. video at night), which includes rapid changes due to flash crowds. Another source of traffic changes is the sharing of the server infrastructure between traffic classes. Such classes include web sites, videos, software downloads, and interactive applications from thousands of content providers [57]. As a shared infrastructure is more cost effective, a CDN server typically serves a mix of traffic classes. Due to load balancing decisions, this mix can change abruptly. This poses a particular challenge as each traffic class has its own distinctive request and object size distribution statistics: large objects can be unpopular during one hour and popular during the next. A HOC admission policy must be able to rapidly adapt to all these changing request patterns in order to achieve consistently high OHRs.

Low overhead and high concurrency. As the first caching level in a CDN, the HOC needs to both respond quickly to requests and deliver high throughput. This requires that the admission and eviction policies have a small processing overhead, i.e., a constant time complexity per request (see the $O(1)$ policies in Table 2 in Section 7), and that they have concurrent implementations (see the corresponding column in Table 2).

No negative side-effects. While the HOC achieves a high OHR and fast responses, it must not impede the performance of the overall CDN server. Specifically, changes to the HOC must not negatively affect the BHR and disk utilization of the DC.

3 Rationale for AdaptSize

The goal of this section is to answer why the HOC needs size-aware admission, why such an admission policy

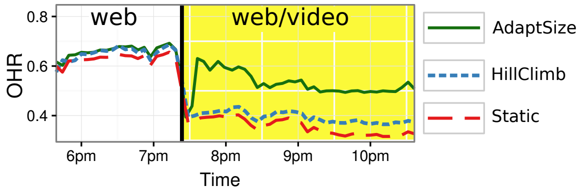


Figure 6: Comparison of AdaptSize, threshold tuning via hill climbing and shadow caches (HillClimb), and a static size threshold (Static) under a traffic mix change from only web to mixed web/video traffic. While AdaptSize quickly adapts to the new traffic mix, HillClimb gets stuck in a suboptimal configuration, and Static (by definition) does not adapt. AdaptSize improves the OHR by 20% over HillClimb and by 25% over Static on this trace.

needs to be adaptively tuned, and why a new approach to parameter tuning is needed.

3.1 Why HOCs need size-aware admission

We start with a toy example. Imagine that there are only two types of objects: 9999 small objects of size 100 KiB (say, web pages) and 1 large object of size 500 MiB (say, a software download). Further, assume that all objects are equally popular and requested forever in round-robin order. Suppose that our HOC has a capacity of 1 GiB.

A HOC that does not use admission control cannot achieve an OHR above 0.5. Every time the large object is requested, it pushes out ≈5000 small objects. It does not matter which objects are evicted: when the evicted objects are requested, they cannot contribute to the OHR.

An obvious solution for this toy example is to control admissions via a size threshold. If the HOC admits only objects with a size at most 100 KiB, then it can achieve an OHR of 0.9999 as all small objects stay in the cache.

This toy example is illustrative of what happens under real production traffic. We observe from Figure 1 that approximately 5% of objects have a size bigger than 1 MiB. Every time a cache admits a 1 MiB object, it needs to evict space equivalent to one thousand 1 KiB objects, which make up about 15% of requests. Again, those evicted objects will not be able to contribute to the OHR. A well-designed cache admission algorithm could help avoid such evictions that have a large impact on OHR.

3.2 Why we need a new tuning method

The key question when implementing size-aware admission is picking its parameters. Figure 2 shows that a static size threshold is inadequate. Next, we explore three canonical approaches for tuning a size threshold. These approaches are well-known in prior literature and have been applied in other contexts (unrelated to the tuning of size thresholds). However, we show that these known approaches are deficient in our context, motivating the need for AdaptSize's new tuning mechanism.

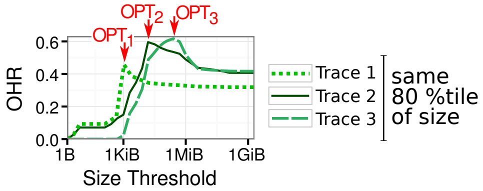


Figure 7: Experimental results showing that setting the size threshold to a fixed function does not work. All three traces shown here have the same 80-th size percentile, but their optimal thresholds differ by two orders of magnitude.

Tuning based on request size percentiles. A common approach used in many contexts (e.g., capacity provisioning) is to derive the required parameter as some function of the request size distribution and arrival rate. A simple way of using this approach in our context is to set the size threshold for cache admission to be a fixed percentile of the object size distribution. However, for production CDN traces, there is no fixed relationship between the percentiles of the object size distribution and optimal size threshold that maximizes the OHR. In Figure 2, the optimal size threshold lands on the 80-th percentile request size. However, in Figure 7, note that all three traces have the same 80-th percentile but very different optimal thresholds. In fact, we found many examples of multiple traces that agree on *all* size percentiles and yet have *different* optimal size thresholds. The reason is that for maximizing OHR it matters whether the number of requests seen for a specific object size come from one (very popular) object or from many (unpopular) objects. This information is not captured by the request size distribution.

Tuning via hill climbing and shadow caches. A common tool for the tuning of caching parameters is the use of shadow caches. For example, in the seminal paper on ARC [53], the authors tune their eviction policy to have the optimal balance between recency and frequency by using a shadow cache (we discuss other related work using shadow caches in Section 7.2). A shadow cache is a simulation which is run in real time simultaneously with the main (implemented) cache, but using a different parameter value than the main cache. Hill climbing then adapts the parameter by comparing the hit ratio achieved by the shadow cache to that of the main cache (or another shadow cache). In theory, we could exploit the same idea to set our size-aware admission threshold. Unfortunately, when we tried this, we found that the OHR-vs-threshold curves are not concave and that they can have several local optima, in which the hill climbing gets frequently stuck. Figure 3b shows such an example, in which the local optima result from mixed traffic (web and video). As a consequence, we will demonstrate experimentally in Section 6.3 that hill climbing is suboptimal. AdaptSize achieves an OHR that is 29% higher than hill climbing on average and 75% higher in some cases. We tried adding more shadow caches, and also randomizing the evaluated parameters, but could not find a robust variant that consistently optimized the OHR across multiple traces[2].

In conclusion, our extensive experiments show that tuning methods like shadow caches with hill climbing are simply not robust enough for the problem of size-aware admission with CDN traffic.

Avoiding tuning by using probabilistic admission. One might imagine that the difficulty in tuning the size threshold lies in the fact that we are limited to a single strict threshold. The vast literature on randomized algorithm suggests that probabilistic parameters are more robust than deterministic ones [55]. We attempted to apply this idea to size-aware tuning by considering probabilistic admission policies, which "favor the smalls" by admitting them with high probability, whereas large objects are admitted with low probability. We chose a probabilistic function that is exponentially decreasing in the object size ($e^{-size/c}$). Unfortunately, the parameterization of the exponential curve (the c) matters a lot – and it's just as hard to find this c parameter as it is to find the optimal size threshold. Furthermore, the best exponential curve (the best c) changes over time. In addition to exponentially decreasing probabilities, we also tried inversely proportional, linear, and log-linear variants. Unfortunately, none of these variants resolves the problem that there is at least one parameter without an obvious way how to choose it.

In conclusion, even randomized admission control requires the tuning of some parameter.

4 The AdaptSize Caching System

AdaptSize admits objects with probability $e^{-size/c}$ and evicts objects using a concurrent variant of LRU [43]. Observe that the function $e^{-size/c}$ is biased in favor of admitting small sizes with higher *probability*.

Why a probabilistic admission function? The simplest size-based admission policy is a deterministic threshold c where only objects with a size $< c$ are admitted. A probabilistic admission function, like $e^{-size/c}$, is more flexible: objects greater than c retain a low but non-zero admission probability, which results in eventual admission for popular objects (but not for unpopular ones). In our experiments $e^{-size/c}$ consistently achieves a 10% higher OHR than the best deterministic threshold.

