Achieving high cache hit ratios for CDN memory caches with size-aware admission

Daniel S. Berger*, Ramesh K. Sitaraman+, Mor Harchol-Balter

July 4, 2016
CMU-CS-16-120

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

* University of Kaiserslautern
+ University of Massachusetts at Amherst & Akamai Technologies

Keywords: web caching; cache admission control, content delivery networks; cache hit ratio; object hit ratio; cache tuning

Abstract

Content delivery networks (CDNs) allow billions of users to expediently access content with higher reliability and performance from proximal "edge" caches deployed around the world. Each edge cache typically uses an in-memory Hot Object Cache (HOC) and a disk-based cache. While the HOC provides very fast response times, it is also very small. Maximizing the object hit ratio (OHR) of the HOCs is a major goal of a CDN.

Almost all prior work on caching policies has focused on improving the OHR via highly complex *eviction* policies. In contrast, this work investigates the superiority of size-aware *admission* policies, where the emphasis is on simplicity. Specifically, we propose two policies that either admit an object with some probability proportional to its size, or as a simple size threshold.

Our results are based on two expansive production HOC traces from 2015 captured in the Akamai CDN. Unlike older traces, we find that object sizes in these traces span ten orders of magnitude, and that highly popular objects have an order of magnitude smaller size on average. Our simple policies efficiently exploit this property and achieve a 23-92% OHR improvement over state-of-the-art research-based and commercial caching systems.

The superior performance of our policies stems from a new statistical cache tuning method, wich automatically adapts the parameters of our admission policies to the request traffic. We find that our statistical tuning method is significantly superior to state-of-the-art cache tuning methods such as hill climbing. Our method consistently achieves 80% of the OHR of offline parameter tuning. To demonstrate feasibility in a production setting, we show that our tuning method can be incorporated into a high-performance caching system with low processing and memory overheads and negligible impact on cache server throughput.

1 Introduction

Content delivery networks (CDNs) [24] are key to the Internet ecosystem today, allowing billions of users to expediently access content with higher reliability and performance from proximal "edge" servers deployed around the world. Most of the world's major web sites, video portals, e-commerce sites, and social networks are served by CDN caches today. It is estimated that CDNs will account for nearly two-thirds of the Internet traffic within five years [20]. A large CDN, such as that operated by Akamai [59], serves trillions of user requests a day from 170,000+ servers located in 1500+ networks in 100+ countries around the world.

CDNs enhance performance by caching objects in servers close to users and rapidly delivering those objects to users. A CDN server employs two levels of caching: a smaller but faster cache that is resident in memory called the Hot Object Cache (HOC) and a larger but slower second-level cache that is resident in disk called the Disk Cache (DC). Each requested object is first looked up in the HOC, and if absent looked up in the DC. We call the fraction of requests satisfied by the HOC its *object hit ratio* (OHR). A major goal of a CDN is to *maximize the OHR* of its HOCs[1] to provide millisecond response times for object retrieval.

Maximizing the OHR is challenging because web object sizes are highly variable and HOCs have a small capacity. In fact, the HOC capacity of a CDN server is typically of the same order of magnitude as the largest objects that it serves. Further, HOC implementations need to have a *low processing overhead*. As the first caching level of a CDN edge server, the HOC processes thousands of requests per second, requiring simple and scalable cache management. The HOC must also *adapt* to changing request patterns, such as those caused by flash crowds that result in rapid surges in the popularity of individual web sites served by the CDN.

1.1 Cache management policies

Conceptually, cache management entails two types of decisions. First, a cache can decide whether or not to admit an object (cache admission). Second, a cache can decide which object to evict from the cache (cache eviction), if there is no space for a newly admitted object. While cache eviction has received significant attention for over three decades, much less is known about cache admission. Even when cache admission policies are considered, the policies are not size-aware. For instance, production web caching systems used in practice use simple LRU eviction and either no admission policy, or an admission policy that does not depend on the object size. The most popular web caching systems, Varnish and Nginx, by default use LRU eviction and either no admission policy or a size-unaware admission policy (Section 2).

Size-aware admission policies have rarely been considered in the research literature either. In the early 1990s, a fixed size threshold (admit object if it's size is below threshold T) was considered [1]. However, fixed size thresholds were rejected by the late 1990s, when studies showed that "the benefits of size-based caching policies have diminished" [9]. Cache admission policies have not

[1] CDNs also optimize a related metric called the byte hit rate (BHR) that is the fraction of *bytes* that are served from cache. BHR is more relevant for (larger) disk caches [73]. The focus of this paper is on smaller HOC caches, which are performance focused as measured by their OHR.

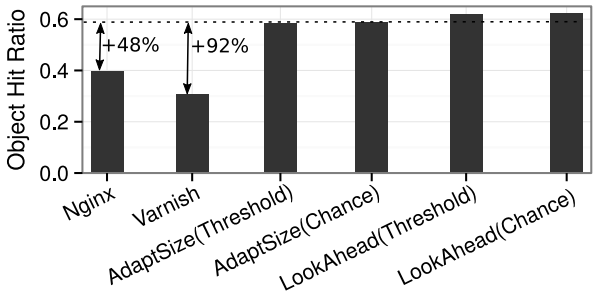


Figure 1: Comparison of AdaptSize's implementation to state-of-the-art production systems. AdaptSize improves the OHR by 48-92%. AdaptSize also achieves 95% of the OHR of LookAhead, which has future knowledge. These results are for the US trace and a typical HOC size (to be defined in Section 4.1).

been considered in recent studies on web caching [38, 75, 28] or in recent work on improving hit ratios for in-memory caches [58, 18, 37, 19].

1.2 Our contributions

We propose a new cache admission policy called *AdaptSize*, the first caching system to uses a size-aware admission policy that is continuously adapted to the request traffic. We propose two variants. AdaptSize(Chance) probabilistically decides whether or not to admit an object into the cache with the admission probability equal to $e^{-size/c}$, where c is a tunable parameter. AdaptSize(Threshold) admits an object if its size is below a tunable parameter c. Our main contributions follow.