What parameter c does AdaptSize use in the $e^{-size/c}$ function? AdaptSize's tuning policy recomputes the optimal c every Δ requests. A natural approach is to use hill-climbing with shadow caches to determine the optimal c parameter. Unfortunately, that leads to a myopic view in that only a *local* neighborhood of the current c can be searched. This leads to sub-optimal results, given the non-convexities present in the OHR-vs-c curve (Figure 3). By contrast, we derive a full Markov chain model of the cache. This model allows AdaptSize to view the entire OHR-vs-c curve and perform a *global* search for the optimal c. The challenge of the Markov model approach is in devising an algorithm for finding the solution quickly and in incorporating that algorithm into a production system.

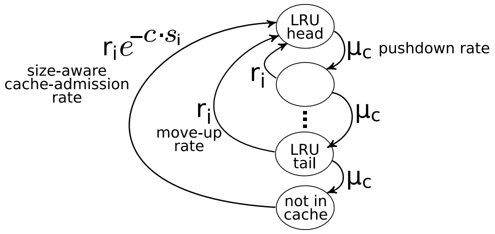


Figure 8: AdaptSize's Markov chain model for object i represents i's position in the LRU list and the possibility that the object is out of the cache. Each object is represented by a separate Markov chain, but all Markov chains are connected by the common "pushdown" rate μ_c. Solving these models yields the OHR as a function of c.

In the following, we describe the derivation of AdaptSize's Markov model (Section 4.1), and how we incorporate AdaptSize into a production system (Section 4.2).

4.1 AdaptSize's Markov chain tuning model

To find the optimal c, AdaptSize uses a novel Markov chain model, which is very different from that typically used for cache modeling. Traditionally, people have modeled the entire state of the cache, tracking all objects in the cache and their ordering in the LRU list [44, 30, 15, 52, 10, 33, 25, 24, 19, 68]. While this is 100% accurate, it also becomes completely infeasible when the number of objects is high, because of a combinatorical state space explosion.

AdaptSize instead creates a *separate* Markov chain for each object (cf. Figure 8). Each object's chain tracks its position in the LRU list (if the object is in the cache), as well as a state for the possibility that the object is out of the cache. Using an individual Markov chain greatly reduces the model complexity, which now scales *linearly* with the number of objects, rather than *exponentially* in the number of objects.

AdaptSize's Markov chain. Figure 8 shows the Markov chain for the i^{th} object. The chain has two important parameters. The first is the rate at which object i is moved *up* to the head of the LRU list, due to accesses to the object. We get the "move up" rate, r_i, by collecting aggregate statistics for object i during the previous Δ time interval. The second parameter is the average rate at which object i is pushed *down* the LRU list. The "pushdown" rate, μ_c, depends on the rate with which any object is moved to the top of the LRU list (due to a hit,

[2] While there are many complicated variants of shadow-cache search algorithms, they all rely on a fundamental assumption of stationarity, which does not need to apply to web traffic.

or after cache admission). As it does not matter which object is moved to the top, μ_c is approximately the same for all objects. So, we consider a single "pushdown" rate for all objects. We calculate μ_c by solving an equation that takes all objects into account, and thus captures the interactions between all the objects[3]. Specifically, we find μ_c by solving an equation that says that the expected size of all cached objects can't exceed the capacity K that is actually available to the cache:

$$\sum_{i=1}^{N} \mathbb{P}[\text{object } i \text{ in cache}]\; s_i = K\ . \tag{1}$$

Here, N is the number of all objects observed over the previous Δ interval, and s_i is the size of object i. Note that $\mathbb{P}[\text{object } i \text{ in cache}]$ is a monotonic function in terms of μ_c, which leads to a unique solution.

Our new model enables us to find $\mathbb{P}[\text{object } i \text{ in cache}]$ as a function of c by solving for the limiting probabilities of all "in" states in Figure 8. Appendix A shows how this is done. We obtain a function of c in closed form.

Theorem 1 *(proof in Appendix A)*

$$\mathbb{P}[\mathit{object\ i\ in\ cache}] = \frac{(e^{r_i/\mu_c} - 1)\cdot e^{-c\cdot s_i}}{1 + (e^{r_i/\mu_c} - 1)\cdot e^{-c\cdot s_i}}$$

Note that the size admission parameter c affects both the admission probability ($e^{-s_i/c}$) and the pushdown rate (μ_c). For example, a lower c results in fewer admissions, which results in fewer evictions, and in a smaller pushdown rate.

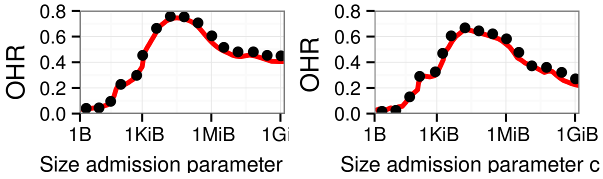
The OHR as a function of c. Theorem 1 and Equation (1) yield the OHR by observing that the expected number of hits of object i equals r_i (i's average request rate) times the long-term probability that i is in the cache. The OHR predicted for the threshold parameter c is then simply the ratio of expected hits to requests:

$$OHR(c) = \frac{\sum_{i=1}^{N} r_i\, \mathbb{P}[\text{object } i \text{ in cache}]}{\sum_{i=1}^{N} r_i}\ .$$

If we consider a discretized range of c values, we can now compute the OHR for each c in the range which gives us a "curve" of OHR-vs-c (similar to the curves in Figure 9).

Global search for the optimal c. Every Δ steps, we derive the OHR-vs-c curve using our Markov model. We search this curve for the c that maximizes the OHR using a standard global search method for non-concave functions [64]. This c is then used for the next Δ steps.

Accuracy of AdaptSize's model. Our Markov chain relies on several simplifying assumptions that can potentially impact the accuracy of the OHR predictions. Figure 9 shows that AdaptSize's OHR equation matches experimental results across the whole range of the threshold parameter c on two typical traces of length Δ. In addition, we continuously compared AdaptSize's model to measurements during our experiments (Section 6). AdaptSize is very accurate with an average error of about 1%.



(a) HK trace (b) US trace

Figure 9: AdaptSize's Markov model predicts the OHR sensitivity curve (red solid line). This is very accurate when compared to the actual OHR (black dots) that results when that threshold is chosen. Each experiment involves a portion of the production trace of length $\Delta = 250K$.

4.2 Integration with a production system

We implemented AdaptSize on top of Varnish [76, 32], a production caching system, by modifying the miss request path. On a cache miss, Varnish accesses the second-level cache to retrieve the object, and places it in its HOC. With AdaptSize, the probabilistic admission decision is executed, which is evaluated independently for all cache threads and adds a constant number of instructions to the request path. If the object is not admitted, it is served from Varnish's transient memory.

Our implementation uses a parameter Δ which is the size of the window of requests over which our Markov model for tuning is computed. In addition to statistics from the current window, we also incorporate the statistical history from prior windows via exponential smoothing, which makes AdaptSize more robust and largely insensitive to Δ on both of our production traces. In our experiments, we choose Δ=250K requests (about 5-10 mins on average), which allows AdaptSize to react quickly to changes in the request traffic.

Lock-free statistics collection. A key problem in implementing AdaptSize lies in efficient statistics collection for the tuning model. Gathering request statistics can add significant overhead to concurrent caching designs [69]. Varnish and AdaptSize use thousands of threads in our experiments, so centralized request counters would cause high lock contention. In fact, we find that Varnish's throughput bottleneck is lock contention for the few remaining synchronization points (e.g., [43]).

Instead of a central request counter, AdaptSize hooks into the internal data structure of the cache threads. Each cache thread keeps debugging information in a concurrent ring buffer, to which all events are simply appended (overwriting old events after some time). AdaptSize's statistics collection frequently scans this ring buffer (read only) and does not require any synchronization.

Robust and efficient model evaluation. The OHR prediction in our statistical model involves two more implementation challenges. The first challenge lies in efficiently solving equation (1). We achieve a constant time

[3] Mean-field theory [45] provides analytical justification for why it is reasonable to assume a single average pushdown rate, when there are thousands of objects (as in our case).

	HK trace	US trace
Total Requests	450 million	440 million
Total Bytes	157.5 TiB	152.3 TiB
Unique Objects	25 million	55 million
Unique Bytes	14.7 TiB	8.9 TiB
Start Date	Jan 29, 2015	Jul 15, 2015
End Date	Feb 06, 2015	Jul 20, 2015

Table 1: Basic information about our web traces.

overhead by using a fixed-point solver [26]. The second challenge is due to the exponential function in the Theorem 1. The value of the exponential function outgrows even 128-bit float number representations. We solve this problem by using an accurate and efficient approximation for the exponential function using a Padé approximant [63] that only uses simple float operations which are compatible with SSE/AVX vectorization, speeding up the model evaluation by about 10-50× in our experiments.

5 Evaluation Methodology

We evaluate AdaptSize using *both* trace-based simulations (Section 5.2) and a Varnish-based implementation (Section 5.3) running on our experimental testbed. For both these approaches, the request load is derived from traces from Akamai's production CDN servers (Section 5.1).

5.1 Production CDN request traces

We collected request traces from two production CDN servers in Akamai's global network. Table 1 summarizes the main characteristics of the two traces. Our first trace is from urban Hong Kong (**HK trace**). Our second trace is from rural Tennessee, in the US, (**US trace**). Both span multiple consecutive days, with over 440 million requests per trace during the months of February and July 2015. Both production servers use a HOC of size 1.2 GiB and several hard disks as second-level caches. They serve a traffic mix of several thousand popular web sites, which represents a typical cross section of the web (news, social networks, downloads, ecommerce, etc.) with highly variable object sizes. Some content providers split very large objects (e.g., videos) into smaller (e.g., 2 MiB) chunks. The chunking approach is accurately represented in our request traces. For example, the cumulative distribution function shown in Figure 1 shows a noticeable jump around the popular 2 MiB chunk size.

5.2 Trace-based simulator

We implemented a cache simulator in C++ that incorporates AdaptSize and several state-of-the-art research caching policies. The simulator is a single-threaded implementation of the admission and eviction policies and performs the appropriate cache actions when it is fed the CDN request traces. Objects are only stored via their ids and the HOC size is enforced by a simple check on the sum of bytes currently stored. While actual caching systems (such as Varnish [43, 42]) use multi-threaded concurrent implementations, our simulator provides a good approximation of the OHR when compared with our prototype implementations that we describe next.

5.3 Prototype Evaluation Testbed

Our implementation testbed is a dedicated (university) data center consisting of a client server, an origin server, and a CDN server that incorporates the HOC. We use FUJITSU CX250 HPC servers, which run RHEL 6.5, kernel 2.6.32 and gcc 4.4.7 on two Intel E5-2670 CPUs with 32 GiB RAM and an IB QDR networking interface.

In our evaluation, the HOC on our CDN server is either running Nginx, Varnish, or AdaptSize. Recall that we implemented AdaptSize by adding it to Varnish[4] as described in Section 4.2. We use Nginx 1.9.12 (February 2016) with its build-in frequency-based admission policy. This policy relies on one parameter: how many requests need to be seen for an object before being admitted to the cache. We use an optimized version of Nginx, since we have tuned its parameter offline for both traces. We use Varnish 4.1.2 (March 2016) with its default configuration that does not use an admission policy.

The experiments in Section 6.1, 6.2, and 6.3 focus on the HOC and do not use a DC. The DC in Section 6.1 uses Varnish in a configuration similar to that of the Wikimedia Foundation's CDN [67]. We use four equal dedicated 1 TB WD-RE3 7200 RPM 32 MiB-Cache hard disks attached via a Dell 6 Gb/s SAS Host Bus Adapter Card in raw mode (RAID disabled).

The client fetches content specified in the request trace from the CDN server using libcurl. The request trace is continuously read into a global queue, which is distributed to worker threads (client threads). Each client thread continually requests objects in a closed-loop fashion. We use up to 200 such threads and verified that the number of client threads has a negligible impact on the OHR.

If the CDN server does not have the requested content, it is fetched from the origin server. Our origin server is implemented in FastCGI. As it is infeasible to store all trace objects (23 TB total) on the origin server, our implementation creates objects with the correct size on the fly before sending them over the network. In order to stress test our caching implementation, the origin server is highly multi-threaded and intentionally never the bottleneck.

6 Empirical Evaluation

This section presents our empirical evaluation of AdaptSize. We divide our evaluation into three parts. In Section 6.1, we compare AdaptSize with production caching systems, as well as with an *offline* caching system called SIZE-OPT that continuously optimizes OHR with knowledge of future requests. While SIZE-OPT is not implementable in practice. it provides an upper bound on the

[4] We refer to AdaptSize incorporated into Varnish as "AdaptSize" and Varnish without modifications as "Varnish".

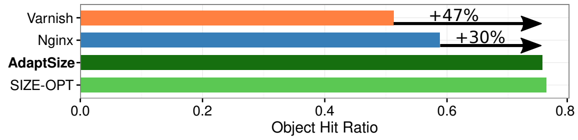


Figure 10: Comparison of AdaptSize's implementation to the Varnish and Nginx production systems and SIZE-OPT. AdaptSize improves the OHR by 30-47% over the production systems and also achieves 99% of the OHR of SIZE-OPT. These results are for the HK trace; corresponding results for the US trace are shown in Figure 4.

achievable OHR to which AdaptSize can be compared. In Section 6.2, we compare AdaptSize with research caching systems that use more elaborate eviction and admission policies. In Section 6.3, we evaluate the robustness of AdaptSize by emulating both randomized and adversarial traffic mix changes. In Section 6.4, we evaluate the side-effects of AdaptSize on the overall CDN server.

6.1 Comparison with production systems

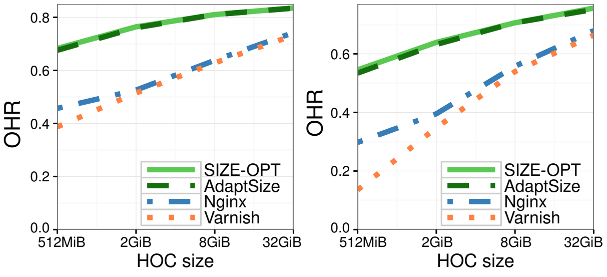
We use our experimental testbed outlined in Section 5.3 and answer four basic questions about AdaptSize.

What is AdaptSize's OHR improvement over production systems? Quick answer: *AdaptSize improves the OHR by 47-91% over Varnish and by 30-48% over Nginx*. We compare the OHR of AdaptSize to Nginx and Varnish using the 1.2 GiB HOC configuration from the corresponding Akamai production servers (Section 5.1). For the HK trace (Figure 10), we find that AdaptSize improves over Nginx by 30% and over Varnish by 47%. For the US trace (Figure 4), the improvement increases to 48% over Nginx and 91% over Varnish.

The difference in the improvement over the two traces stems from the fact that the US trace contains 55 million unique objects as compared to only 25 million unique objects in the HK trace. We further find that AdaptSize improves the OHR variability (the coefficient of variation) by 1.9× on the HK trace and by 3.8× on the US trace (compared to Nginx and Varnish).