1. *Exposing the power of size-aware cache admission.* We provide the first experimental evidence that size-aware admission policies achieve much higher OHR than other low-overhead state-of-the-art caching systems. AdaptSize achieves higher OHR than commonly-used production systems such as Varnish and Nginx and many well-known caching policies proposed in research literature.
2. *Rigorous evaluation using extensive production traces from one of the world's largest CDNs.* We evaluate AdaptSize on production request traces from Akamai, one of the world's largest CDNs. Figure 1 compares both AdaptSize variants to the popular Nginx and Varnish caching systems for a typical HOC size. We find that AdaptSize improves the object hit ratio by 47-48% over Nginx, and that AdaptSize improves the OHR by 91-93% over Varnish. Figure 2 compares AdaptSize to research-based caching systems from the literature. AdaptSize improves the OHR over the next best system by 31%.
3. *Achieving near-optimal OHR.* We formulate a cache admission policy called LookAhead that has perfect knowledge of a large window of future requests and picks the optimal size threshold parameter c based on that knowledge. We show that AdaptSize consistently achieves an OHR close to the LookAhead policy, *even without the benefit of knowledge of future requests*. In Figure 1, AdaptSize(Threshold) and AdaptSize(Chance) achieve 95% of the OHR of their respective LookAhead policies.

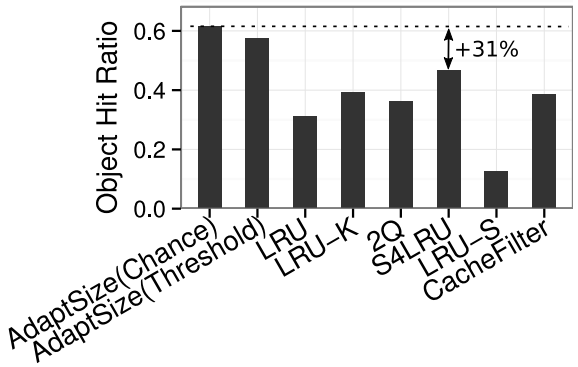


Figure 2: Comparison of AdaptSize to other efficient caching systems from Table 1 using trace-driven simulations. AdaptSize achieves a 32% higher OHR than the next best caching system. These results are for the US trace and a typical HOC size (to be defined in Section 4.1).

4. *Incorporating size-aware cache admission in production systems with low overhead.* To demonstrate the feasibility in a production environment, we incorporate AdaptSize into Varnish, a caching systems used by several prominent web sites and CDNs, including Wikipedia, Facebook, and Twitter. We empirically show that AdaptSize does not add significant processing or memory overheads and is able to maintain the high concurrency and high throughput of Varnish[47, 46].
5. *A new and robust cache tuning method that is better than state-of-the-art hill climbing techniques.* A reason why few current systems use size-aware admission policies is that they are hard to tune. For example, a fixed size threshold c improves the OHR only for a narrow parameter range. If the parameter is too small, the OHR can even drop below that of a cache without admission control. Additionally, changes in the request traffic require the threshold parameter to change rapidly over time. We propose a new statistical tuning method that continuously optimizes the parameter c used by AdaptSize. We show that AdaptSize performs near-optimally within 81% of LookAhead, even when the request patterns change drastically due to flash crowds or traffic mix changes. In contrast, the widely-used method of tuning caches via shadow caches [45, 56, 49, 43, 81, 8, 19], achieves only 23-43% of LookAhead's OHR. and tends to stagnate at local optima.

1.3 Roadmap

The rest of this paper is organized as follows. Section 2 positions AdaptSize's contribution in the context of prior work on caching systems. Section 3 details the design and implementation of AdaptSize. Our experimental setup is discussed in Section 4 and the results from our experimental evaluated are given in Section 5. We conclude in Section 6.

Name	Year	Eviction Policy	Admission Policy	Complexity	Synchronization overhead	Evaluation
AdaptSize	**2016**	**recency**	**size**	**O(1)**	**low**	**implementation**
Cliffhanger [19]	2016	recency	none	O(1)	high	implementation
Billion [50]	2015	recency	none	O(1)	zero	implementation
CacheFilter [52]	2015	recency	frequency	O(1)	high	implementation
Lama [37]	2015	recency	none	O(1)	high	implementation
DynaCache [18]	2015	recency	none	O(1)	high	implementation
MICA [51]	2014	recency	none	O(1)	zero	implementation
TinyLFU [26]	2014	recency	frequency	O(1)	high	simulation
MemC3 [27]	2013	recency	none	O(1)	low	implementation
S4LRU [38]	2013	recency+frequency	none	O(1)	high	simulation
CFLRU [64]	2006	recency+cost	none	O(1)	high	simulation
Clock-Pro [42]	2005	recency+frequency	none	O(1)	low	simulation
CAR [8]	2004	recency+frequency	none	O(1)	low	simulation
ARC [56]	2003	recency+frequency	none	O(1)	high	simulation
LIRS [43]	2002	recency+frequency	none	O(1)	high	simulation
LUV [7]	2002	recency+size	none	O(log n)	high	simulation
MQ [81]	2001	recency+frequency	none	O(1)	high	simulation
PGDS [16]	2001	recency+frequency +size	none	O(log n)	high	simulation
GD* [44]	2001	recency+frequency +size	none	O(log n)	high	simulation
LRU-S [74]	2001	recency+size	size	O(1)	high	simulation
LRV [69]	2000	frequency+recency +size	none	O(log n)	high	simulation
LFU-DA [6]	2000	frequency	none	O(log n)	high	simulation
LRFU [49]	1999	recency+frequency	none	O(log n)	high	simulation
PSS [3]	1999	frequency+size	frequency	O(log n)	high	simulation
GDS [14]	1997	recency+size	none	O(log n)	high	simulation
Hybrid [79]	1997	recency+frequency +size+cost	none	O(log n)	high	simulation
Mix [57]	1997	recency+size+cost	none	O(log n)	high	simulation
LNC-RW3 [72]	1997	frequency+recency +size+cost	none	O(n)	high	simulation
SIZE [2]	1996	size	none	O(log n)	high	simulation
Hyper [2]	1996	frequency+recency	none	O(log n)	high	simulation
Log2(SIZE) [1]	1995	recency+size	none	O(log n)	high	simulation
LRU-MIN [1]	1995	recency+size	none	O(n)	high	simulation
Threshold [1]	1995	recency	size	O(1)	high	simulation
2Q [45]	1994	recency+frequency	frequency	O(1)	high	simulation
LRU-K [60]	1993	recency+frequency	none	O(log n)	high	implementation

Table 1: Historical overview of web caching systems.