How does AdaptSize compare with SIZE-OPT? Quick answer: *for the typical HOC size, AdaptSize achieves an OHR within 95% of SIZE-OPT*. We benchmark AdaptSize against the SIZE-OPT policy, which tunes the threshold parameter c using a priori knowledge of the next one million requests. Figures 4 and 10 show that AdaptSize is within 95% of SIZE-OPT on the US trace, and within 99% of SIZE-OPT on the HK trace, respectively.

How much is AdaptSize's performance affected by the HOC size? Quick answer: *AdaptSize's improvement over production caching systems becomes greater for smaller HOC sizes, and decreases for larger HOC sizes*. We consider the OHR when scaling the HOC size between 512 MiB and 32 GiB under the production server traffic of a 1.2 GiB HOC. Figures 11a and 11b shows that the performance of AdaptSize is close to SIZE-OPT for all HOC sizes. The improvement of AdaptSize over Nginx and Varnish is most pronounced for HOC sizes close to the original configuration. As the HOC size increases, the OHR of all caching systems improves, since the HOC can store more objects. This leads to a smaller relative improvement of AdaptSize for a HOC size of 32 GiB: 10-12% over Nginx and 13-14% over Varnish.



(a) HK trace

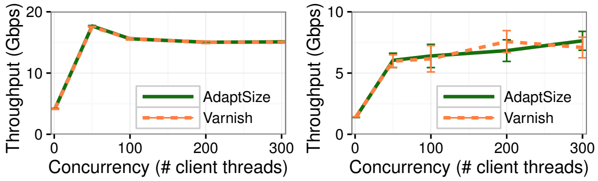
(b) US trace

Figure 11: Comparison of AdaptSize to SIZE-OPT, Varnish, and Nginx when scaling the HOC size under the production server traffic of two 1.2 GiB HOCs. AdaptSize always stays close to SIZE-OPT and significantly improves the OHR for all HOC sizes.

How much is AdaptSize's performance affected when jointly scaling up HOC size and traffic rate? Quick answer: *AdaptSize's improvement over production caching systems remains constant for larger HOC sizes*. We consider the OHR when jointly scaling the HOC size and the traffic rate by up 128x (153 GiB HOC size). This is done by splitting a production trace into 128 non-overlapping segments and replaying all 128 segments concurrently. We find that the OHR remains approximately constant as we scale up the system, and that AdaptSize achieves similar OHR improvements as under the original 1.2 GiB HOC configuration.

What about AdaptSize's overhead? Quick answer: *AdaptSize's throughput is comparable to existing production systems and AdaptSize's memory overhead is reasonably small*. AdaptSize is build on top of Varnish, which focuses on high concurrency and simplicity. In Figure 12, we compare the throughput (bytes per second of satisfied requests) of AdaptSize to an unmodified Varnish system. We use two micro experiments. The first benchmarks the hit request path (100% OHR scenario), to verify that there is indeed no overhead for cache hits (see section 4.2). The second benchmarks the miss request path (0% OHR scenario), to assess the worst-case overhead due to the admission decision.

We replay one million requests and configure different concurrency levels via the number of client threads. Note that a client thread does not represent an individual user (Section 5.3). The results are based on 50 repetitions.



(a) 100% OHR scenario.

(b) 0% OHR scenario.

Figure 12: Comparison of the throughput of AdaptSize and Varnish in micro experiments with (a) 100% OHR and (b) 0% OHR. Scenario (a) stress tests the hit request path and shows that there is no difference between AdaptSize and Varnish. Scenario (b) stress tests the miss request path (every request requires an admission decision) and shows that the throughput of AdaptSize and Varnish is very close (within confidence intervals).

Figure 12a shows that the application throughput of AdaptSize and Varnish are indistinguishable in the 100% OHR scenario. Both systems achieve a peak throughput of 17.5 Gb/s for 50 clients threads. Due to lock contention, the throughput of both systems decreases to around 15 Gb/s for 100-300 clients threads. Figure 12b shows that the application throughput of both systems in the 0% OHR scenario is very close, and always within the 95% confidence interval.

The memory overhead of AdaptSize is small. The memory overhead comes from the request statistics needed for AdaptSize's tuning model. Each entry in this list describes one object (size, request count, hash), which requires less than 40 bytes. The maximum length of this list, across all experiments, is 1.5 million objects (58 MiB), which also agrees with the memory high water mark (VmHWM) reported by the Kernel for AdaptSize's tuning process.

6.2 Comparison with research systems

We have seen that AdaptSize performs very well against production systems. We now ask the following.

How does AdaptSize compare with research caching systems, which involve more sophisticated admission and eviction policies? Quick answer: *AdaptSize improves by 33-46% over state-of-the-art research caching system.* We use the simulation evaluation setup explained in Section 5.2 with eight systems from Table 2, which are selected with the criteria of having an efficient constant-time implementation. Four of the eight systems use a recency and frequency trade-off with fixed weights between recency and frequency. Another three systems (ending with "++") use sophisticated recency and frequency trade-offs with variable weights, which we hand-tuned to our traces to create optimistic variants[5]. The remaining system is LRU-S [72], which uses size-aware eviction and admission with static parameters.

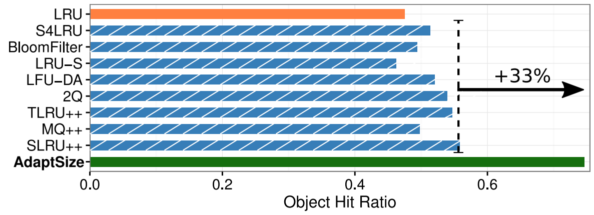


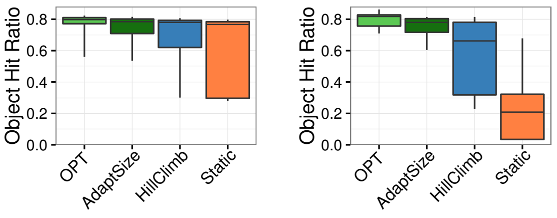
Figure 13: Comparison of AdaptSize to state-of-the-art research caching systems. Most of these are sophisticated admission and eviction policies that combine recency and frequency (striped blue bars). LRU-S is the only system – besides AdaptSize – that incorporates size. AdaptSize improves the OHR by 33% over the next best system. Policies annotated by "++" are actually optimistic, because we offline-tuned their parameters to the trace. These results are for the HK trace; correspondings results for the US trace are shown in Figure 5.

Figure 13 shows the simulation results for a HOC of size 1.2 GiB on the HK trace. We find that AdaptSize achieves a 33% higher OHR than the second best system, which is SLRU++. Figure 5 shows the simulation results for the US trace. AdaptSize achieves a 46% higher OHR than the second best system, which is again SLRU++. Note that SLRU's performance heavily relies on offline parameters as can be seen by the much smaller OHR of S4LRU, which is a static-parameter variant of SLRU. In contrast, AdaptSize achieves its superior performance without needing offline parameter optimization. In conclusion, we find that AdaptSize's policies outperform sophisticated eviction and admission policies, which do not depend on the object size.

6.3 Robustness of alternative tuning methods for size-aware admission

So far we have seen that AdaptSize significantly improves the OHR over caching systems without size-aware admission, including production caching systems (Section 6.1) and research caching systems (Section 6.2). We now focus on different cache tuning methods for the size-aware admission parameter c (see the beginning of Section 4). Specifically, we compare AdaptSize with hill climbing (**HillClimb**), based on shadow caches (cf. Section 3). HillClimb uses two shadow caches and we hand-optimized its parameters (interval of climbing steps, step size) on our production traces. We also compare to a static size threshold (**Static**), where the value of this static threshold is offline optimized on our production traces. We also compare to SIZE-OPT, which tunes c based on offline knowledge of the next one million requests. All four policies are implemented on Varnish using the setup explained in Section 5.3.

[5] There are self-tuning variants of recency-frequency trade-offs such as ARC [53]. Unfortunately, we could not test ARC itself, because its learning rule relies on the assumption of unit-sized object sizes.