2 Background

There is a tremendous amount of literature related to caching and caching policies. This section presents a taxonomy of relevant prior work. We first consider existing production systems (Section 2.1), then discuss research-based caching systems (Section 2.2), and finally review cache tuning methods (Section 2.3). There is also a huge amount of work on cache models from the CS theory community that assumes unit-sized objects [48, 35, 22, 55, 13, 36, 23, 76, 30, 29, 40, 25, 70, 21, 41, 63, 68, 33, 80, 61, 32, 53, 10, 34, 11] and thus is less relevant for our work.

2.1 Production web caching systems

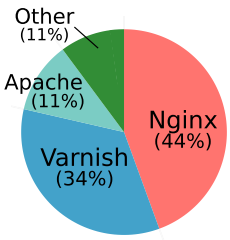
We surveyed the most popular web caches by crawling the Alexa top 5000 websites [4]. Figure 3a shows that three caching systems, Nginx, Varnish, and Apache, are used by 89% of the popular websites. Figure 3b lists key characteristics of these systems. Nginx uses LRU eviction and offers a frequency-based admission policy that admits objects after x requests. The Nginx OHR result in Figure 1 is based on an offline-optimized x parameter. AdaptSize still improves the OHR by 48% over Nginx, which motivates the need for size-aware admission. Varnish and Apache both use LRU and do not offer an admission policy by default.

2.2 Research-based caching systems

We survey 35 caching systems that were proposed in the research literature between 1993 and 2016, in Table 1. We classify these systems in terms of the eviction and admission policies used, time complexity, and concurrency.

Eviction and admission policies. While a variety of cache eviction policies are used, few systems use a size-aware admission policy. Table 1 lists the eviction and admission policy for 35 cache systems. Almost all systems use an eviction policy based on either recency or a combination of recency with other access characteristics. In contrast, very few systems use an admission policy: excluding AdaptSize there are only six systems that use an admission policy. Of these six systems, four use a frequency-based admission policies (similar to Nginx). Only two systems use a size-aware admission policy: LRU-S and Threshold. LRU-S uses a parameterless admission policy, which admits objects with probability $1/size$. Unfortunately, LRU-S performs worse than LRU without an admission policy (see Figure 2). Threshold uses a fixed size threshold, which has to be determined in advance. This is represented by our policy Static in Section 5.3, which has a poor performance in our experiments. AdaptSize is different from all these systems because it uses a size-aware admission policy and automatically adapts the admission parameter over time.

Low complexity and highly concurrent caching systems. Most recent systems focus on low complexity and highly concurrent caching systems. The complexity column in Table 1 shows that some proposals before 2002 have a computational overhead that scales logarithmically in the number of objects in the cache, which is impractical. Similarly, most systems before 2003 did



(a) Market share.

System	Eviction Policy	Admission Policy
Nginx	LRU	frequency-based
Varnish	LRU	none
Apache	LRU	none

(b) Key characteristics of the most popular open-source web caches.

Figure 3: The most widely-used web caching systems use simple LRU eviction and do not use a size-aware admission policy. We determined the market share of these systems by crawling the Alexa top 5000 websites [4].

not address the synchronization overhead in a concurrent cache implementation. Recent systems provide constant time complexity and often have a low (or even zero) synchronization overhead. AdaptSize focuses on improving the OHR while maintaining constant time complexity and high concurrency. Additionally, we also show that our statistical tuning method can be implemented without synchronization overhead on the request path.

2.3 Cache tuning methods

The most common approach to tuning cache parameters is the use of a shadow cache [45, 56, 49, 43, 81, 8, 19]. Each shadow cache simulates the effect of one parameter without storing the actual objects. A tuning step first evaluates several alternatives with corresponding shadow-caches, and then uses a variant of hill climbing to move towards a better configuration. Unfortunately, this local-search approach is prone to getting stuck at suboptimal parameter choices[2] in our experiments in Section 5

An alternative approach to cache tuning is the use of a prediction model together with a global search algorithm. The most widely used prediction model is the calculation of stack distances [54, 5, 78, 77]. This approach has been used by recent works as an alternative to shadow caches [37, 18, 71]. However, the stack distance model is not suited to optimizing the parameters of an admission policy, since each admission parameter setting leads to a different request sequence and thus a different stack distance distribution that needs to be recalculated. AdaptSize uses a new Markov-chain-based statistical model that uses aggregated request statistics, which makes the OHR predictions for admission parameters very cheap. In addition, our statistical model allows us to incorporate the request history via exponential smoothing, which is instrumental to making AdaptSize robust against typical short-term variability found in our traces.

[2]While there are more complicated shadow-cache search algorithms (e.g., using randomization), shadow caches rely on a fundamental assumption of stationarity, which does not need to apply to web traffic.

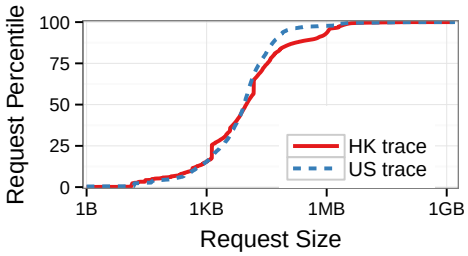


Figure 4: Object sizes are highly variable in both our HK trace and our US trace (see Section 4.1).

3 AdaptSize: Our Admission Policy

AdaptSize is motivated by the high variability of object sizes in CDN request traffic. Figure 4 shows the cumulative distribution of object sizes in two production traces, which are further described in Section 4.1. We find objects with sizes between 1 Byte and 1.6 GB. For caching systems that focus on the OHR (such as a HOC), the size variability has significant implications. We observe from Figure 4 that approximately 5% of objects have a size bigger than 1 MB. Every time a cache admits a 1 MB object, it needs to evict space equivalent to one thousand 1 KB objects, which make up about 15% of requests. Such evictions can clearly degrade the OHR.

A cache admission policy seeks to shield already admitted objects from being evicted again too soon. If objects can stay in the cache for a longer period, they are more likely to receive hits. Because high eviction numbers are triggered mainly by large objects, it is not sufficient to use a purely frequency-based admission policy.

AdaptSize makes admission decisions depending on an object's size and uses one of two size-aware policies. Both of them rely on a single parameter (called threshold parameter c). The first policy is deterministic, and admits objects with a size below a threshold parameter. The second policy is probabilistic, and the admission probability decreases exponentially with the object size.

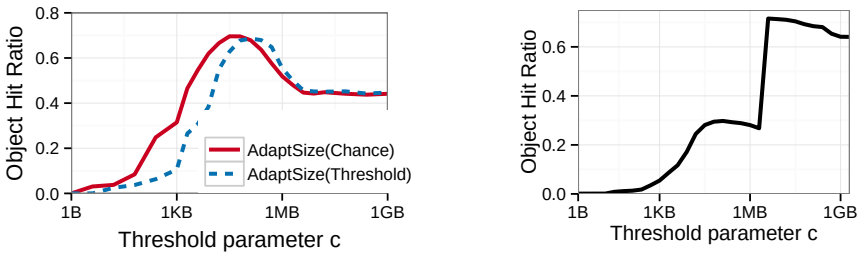
AdaptSize(Threshold) admit if size $<$ threshold c

AdaptSize(Chance) admit with probability $e^{-size/c}$

The probabilistic decision of AdaptSize(Chance) allows to incorporate both size and request frequency. For example, if there were a popular object with a size just above the threshold parameter c, that object can never get admitted with AdaptSize(Threshold). In contrast, AdaptSize(Chance) gives every request a small chance of being admitted, so that a popular object eventually can get into the cache. In our experiments in Section 5, AdaptSize(Chance) achieves slightly higher OHRs than AdaptSize(Threshold).

3.1 The need for tuning the size threshold

Figure 5a shows the OHR as a function of the threshold parameter c using AdaptSize(Threshold) and AdaptSize(Chance) for a typical HOC under one hour of the HK trace from Section 4.1. As we



(a) Typical hour of trace.

(b) Flash crowd hour.

Figure 5: Sensitivity of OHR to size threshold parameter c. (a) Both AdaptSize(Threshold) and AdaptSize(Chance) are very sensitive to c. (b) The shape of the sensitivity curve changes under flash crowds. These results are for the HK trace from Section 4.1.

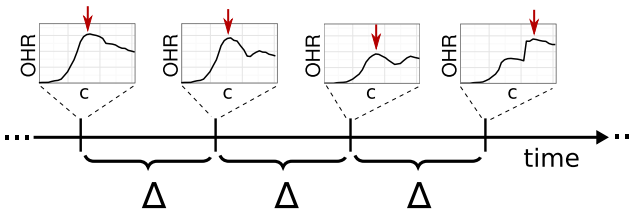


Figure 6: AdaptSize optimizes the threshold parameter every Δ requests. AdaptSize uses its statistical model to predict the OHR curve and then performs a global search for the optimal threshold parameter c.

see, the optimal threshold can move the OHR between 0 and 0.7. For AdaptSize(Threshold) the optimal c was 128 KB, whereas for AdaptSize(Chance) 32 KB.

The OHR is also very sensitive to changes in popularity. For example, a popularity change can happen when a video (with 2 MB chunks) is hotlinked from several news websites. Figure 5b shows the OHR curve during a flash crowd with 2 MB chunks. The threshold parameter, which was previously optimal, is now only a local optimum.

3.2 How AdaptSize adapts its size threshold

Section 3.1 shows that it is critical to adjust the threshold parameter of AdaptSize's admission policies over time. The goal of the statistical model is to find the optimal threshold parameter c for the next interval of Δ requests.

As shown in Figure 6, AdaptSize uses a two step process. First, AdaptSize constructs the OHR-vs-c sensitivity curve for every Δ interval. Second, AdaptSize performs a global search on each Δ's sensitivity curve for the optimal threshold parameter.

The first step of AdaptSize is based on a new statistical prediction model using Markov chains. The parameters of the Markov model are obtained using aggregated request statistics – the average request rate and size of each object – over each Δ interval. To increase prediction robustness, AdaptSize retains a history of request statistics across consecutive Δ intervals. Further details are given in Section 3.2.1.

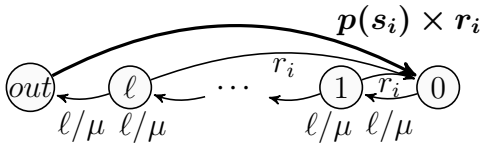


Figure 7: The AdaptSize Markov chain models the cache state for each object i. The admission policy is represented by the admission probability $p(s_i)$, the LRU head is state 0, and eviction happens at state ℓ.

The second step of AdaptSize searches for the global maximum using a standard algorithm. Further details are given in in Section 3.2.2.

3.2.1 First step: effect of threshold parameter c on OHR

The goal of this section is to derive the OHR for every threshold parameter c. This derivation is redone every Δ requests. We use $\Delta = 250K$ requests, but on both of our production traces our results are largely insensitive to Δ. To do the derivation, we will develop a theorem and two lemmas. The final lemma, Lemma 2 states the OHR as a function of c, which can be numerically evaluated.

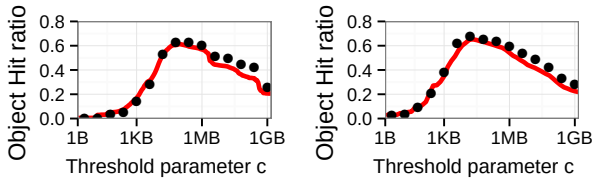
We use a Markov chain to model the caching state of each object i and use the following notation. Objects are characterized by their average request arrival rate r_i and size s_i. AdaptSize's two admission policies are represented via their admission probability $p(s_i)$.

$$AdaptSize(Threshold): \quad p(s_i) = \begin{cases} 1, & if s_i \leq c \\ 0, & if s_i > c \end{cases}$$
$$AdaptSize(Chance): \quad p(s_i) = e^{-c \cdot s_i}$$

In addition, we use two auxiliary variables, μ and ℓ. The first, μ, roughly represents the average time an object spends in the cache before being evicted. The second, ℓ, represents the length of the LRU list. Recall that LRU is an ordered list where objects enter at the top and are pushed towards the bottom, where they are evicted.

The Markov chain for an object i is shown in Figure 7. It consists consists of an "out" state (in which all objects start and return after they have been evicted) and ℓ states for the object's position in the LRU list. When object i is requested while out of the cache, it is moved to the first position (position 0), with probability $p(s_i)$ (cache admission). When object i is requested while being between position ℓ and 1, it is moved to the first position. Requests to objects other than i, lead to i being moved to a lower position on the LRU list (cache eviction). For each LRU position, this is modeled by the auxiliary variable μ/ℓ.