(a) Randomized.

(b) Adversarial.

Figure 14: Comparison of cache tuning methods under traffic mix changes. We performed 50 randomized traffic mix changes (a), and 25 adversarial traffic mix changes (b). The boxes show the range of OHR from the 25-th to the 75-th percentile among the 25-50 experiments. The whiskers show the 5-th to the 95-th percentile.

We consider two scenarios: 1) *randomized traffic mix changes* and 2) *adversarial traffic mix changes*. A randomized traffic mix change involves a random selection of objects which abruptly become very popular (similar to a flash crowd event). An adversarial traffic mix change involves frequently changing the traffic mix between classes that require vastly different size-aware admission parameters (e.g., web, video, or download traffic). An example of an adversarial change is the case where objects larger than the previously-optimal threshold suddenly become very popular.

Is AdaptSize robust against randomized traffic mix changes? Quick answer: *AdaptSize performs within 95% of SIZE-OPT's OHR even for the worst 5% of experiments, whereas HillClimb and Static achieve only 47-54% of SIZE-OPT's OHR.* We create 50 different randomized traffic mix changes. Each experiment consists of two parts. The first part is five million requests long, and allows each tuning method to converge to a stable configuration. The second part is ten million requests long, and consists of 50% production-trace requests and 50% of very popular objects. The very popular objects consist of a random number of objects (between 200 and 1000), which are randomly sampled from the trace.

Figure 14a shows a boxplot of the OHR for each caching tuning method across the 50 experiments. The boxes indicate the 25-th and 75-th percentile, the whiskers indicate the 5-th and 95-th percentile. AdaptSize improves the OHR over HillClimb across every percentile, by 9% on average, and by more than 75% in five of the 50 experiments. AdaptSize improves the OHR over Static across every percentile, by 30% on average, and by more than 100% in five of the 50 experiments. Compared to SIZE-OPT, AdaptSize achieves 95% of the OHR for all percentiles.

Is AdaptSize robust against adversarial traffic mix changes? Quick answer: *AdaptSize performs within 81% of SIZE-OPT's OHR even for the worst 5% of experiments, whereas HillClimb and Static achieve only 5-15% of SIZE-OPT's OHR.* Our experiment consists of 25 traffic mix changes. Each traffic mix is three million requests long, and the optimal *c* parameter changes from 32-256 KiB to 1-2 MiB, then to 16-32 MiB, and back again.

Figure 14b shows a boxplot of the OHR for each caching tuning method across all 50 experiments. The boxes indicate the 25-th and 75-th percentile, the whiskers indicate the 5-th and 95-th percentile. AdaptSize improves the OHR over HillClimb across every percentile, by 29% on average, and by more than 75% in seven of the 25 experiments. AdaptSize improves the OHR over Static across every percentile, by almost 3x on average, and by more than 10x in eleven of the 25 experiments. Compared to SIZE-OPT, AdaptSize achieves 81% of the OHR for all percentiles.

6.4 Side effects of Size-Aware Admission

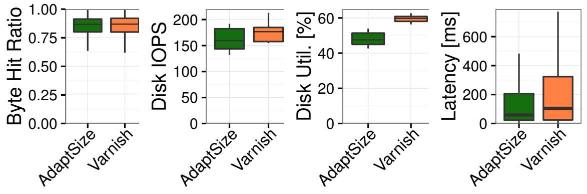
So far our evaluation has focused on AdaptSize's improvement with regard to the OHR. We evaluate AdaptSize's side-effects on the DC and on the client's request latency (cf. Section 2). Specifically, we compare AdaptSize to an unmodified Varnish system using the setup explained in Section 5.3. Network latencies are emulated using the Linux kernel (tc-netem). We set a 30ms round-trip latency between client and CDN server, and 100ms round-trip latency between CDN server and origin server. We answer the following three questions on the CDN server's performance.

How much does AdaptSize affect the BHR of the DC? Quick answer: *AdaptSize has a neutral effect on the BHR of the DC.* The DC's goal is to maximize the BHR, which is achieved by a very large DC capacity [49]. In fact, compared to the DC the HOC has less than on thousandth the capacity. Therefore, changes to the HOC have little effect on the DC's BHR.

In our experiment, we measure the DC's byte hit ratio (BHR) from the origin server. Figure 15a shows that there is no noticeable difference between the BHR under AdaptSize and under an unmodified Varnish.

Does AdaptSize increase the load of the DC's hard-disks? Quick answer: *No. In fact, AdaptSize reduces the average disk utilization by 20%.* With AdaptSize, the HOC admits fewer large objects, but caches many more small objects. The DC's request traffic therefore consists of more requests to large objects, and significantly fewer requests to small objects.

We measure the request size distribution at the DC and report the corresponding histogram in Figure 16. We observe that AdaptSize decreases the number of cache misses significantly for all object sizes below 256 KiB. For object sizes above 256 KiB, we observe a slight increase in the number of cache misses. Overall, we find that the DC has to serve 60% fewer requests with AdaptSize, but that the disks have to transfer a 30% higher



(a) BHR. (b) IOPS. (c) Utilization. (d) Latency.

Figure 15: Evaluation of AdaptSize's side effects across ten different sections of the US trace. AdaptSize has a neutral impact on the byte hit ratio, and leads to a 10% reduction in the median number of I/O operations going to the disk, and a 20% reduction in disk utilization.

byte volume. The average request size is also 4x larger with AdaptSize, which improves the sequentiality of disk access and thus makes the DC's disks more efficient.

To quantify the performance impact on the DC's hard-disks we use iostat [31]. Figure 15b shows that the average rate of I/O operations per second decreases by about 10%. Moreover, Figure 15c shows that AdaptSize reduces the disk's utilization (the fraction of time with busy periods) by more than 20%. We conclude that the increase in byte volume is more than offset by the fact that AdaptSize shields the DC from many small requests and also improves the sequentiality of requests served by the DC.

How much does AdaptSize reduce the request latency? Quick answer: *AdaptSize reduces the request latency across all percentiles by at least 30%.*

We measure the end-to-end request latency (time until completion of a request) from the client server. Figure 15d shows that AdaptSize reduces the median request latency by 43%, which is mostly achieved by the fast HOC answering a higher fraction of requests. The figure also shows significant reduction of tail latency, e.g., the 90-th and 99-th latency percentiles are reduced by more than 30%. This reduction in the tail latency is due to the DC's improved utilization factor, which leads to a much smaller number of outstanding requests, which makes it easier to absorb traffic bursts.

7 Related Work

The extensive related work in caching can be divided into two major lines of work: research caching systems (Section 7.1), and cache tuning methods (Section 7.2).

7.1 Research caching systems

Table 2 surveys 33 caching systems proposed in the research literature between 1993 and 2016. We classify these systems in terms of the per-request time complexity, the eviction and admission policies used, the support for a concurrent implementation, and the evaluation method.

Not all of the 33 caching systems fulfill the low overhead design goal of Section 2. Specifically, the complexity column in Table 2 shows that some proposals before 2002 have a computational overhead that scales logarithmically in the number of objects in the cache, which is impractical. AdaptSize differs from these systems because it has a constant complexity, and a low synchronization overhead, which we demonstrated by incorporating AdaptSize into the Varnish production system.

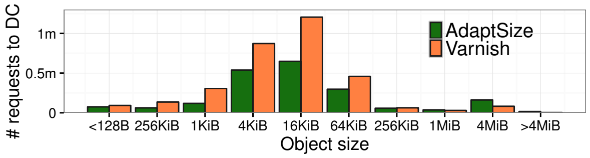


Figure 16: Comparison of the distribution of request sizes to the disk cache under a HOC running AdaptSize versus unmodified Varnish. All object sizes below 256 KiB are significantly less frequent under AdaptSize, whereas larger objects are slightly more frequent.