A key challenge to solving this Markov chain is that the length of the LRU list (ℓ) changes over time. We solve this problem by exploiting a recent mathematical result [62]. The result proves that the time it takes to get from 0 to out (if there are no further requests) is approximately the same over time. We can use this fact by letting ℓ go to infinity, which makes the time to get from 0 to out go to the deterministic number μ. Using this trick, we find a closed-form of the long-term probability that object i is present in the cache (Theorem 1).



(a) AdaptSize(Threshold). (b) AdaptSize(Chance).

Figure 8: The AdaptSize statistical model predicts the OHR sensitivity curve (solid line). (a) This is accurate for experimental results (black dots) for the AdaptSize(Threshold) policy (b) This is also accurate for the AdaptSize(Chance) policy. These results are for the US trace from Section 4.1.

Theorem 1.

$$\mathbb{P}[\textit{object } i \textit{ in cache}] = \frac{(e^{r_i\mu} - 1) \times p(s_i)}{1 + (e^{r_i\mu} - 1) \times p(s_i)} \ . \tag{1}$$

The probability of being in the cache depends on the auxiliary variable μ, which can be found by averaging over time. Specifically, we require that the average size of the cache in the model is equal to the actual cache size C. This equation is an extension of existing results for caching with unit-sized objects [15, 34, 32, 53].

Lemma 1.

$$\sum_{i=1}^{N} \mathbb{P}[\textit{object } i \textit{ in cache}] \ s_i = C \ .$$

Finally, the hit ratio of each object is found by using the observation that the expected number of hits of object i equals the average request rate of i 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.

Lemma 2.

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

Note that c is used in $p(s_i)$, which is used in $\mathbb{P}[\text{object } i \text{ in cache}]$.

At the end of the prediction step, we will have OHR curves like the ones shown in Figure 8. These curves are very accurate. As shown in the figures, the curves match experimental results across the whole range of the threshold parameter c.

3.2.2 Second step: global optimization of the threshold parameter

For each Δ interval we are capable of producing an OHR-vs-c plot (Figure 8). We now use standard techniques to search for the optimal c. Specifically, we use a systematic sampling of the search space combined with a local search method (as suggested in [66]). The systematic sampling uses

logarithmic step sizes (1B-2B, 2B-4B, etc) and starts in parallel from the smallest (c =1B) and largest threshold parameter (c =cache capacity). The local search method is a text book approach, golden section search, with the default parameters [67].

At the end of the parameter search step, we have found the threshold parameter that maximizes the OHR for each Δ interval as indicated in Figure 6.

3.3 Integration with a production system

We now describe implementing AdaptSize by incorporating it into Varnish, a production caching system. On a cache miss, Varnish accesses the second-level cache to retrieve the object, and places it in its HOC. With AdaptSize, an additional admission procedure is executed, and if the object is not admitted it is served from transient memory. We now outline two of our key ideas in creating an efficient and low-overhead implementation of AdaptSize.

3.3.1 Lock-free statistics aggregation.

Our statistical tuning method (Section 3) requires request statistics as input. Gathering such request statistics can add significant overhead to concurrent caching designs [71]. Varnish and AdaptSize use up to 5000 threads in our experiments, and centralized request counters would cause high lock contention. In fact, in our experiments, the throughput bottleneck is lock contention for the LRU head, which is optimized for concurrency but not completely lock free [47, 46]. Instead of a central request counter, AdaptSize hooks into the internal data structure of the cache threads. Each cache thread keeps logging and debug information in a concurrent ring buffer, to which all events are simply appended (overwriting old events after some time). AdaptSize's statistics aggregation accesses this ring buffer (read only) and does not require any synchronization.

3.3.2 Achieving both efficiency and high numerical accuracy.

The OHR prediction in our statistical model involves two implementation challenges. The first challenge lies in efficiently solving the equation in Lemma 1. We achieve a constant time overhead by using a domain-specific algorithm [31], which can be evaluated in a fixed number of steps. The second challenge is due to the exponential function in the important expression Theorem 1. The value of exponential function outgrows even 128-bit number representations. We solve this problem by using an accurate and efficient approximation for the exponential function using a Padé approximant [65] that uses only addition and multiplication operations, allowing the use of SSE3 vectorization on a wide variety of platforms, speeding up the model evaluation by about $10\times$ in our experiment.

4 Evaluation Methodology

We evaluate AdaptSize using *both* trace-based simulations (Section 4.2) and an actual Varnish-based implementation (Section 4.3) running on our experimental testbed. For both these approaches, the request load is derived from traces from Akamai's production CDN servers (Section 5).

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

Table 2: Basic information about our web traces.

4.1 Production CDN request traces

We collected request traces from two production CDN servers in Akamai's global network. Table 2 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. The servers each have a HOC of size 1.2 GB and serve a traffic mix of several thousand popular web sites. The traces represent a typical cross section of the web (news, social networks, downloads, ecommerce, etc.) with highly variable object sizes.

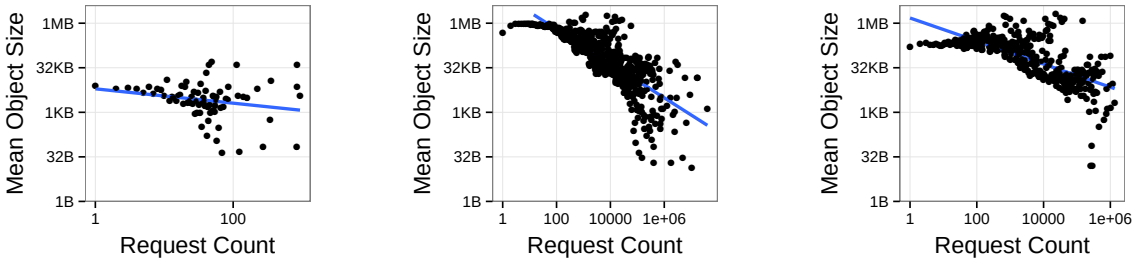
The cumulative distribution function for request sizes is shown in Figure 4 and discussed in Section 3. We find that sizes are highly variable and span ten orders of magnitude. Large objects can therefore replace many small objects, which motivates the design of AdaptSize. Another aspect are potential correlations between popularity and object size.