Of those caching systems that have a low overhead, almost none (except LRU-S and Threshold) incorporate object sizes. In particular, these systems admit and evict objects based only on recency, frequency, or a combination thereof. AdaptSize differs from these systems because it is size aware, which improves the OHR by 33-46% (as shown in Section 6.2).

There are only three low-overhead caching systems that are size aware. Threshold [2] uses a static size threshold, which has to be determined in advance. The corresponding Static policy in Section 6.3 performs poorly in our experiments. LRU-S [72] uses size-aware admission, where it admits objects with probability $1/size$. Unfortunately, this static probability is too low[6]. AdaptSize achieves a 61-78% OHR improvement over LRU-S (Figures 5 and 13). The third system [56] also uses a static parameter, and was developed in parallel to AdaptSize. AdaptSize differs from these caching systems by automatically adapting the size-aware admission parameter over time.

While most of these caching systems share our goal of improving the OHR, an orthogonal line of research seeks to achieve superior throughput using concurrent cache implementations (compare the concurrent implementation column in Table 2). AdaptSize also uses a concurrent implementation and achieves throughput comparable to production systems (Section 6.1). AdaptSize differs from these systems by improving the OHR – without sacrificing cache throughput.

The last column in Table 2 shows that most recent caching systems are evaluated using prototype implementations. Likewise, this work evaluates an actual implementation of AdaptSize (Sections 5 and 6) through experiments in a dedicated data center. We additionally use trace-driven simulations to compare to some of those systems that have only been used in simulations.

[6] We also tested several variants of LRU-S. We were either confronted with a cache tuning problem with no obvious solution (Section 3.2), or (by removing the admission component) with an OHR similar to LRU.

Name	Year	Complexity	Admission Policy	Eviction Policy	Concurrent Imp.	Evaluation
AdaptSize	**2016**	$O(1)$	**size**	**recency**	**yes**	**implementation**
Cliffhanger [14]	2016	$O(1)$	none	recency	no	implementation
Billion [47]	2015	$O(1)$	none	recency	yes	implementation
BloomFilter [49]	2015	$O(1)$	frequency	recency	no	implementation
SLRU [29]	2015	$O(1)$	none	recency+frequency	no	analysis
Lama [34]	2015	$O(1)$	none	recency	no	implementation
DynaCache [13]	2015	$O(1)$	none	recency	no	implementation
MICA [48]	2014	$O(1)$	none	recency	yes	implementation
TLRU [20]	2014	$O(1)$	frequency	recency	no	simulation
MemC3 [23]	2013	$O(1)$	none	recency	yes	implementation
S4LRU [35]	2013	$O(1)$	none	recency+frequency	no	simulation
CFLRU [62]	2006	$O(1)$	none	recency+cost	no	simulation
Clock-Pro [38]	2005	$O(1)$	none	recency+frequency	yes	simulation
CAR [7]	2004	$O(1)$	none	recency+frequency	yes	simulation
ARC [53]	2003	$O(1)$	none	recency+frequency	no	simulation
LIRS [39]	2002	$O(1)$	none	recency+frequency	no	simulation
LUV [6]	2002	$O(\log n)$	none	recency+size	no	simulation
MQ [81]	2001	$O(1)$	none	recency+frequency	no	simulation
PGDS [12]	2001	$O(\log n)$	none	recency+frequency+size	no	simulation
GD* [40]	2001	$O(\log n)$	none	recency+frequency+size	no	simulation
LRU-S [72]	2001	$O(1)$	size	recency+size	no	simulation
LRV [66]	2000	$O(\log n)$	none	frequency+recency+size	no	simulation
LFU-DA [5, 70]	2000	$O(1)$	none	frequency	no	simulation
LRFU [46]	1999	$O(\log n)$	none	recency+frequency	no	simulation
PSS [3]	1999	$O(\log n)$	frequency	frequency+size	no	simulation
GDS [11]	1997	$O(\log n)$	none	recency+size	no	simulation
Hybrid [79]	1997	$O(\log n)$	none	recency+frequency+size	no	simulation
SIZE [1]	1996	$O(\log n)$	none	size	no	simulation
Hyper [1]	1996	$O(\log n)$	none	frequency+recency	no	simulation
Log2(SIZE) [2]	1995	$O(\log n)$	none	recency+size	no	simulation
LRU-MIN [2]	1995	$O(n)$	none	recency+size	no	simulation
Threshold [2]	1995	$O(1)$	size	recency	no	simulation
2Q [41]	1994	$O(1)$	frequency	recency+frequency	no	simulation
LRU-K [58]	1993	$O(\log n)$	none	recency+frequency	no	implementation

Table 2: Historical overview of web caching systems.

7.2 Cache tuning methods

While tuning for size-based admission is entirely new, tuning has been used in other caching contexts such as tuning for the optimal balance between recency and frequency [41, 53, 46, 39, 81, 7, 9] and for the allocation of capacity to cache partitions [14, 34, 13, 69].

In these other contexts, the most common tuning approach is hill climbing with shadow caches [41, 53, 46, 39, 81, 7, 14]. Section 3.2 discusses why this approach often performs poorly when tuning size-aware admission, and Section 6 provides corresponding experimental evidence.

Another method involves a prediction model together with a global search algorithm. The most widely used prediction model is the calculation of stack distances [51, 4, 78, 77], which has been recently used as an alternative to shadow caches [69, 13, 69]. Unfortunately, the stack distance model is not suited to optimizing the parameters of an admission policy, since each admission parameter leads to a different request sequence and thus a different stack distance distribution that needs to be recalculated.

Many cache models have been studied in the CS theory community [44, 30, 15, 52, 10, 33, 17, 75, 25, 24, 36, 19, 68, 16, 37, 61, 65, 28, 80, 59, 27, 50, 8, 29, 9]. Unfortunately, all these models assume unit-sized objects. AdaptSize's Markov model allows to model size-aware admission and variable-sized objects.

8 Conclusion

AdaptSize is a new caching system for the hot object cache in CDN servers. The power of AdaptSize stems from a size-aware admission policy that is continuously optimized using a new Markov model of the HOC. In experiments with Akamai production traces, we show that AdaptSize vastly improves the OHR over both state-of-the-art production systems and research systems. We also show that our implementation of AdaptSize is robust and scalable, and improves the DC's disk utilization.

As more diverse applications with richer content migrate onto the Internet, future CDNs will experience even greater variability in request patterns and object sizes. We believe that AdaptSize and its underlying mathematical model will be valuable in addressing this challenge.

9 Acknowledgments

We thank Jens Schmitt, Sebastian Michel, our shepherd Mahesh Balakrishnan, and our anonymous reviewers for their valuable feedback.

This research is supported in part by a Google Faculty Award, and NSF grants CNS-1413998, CMMI-1538204, CMMI-1334194, and XPS-1629444.

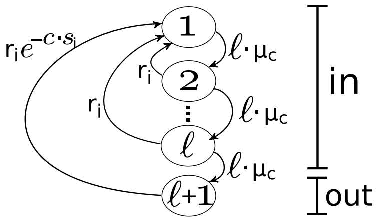
Appendix

A Proof of Theorem 1

The result in Theorem 1 is achieved by solving the Markov chain shown in Figure 8.

The key challenge when solving this chain is that the length of the LRU list changes over time. We solve this by using a mathematical convergence result.

We consider a fixed object i, and a fixed size-aware admission parameter c. Let ℓ denote the length of the LRU list. Now the Markov chain has $\ell+1$ states: one for each position in the list and one to represent the object is out of the cache, as shown below:



Over time, ℓ changes as either larger or small objects populate the cache. However, what remains constant is the expected time for an object to get evicted (if it is not requested again) as this time only depends on the overall admission rate (i.e. the size-aware admission parameter c), which is independent of ℓ. Using this insight, we modify the Markov chain to increase the push-down rate μ by a factor of ℓ: now, the expected time to traverse from position 1 to $\ell+1$ (without new requests) is constant at $1/\mu$.