Previous works have studied this correlation and found it to be negligible [12]. Additionally, [9] shows that size-popularity correlation in real request traces diminished from 1995 to 1999. More recent measurement studies – between 1999 and 2013 – have not studied this question [39, 38]. In our 2015 traces, we find that popular objects are much more likely to have a small size. This bias is strongly evident in Figure 9, which presents the mean object size versus request counts. In particular, the mean request size for the HK trace is 335 Kb, while the mean size of highly popular objects (more than 200,000 requests) is only 8.6 Kb, which is 40 times smaller. Results are similar for the US trace.

This bias further validates the idea of size-aware admission policies.

4.2 Trace-based simulator

We implemented a cache simulator in C++ that incorporates AdaptSize and several of the research-based 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 cache size is enforced by a simple check on the sum of bytes currently stored. While actual caching systems (such as Varnish [47, 46]) use multi-threaded concurrent implementations, our simulator provides a good approximation of the OHR when compared with our prototype implementations that we describe next.



(a) 1998 web trace from [9] (b) 2015 HK trace (c) 2015 US trace

Figure 9: The correlation between request count and object size was weak in 1998 [9, 12]; but is strong in 2015.

4.3 Prototype Evaluation Testbed

Our testbed consists of a client server, an origin server, and a CDN server that incorporates the HOC. We use a dedicated (university) data center to perform our experiments. The experimental servers are FUJITSU Primergy CX250 HPC servers with each two Intel E5-2670 CPUs, 32 GB RAM, and an InfiniBand QDR networking interface. The OS is RHEL 6.5, kernel 2.6.32, gcc version 4.4.7.

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[3] as described in Section 3.3. Since our goal is to improve the OHR of the HOC, our evaluation focuses on the HOC. We did not explicitly implement a disk cache (DC) that typically serves as a second-level cache below the HOC, as the OHR of HOC is not influenced by the DC. However, we do outline the potential impact of AdaptSize on second-level disk caching in Section 6. 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 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 and there are up to 200 such threads. We verified that the number of client threads has a negligible impact on the OHR.

If the CDN server's HOC does not have the requested content, the content is fetched from the origin server and cached (if necessary), before it is served out to the client. The origin server is implemented using FastCGI. Storing the unique objects of the request traces is infeasible because we would need about 8-15TB. Therefore, the origin server stores only a list of object ids and sizes and creates objects with the correct size on the fly, which are then sent over the network. In order to stress test our caching implementation, the origin server is highly multi-threaded and intentionally never the bottleneck.

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

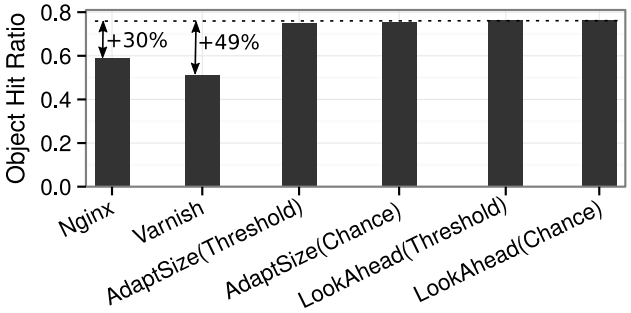


Figure 10: Comparison of AdaptSize's implementation to state-of-the-art production systems. AdaptSize improves the OHR by 30-49% and is within within 95% of LookAhead, which has future knowledge. These results are for the HK trace and a typical HOC size of 1.2 GB.The corresponding results for the US trace are shown in Figure 1.

5 Evaluation of AdaptSize

In this section we present the evaluation of AdaptSize, using the setup described in Section 4. We divide our evaluation into three parts. In Section 5.1, we compare AdaptSize with production caching systems, as well as with a caching system that has future knowledge. In Section 5.2, we compare AdaptSize to research-based caching systems that use more elaborate eviction and admission policies. In Section 5.3, we challenge AdaptSize by emulating flash crowds involving abrupt popularity changes.

5.1 Comparison with production systems

We use our experimental testbed outlined in Section 4.3 and answer five basic questions about AdaptSize.

5.1.1 What is AdaptSize's OHR improvement over production systems?

Quick answer: *AdaptSize improves by 30% over Nginx and by almost 50% over Varnish with respect to the OHR.*

We compare the OHR of AdaptSize(Threshold) and AdaptSize(Chance) to the two state-of-the-art caching systems Nginx and Varnish. We use a 1.2 GB HOC size that is representative of the CDN servers from which our traces are derived (Section 4.1). Figure 10 shows the OHR on the HK trace, and Figure 1 shows the OHR on the US trace. For the HK trace (Figure 10), we find that AdaptSize improves over Nginx by 30% and over Varnish by 49%. For the US trace (Figure 1), the improvement increases to 48% over Nginx and 92% 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 $1.9\times$ on the HK trace and by $3.8\times$ on the US trace (compared to Nginx and Varnish).

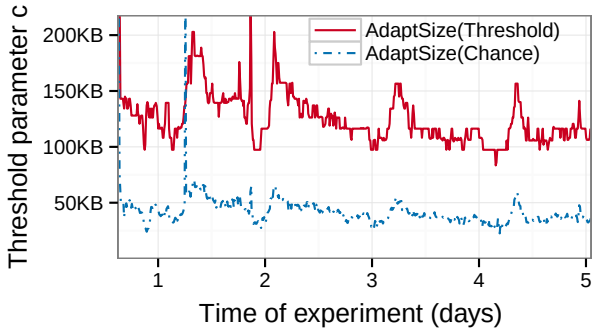


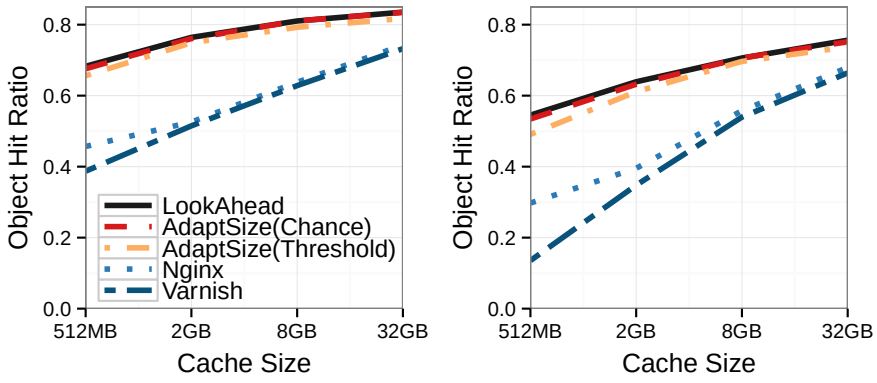
Figure 11: AdaptSize achieves its high performance by continuously adapting the size threshold parameter to the request traffic (2500 adaptions).

5.1.2 How does AdaptSize compare with offline-parameter tuning?

Quick answer: *for the typical HOC size, AdaptSize achieves an OHR within 94% of a LookAhead policy, which has future knowledge of one million requests*. To assess whether AdaptSize leaves room for further OHR improvements, we want to compare AdaptSize's OHR to the offline-optimal OHR. Unfortunately, calculating the offline-optimal OHR is NP hard, which makes it infeasible for traces with millions of requests [17]. Instead, we benchmark AdaptSize against a LookAhead policy, which tunes the threshold parameter c using a priori future knowledge of the next one million requests. Figure 10 compares the OHR of AdaptSize(Threshold) and AdaptSize(Chance) to the corresponding LookAhead policies for the HK trace, and Figure 1 shows these results on the US trace. On the HK trace (Figure 10), AdaptSize(Threshold) achieves 98% of the OHR of LookAhead(Threshold), and AdaptSize(Chance) achieves 99% of LookAhead(Chance). For the US trace (Figure 1), AdaptSize(Threshold) achieves 95% of the OHR of LookAhead(Threshold), and AdaptSize(Chance) achieves 94% of LookAhead(Chance).

5.1.3 How much does AdaptSize's threshold parameter vary?

Quick answer: *AdaptSize's threshold can vary by more than 100% within a single day*. As described in Section 3.2, AdaptSize tunes its threshold parameter c over time. Figure 11 shows AdaptSize's choice of c over the course of the 5 day-long experiment on the US trace. We find that AdaptSize(Threshold) and AdaptSize(Chance) make about 2000-2500 parameter adaptions, and that the decisions roughly mirror each other. The two policies are different in that AdaptSize(Threshold) uses a higher c than AdaptSize(Chance) on average (115 KB vs 40 KB). The reason for this difference is that with AdaptSize(Chance), large objects (with a size above c) still have a small probability to get into the cache. This means that AdaptSize(Chance) can more aggressively optimize its c to lower values. For example, consider the case where most popular objects have a size below 40 KB, but there is one very popular object at 115 KB. AdaptSize(Chance) can choose $c = 40KB$ because the 115 KB object still has a 5% chance of being admitted (which happens eventually because the 115 KB object is popular). AdaptSize(Threshold) needs to choose $c = 115KB$ because otherwise the 115 KB object would never get admitted, which would lead to a



(a) HK trace (b) US trace

Figure 12: Object hit ratio of AdaptSize in comparison to LookAhead and Varnish and Nginx on the two Akamai production traces. AdaptSize improves the OHR across a wide range of HOC sizes. For the typical HOC size of 1.2 GB, AdaptSize improves by 40-92% over Nginx and Varnish, while achieving 94-99% of the offline LookAhead policy OHR.

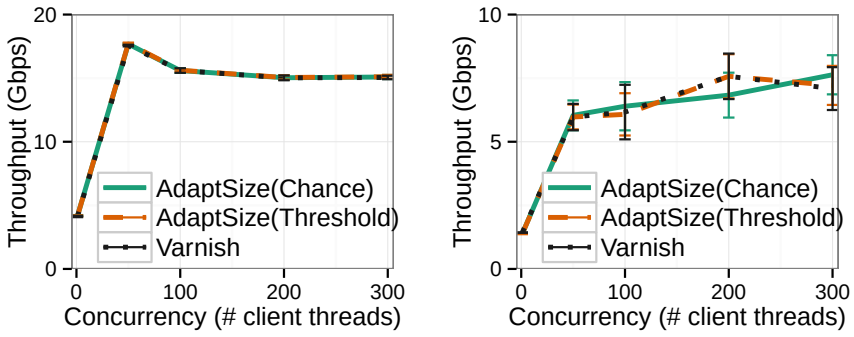
low OHR. AdaptSize continuously evaluates such choices using the statistical model (Section 3.2), which causes the changes of c shown in Figure 11.

5.1.4 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 (512 MB), but decreases for larger HOC sizes (32 GB).* Our results thus far have assumed a HOC size of 1.2 GB. We now consider the effect of both smaller (512 MB) and larger (32GB) HOC sizes. Figures 12a and 12b, the performance of AdaptSize(Chance) and AdaptSize(Threshold) is very close (within 3-10%), and always very close to the LookAhead policy. The improvement of AdaptSize over Nginx and Varnish is most pronounced for the smaller HOC size (512 MB), where AdaptSize improves by 48-80% over Nginx and 75%-395% over Varnish. 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 over the others. Nonetheless, even for a HOC of 32GB, the improvement of AdaptSize over Nginx is 10-12% and over Varnish is 13-14%.

5.1.5 How does AdaptSize's throughput compare with other production caching systems?

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 13, we compare the throughput (bytes per second of satisfied requests) of AdaptSize to an unmodified Varnish system. We use two micro experiments, one that benchmarks the hit request path (100% OHR scenario), and one that benchmarks the miss request path (0% OHR scenario). We expect no overhead in the 100% OHR scenario, because AdaptSize's



(a) 100% OHR scenario.

(b) 0% OHR scenario.

Figure 13: 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).

request statistics are gathered without overhead (i.e., there are no changes to the hit request path). We expect a small overhead in the 0% scenario, because every miss requires the evaluation of the admission decision. We replay one million requests on dedicated servers, and configure different concurrency levels via the number of client threads. Note that the number of client threads does not represent individual users (Section 4.3). The results are based on 50 repetitions.

Figure 13a shows that the application throughput AdaptSize(Chance), AdaptSize(Threshold), and Varnish are the same in the 100% OHR scenario. All three systems achieve a peak throughput of 17.5 Gbps for 50 concurrent clients Due to lock contention, the throughput decreases to around 15 Gbps for 100-300 clients threads for both AdaptSize and Varnish. Figure 13b shows that the application throughput of the three systems in the 0% OHR scenario is very close, and always within the 95% confidence interval.

The memory overhead of AdaptSize is small. Almost all of the memory overhead is caused by the request statistics for the statistical 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, was 1.5 million objects (60 MB), which also agrees with the high water mark (on Linux: VmHWM) for the tuning thread.

5.2 Comparison with research caching systems

In the prior section we saw that AdaptSize performed very well against production cache systems. The purpose of this section is to address the following question.