We now solve the Markov chain for a fixed ℓ and obtain the limiting probability π_i of each position $i \in \{0,\ldots,\ell,\ell+1\}$. Using the π_i, we can now derive the limiting probability of being "in" the cache, $\pi_{in} = \sum_{i=0}^{\ell} \pi_i$, which can be algebraically simplified to:

$$\pi_{in} = 1 - \frac{\left(\frac{\ell}{\ell + r_i/\mu}\right)^{\ell}}{e^{-s_i/c} + \left(\frac{\ell}{\ell + r_i/\mu}\right)^{\ell} - e^{-s_i/c}\left(\frac{\ell}{\ell + r_i/\mu}\right)^{\ell}}$$

We observe that the π_{in} quickly converges in ℓ; numerically, convergence happens around $\ell > 100$. In our simulations, the cache typically holds many more objects than 100, simultaneously. Therefore, it is reasonable to always use the converged result $\ell \to \infty$. We formally solve this limit for π_{in} and obtain the closed-form solution of the long-term probability that object i is present in the cache, as stated in Theorem 1.

We remark that our convergence result uses a similar intuition as recent studies on equal-sized objects [60, 27], which is that the time it takes an object to get from position 1 to $\ell+1$ (if there are no further requests to it) converges to a constant in a LRU cache. What makes AdaptSize's model different from these models is that we consider size-aware admission and variable object sizes.

References

[1] Abrams, M., Standridge, C. R., Abdulla, G., Fox, E. A., and Williams, S. Removal policies in network caches for World-Wide Web documents. In *ACM SIGCOMM* (1996), pp. 293–305.

[2] Abrams, M., Standridge, C. R., Abdulla, G., Williams, S., and Fox, E. A. Caching Proxies: Limitations and Potentials. Tech. rep., Virginia Polytechnic Institute & State University Blacksburgh, VA, 1995.

[3] Aggarwal, C., Wolf, J. L., and Yu, P. S. Caching on the world wide web. *IEEE Transactions on Knowledge and Data Engineering 11*, 1 (1999), 94–107.

[4] Almasi, G., Caşcaval, C., and Padua, D. A. Calculating stack distances efficiently. In *ACM SIGPLAN Notices* (2002), vol. 38, pp. 37–43.

[5] Arlitt, M., Cherkasova, L., Dilley, J., Friedrich, R., and Jin, T. Evaluating content management techniques for web proxy caches. *Performance Evaluation Review 27*, 4 (2000), 3–11.

[6] Bahn, H., Koh, K., Noh, S. H., and Lyul, S. Efficient replacement of nonuniform objects in web caches. *IEEE Computer 35*, 6 (2002), 65–73.

[7] Bansal, S., and Modha, D. S. CAR: Clock with adaptive replacement. In *USENIX FAST* (2004), vol. 4, pp. 187–200.

[8] Berger, D. S., Gland, P., Singla, S., and Ciucu, F. Exact analysis of TTL cache networks. *Perform. Eval. 79* (2014), 2 – 23. Special Issue: Performance 2014.

[9] Berger, D. S., Henningsen, S., Ciucu, F., and Schmitt, J. B. Maximizing cache hit ratios by variance reduction. *ACM SIGMETRICS Performance Evaluation Review 43*, 2 (2015), 57–59.

[10] Burville, P., and Kingman, J. On a model for storage and search. *Journal of Applied Probability* (1973), 697–701.

[11] Cao, P., and Irani, S. Cost-aware WWW proxy caching algorithms. In *USENIX symposium on Internet technologies and systems* (1997), vol. 12, pp. 193–206.

[12] Cherkasova, L., and Ciardo, G. Role of aging, frequency, and size in web cache replacement policies. In *High-Performance Computing and Networking* (2001), pp. 114–123.

[13] Cidon, A., Eisenman, A., Alizadeh, M., and Katti, S. Dynacache: dynamic cloud caching. In *USENIX HotCloud* (2015).

[14] Cidon, A., Eisenman, A., Alizadeh, M., and Katti, S. Cliffhanger: Scaling performance cliffs in web memory caches. In *USENIX NSDI* (2016), pp. 379–392.

[15] Coffman, E. G., and Denning, P. J. *Operating systems theory*. Prentice-Hall, 1973.

[16] Coffman, E. G., and Jelenković, P. Performance of the move-to-front algorithm with Markov-modulated request sequences. *Operations Research Letters 25* (1999), 109–118.

[17] Dan, A., and Towsley, D. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *ACM SIGMETRICS* (1990), pp. 143–152.

[18] Dilley, J., Maggs, B. M., Parikh, J., Prokop, H., Sitaraman, R. K., and Weihl, W. E. Globally distributed content delivery. *IEEE Internet Computing 6*, 5 (2002), 50–58.

[19] Dobrow, R. P., and Fill, J. A. The move-to-front rule for self-organizing lists with Markov dependent requests. In *Discrete Probability and Algorithms*. Springer, 1995, pp. 57–80.

[20] Einziger, G., and Friedman, R. Tinylfu: A highly efficient cache admission policy. In *IEE Euromicro PDP* (2014), pp. 146–153.

[21] ElAarag, H., Romano, S., and Cobb, J. *Web Proxy Cache Replacement Strategies: Simulation, Implementation, and Performance Evaluation*. Springer Briefs in Computer Science. Springer London, 2013.

[22] Era—Trends, C. V. G. I. T. F. T. Z., and Analysis. CISCO VNI global IP traffic forecast: The zettabyte era—trends and analysis, May 2015. Available at `http://goo.gl/wxuvVk`, accessed 09/12/16.

[23] Fan, B., Andersen, D. G., and Kaminsky, M. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *USENIX NSDI* (2013), pp. 371–384.

[24] Fill, J. A., and Holst, L. On the distribution of search cost for the move-to-front rule. *Random Structures & Algorithms 8* (1996), 179–186.

[25] Flajolet, P., Gardy, D., and Thimonier, L. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics 39* (1992), 207–229.

[26] Fofack, N. C., Dehghan, M., Towsley, D., Badov, M., and Goeckel, D. L. On the performance of general cache networks. In *VALUETOOLS* (2014), pp. 106–113.

[27] Fricker, C., Robert, P., and Roberts, J. A versatile and accurate approximation for LRU cache performance. In *ITC* (2012), p. 8.

[28] Gallo, M., Kauffmann, B., Muscariello, L., Simonian, A., and Tanguy, C. Performance evaluation of the random replacement policy for networks of caches. In *ACM SIGMETRICS/ PERFORMANCE* (2012), pp. 395–396.

[29] Gast, N., and Van Houdt, B. Transient and steady-state regime of a family of list-based cache replacement algorithms. In *ACM SIGMETRICS* (2015), pp. 123–136.

[30] Gelenbe, E. A unified approach to the evaluation of a class of replacement algorithms. *IEEE Transactions on Computers 100* (1973), 611–618.

[31] Godard, S. Iostat, 2015. Available at `http://goo.gl/JZmbUp`, accessed 09/12/16.

[32] Graziano, P. Speed up your web site with Varnish. *Linux Journal 2013*, 227 (2013), 4.

[33] Hendricks, W. The stationary distribution of an interesting Markov chain. *Journal of Applied Probability* (1972), 231–233.

[34] Hu, X., Wang, X., Li, Y., Zhou, L., Luo, Y., Ding, C., Jiang, S., and Wang, Z. LAMA: Optimized locality-aware memory allocation for key-value cache. In *USENIX ATC* (2015), pp. 57–69.

[35] Huang, Q., Birman, K., van Renesse, R., Lloyd, W., Kumar, S., and Li, H. C. An analysis of Facebook photo caching. In *ACM SOSP* (2013), pp. 167–181.

[36] Jelenković, P. R. Asymptotic approximation of the move-to-front search cost distribution and least-recently used caching fault probabilities. *The Annals of Applied Probability 9* (1999), 430–464.

[37] Jelenković, P. R., and Radovanović, A. Least-recently-used caching with dependent requests. *Theoretical computer science 326* (2004), 293–327.

[38] Jiang, S., Chen, F., and Zhang, X. CLOCK-Pro: An effective improvement of the clock replacement. In *USENIX ATC* (2005), pp. 323–336.

[39] Jiang, S., and Zhang, X. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS 30*, 1 (2002), 31–42.

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

[41] Johnson, T., and Shasha, D. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB* (1994), pp. 439–450.

[42] Kamp, P.-H. Varnish notes from the architect, 2006. Available at `https://www.varnish-cache.org/docs/trunk/phk/notes.html`, accessed 09/12/16.

[43] Kamp, P.-H. Varnish LRU architecture, June 2007. Available at `https://www.varnish-cache.org/trac/wiki/ArchitectureLRU`, accessed 09/12/16.

[44] King, W. F. Analysis of demand paging algorithms. In *IFIP Congress (1)* (1971), pp. 485–490.

[45] Le Boudec, J.-Y., McDonald, D., and Mundinger, J. A generic mean field convergence result for systems of interacting objects. In *Quantitative Evaluation of Systems* (2007), IEEE, pp. 3–18.

[46] Lee, D., Choi, J., Kim, J.-H., Noh, S. H., Min, S. L., Cho, Y., and Kim, C. S. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *ACM SIGMETRICS* (1999), vol. 27, pp. 134–143.

[47] Li, S., Lim, H., Lee, V. W., Ahn, J. H., Kalia, A., Kaminsky, M., Andersen, D. G., Seongil, O., Lee, S., and Dubey, P. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *ACM ISCA* (2015), pp. 476–488.

[48] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. MICA: A holistic approach to fast in-memory key-value storage. In *USENIX NSDI* (2014), pp. 429–444.

[49] Maggs, B. M., and Sitaraman, R. K. Algorithmic nuggets in content delivery. *ACM SIGCOMM CCR 45* (2015), 52–66.

[50] Martina, V., Garetto, M., and Leonardi, E. A unified approach to the performance analysis of caching systems. In *IEEE INFOCOM* (2014).

[51] Mattson, R. L., Gecsei, J., Slutz, D. R., and Traiger, I. L. Evaluation techniques for storage hierarchies. *IBM Systems journal 9*, 2 (1970), 78–117.

[52] McCabe, J. On serial files with relocatable records. *Operations Research 13* (1965), 609–618.

[53] Megiddo, N., and Modha, D. S. ARC: A self-tuning, low overhead replacement cache. In *USENIX FAST* (2003), vol. 3, pp. 115–130.

[54] Modern network design, November 2016. Available at `https://www.fastly.com/products/modern-network-design`, accessed 02/17/17.

[55] Motwani, R., and Raghavan, P. *Randomized algorithms*. Chapman & Hall/CRC, 2010.

[56] Neglia, G., Carra, D., Feng, M., Janardhan, V., Michiardi, P., and Tsigkari, D. Access-time aware cache algorithms. In *IEEE ITC* (2016), vol. 1, pp. 148–156.

[57] Nygren, E., Sitaraman, R. K., and Sun, J. The Akamai Network: A platform for high-performance Internet applications. *ACM SIGOPS Operating Systems Review 44*, 3 (2010), 2–19.

[58] O'Neil, E. J., O'Neil, P. E., and Weikum, G. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD 22*, 2 (1993), 297–306.

[59] O'Neil, E. J., O'Neil, P. E., and Weikum, G. An optimality proof of the LRU-K page replacement algorithm. *JACM 46* (1999), 92–112.

[60] Osogami, T. A fluid limit for a cache algorithm with general request processes. *Advances in Applied Probability 42*, 3 (2010), 816–833.

[61] Panagakis, A., Vaios, A., and Stavrakakis, I. Approximate analysis of LRU in the case of short term correlations. *Computer Networks 52* (2008), 1142–1152.

[62] Park, S.-y., Jung, D., Kang, J.-u., Kim, J.-s., and Lee, J. CFLRU: a replacement algorithm for flash memory. In *ACM/IEEE CASES* (2006), pp. 234–241.

[63] PETRUSHEV, P. P., AND POPOV, V. A. *Rational approximation of real functions*, vol. 28. Cambridge University Press, 2011.

[64] POŠÍK, P., HUYER, W., AND PÁL, L. A comparison of global search algorithms for continuous black box optimization. *Evolutionary computation 20*, 4 (2012), 509–541.

[65] PSOUNIS, K., ZHU, A., PRABHAKAR, B., AND MOTWANI, R. Modeling correlations in web traces and implications for designing replacement policies. *Computer Networks 45* (2004), 379–398.

[66] RIZZO, L., AND VICISANO, L. Replacement policies for a proxy cache. *IEEE/ACM TON 8* (2000), 158–170.

[67] ROCCA, E. Running Wikipedia.org, June 2016. Available at `https://www.mediawiki.org/wiki/File:WMF_Traffic_Varnishcon_2016.pdf`, accessed 09/12/16.

[68] RODRIGUES, E. R. The performance of the move-to-front scheme under some particular forms of Markov requests. *Journal of applied probability* (1995), 1089–1102.

[69] SAEMUNDSSON, T., BJORNSSON, H., CHOCKLER, G., AND VIGFUSSON, Y. Dynamic performance profiling of cloud caches. In *ACM SoCC* (2014), pp. 1–14.

[70] SHAH, K., MITRA, A., AND MATANI, D. An O(1) algorithm for implementing the LFU cache eviction scheme. Tech. rep., Stony Brook University, 2010.

[71] SITARAMAN, R. K., KASBEKAR, M., LICHTENSTEIN, W., AND JAIN, M. Overlay networks: An Akamai perspective. In *Advanced Content Delivery, Streaming, and Cloud Services*. John Wiley & Sons, 2014.

[72] STAROBINSKI, D., AND TSE, D. Probabilistic methods for web caching. *Perform. Eval. 46* (2001), 125–137.

[73] TANGE, O. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine 36*, 1 (Feb 2011), 42–47.

[74] Alexa top sites on the web, March 2016. `http:///www.alexa.com/topsites`, accessed 03/16/16.

[75] TSUKADA, N., HIRADE, R., AND MIYOSHI, N. Fluid limit analysis of FIFO and RR caching for independent reference model. *Perform. Eval. 69* (Sept. 2012), 403–412.

[76] VELÁZQUEZ, F., LYNGSTØL, K., FOG HEEN, T., AND RENARD, J. *The Varnish Book for Varnish 4.0*. Varnish Software AS, March 2016.

[77] WALDSPURGER, C. A., PARK, N., GARTHWAITE, A., AND AHMAD, I. Efficient MRC construction with SHARDS. In *USENIX FAST* (2015), pp. 95–110.

[78] WIRES, J., INGRAM, S., DRUDI, Z., HARVEY, N. J., AND WARFIELD, A. Characterizing storage workloads with counter stacks. In *USENIX OSDI* (2014), pp. 335–349.

[79] WOOSTER, R. P., AND ABRAMS, M. Proxy caching that estimates page load delays. *Computer Networks and ISDN Systems 29*, 8 (1997), 977–986.

[80] YOUNG, N. E. Online paging against adversarially biased random inputs. *Journal of Algorithms 37* (2000), 218–235.

[81] ZHOU, Y., PHILBIN, J., AND LI, K. The multi-queue replacement algorithm for second level buffer caches. In *USENIX ATC* (2001), pp. 91–104.