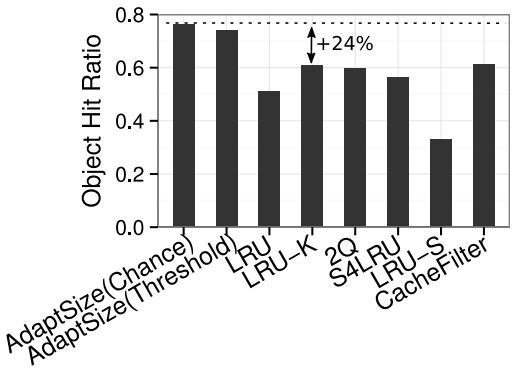


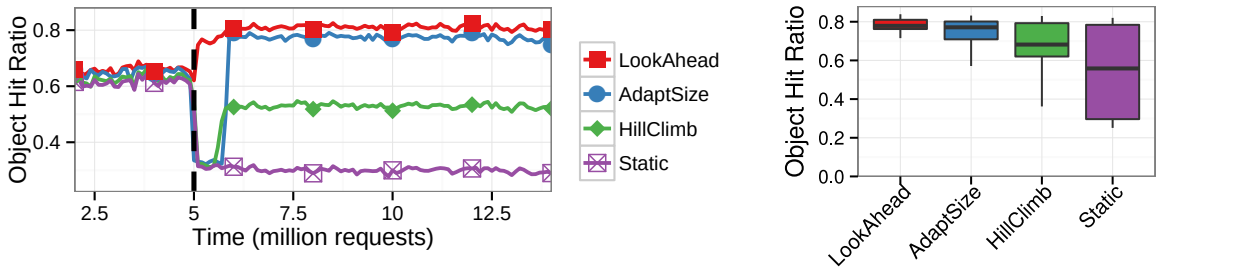
Figure 14: Comparison of AdaptSize to other efficient caching systems from Table 1 on the HK trace. AdaptSize outperforms the best of the other systems by 24%.

5.2.1 How does AdaptSize compare with research-based caching systems from the literature, which involve more sophisticated admission and eviction policies?

Quick answer: *AdaptSize improves by 24-31% over the next best research-based caching system*. We use the simulation evaluation setup explained in Section 4.2 with five caching systems from Table 1, which are selected with the criteria of 1) having an efficient constant-time implementation, and 2) covering representative algorithm from different groups of caching policies. Specifically, we pick LRU-K [60], 2Q [45], S4LRU [38], LRU-S [74], and CacheFilter [52]. LRU-K represents policies with recency and frequency trade-offs, of which ARC [56] is an efficient and self-tuning variant[4]. We swept across 32 configurations of LRU-K to find the optimal recency-frequency trade-off, which is different on both of our traces. 2Q is a parameterless variant of LRU-K that additionally uses frequency-based admission. S4LRU represents the class of recent cache policies [34], which partition the cache space into segments. We swept across 100 permutations of S4LRU's partition sizes to find the optimal partitioning. LRU-S is the only parameterless size-aware eviction and admission policy, which has a constant-time implementation. CacheFilter use frequency-based admission, which we tuned for 32 values of its parameter.

Figure 14 shows the simulation results for a HOC of size 1.2 GB on the HK trace. We find that AdaptSize achieves a 24% higher OHR than the second best system. Figure 14 shows the simulation results the US trace. AdaptSize achieves a 31% higher OHR than the second best system. The second best systems are LRU-K, S4LRU, and CacheFilter, which all rely on offline parameters. 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.

[4]Unfortunately, we could not test ARC itself, because its learning rule relies on the assumption of uniform object sizes.



(a) Single flash crowd event.

(b) 50 randomized flash crowd events.

Figure 15: Comparison of the OHR of cache tuning methods under flash crowd events, which involve a small subset of object abruptly becoming popular (after five million requests). (a) OHR over time for a single flash crowd event. AdaptSize consistently performs close to LookAhead, whereas HillClimb and Static perform badly on this flash crowd event. (b) OHR box plot across 50 randomized flash crowds. The boxes show the range of OHR from the 25-percentile to the 75-percentile among the 50 experiments. The whiskers show the 5-percentile to the 95-percentile.

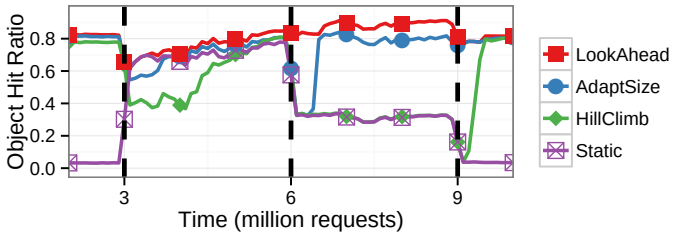
5.3 Comparison with alternative tuning methods

So far we have seen that AdaptSize is vastly superior to caching systems that do not use size-aware admission, including production caching systems (Section 5.1) and research caching systems (Section 5.2). Since we already verified AdaptSize's superiority over systems without size-aware admission control, we now focus on different cache tuning methods for the threshold parameter c. Specifically, we compare AdaptSize with hill climbing (**HillClimb**), based on shadow caches. HillClimb uses two shadow caches to sample the slope of the OHR curve and moves towards the better threshold parameter. We optimized HillClimb's parameters (interval of climbing steps, step size) on our production traces. We also compare to a fixed value of c (**Static**), which is offline optimized on our production traces. We also compare to LookAhead, which tunes c based on offline knowledge of the next one million requests. To simplify the experiment, AdaptSize, HillClimb and LookAhead tune the AdaptSize(Chance) admission policy, and Static tunes the AdaptSize(Threshold) admission policy. All four policies are implemented on Varnish using the the implementation evaluation setup explained in Section 4.3.

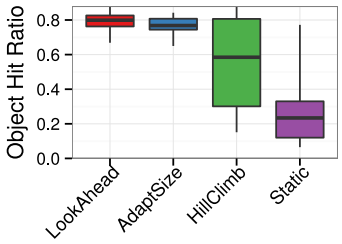
We consider two scenarios: 1) randomized flash crowds, in which random objects suddenly become extremely popular, and 2) changes to the overall traffic mix, which require extreme adjustments to the threshold parameter. We consider each scenario in turn.

5.3.1 Is AdaptSize robust against randomized flash crowd events?

Quick answer: *AdaptSize performs within 81% of LookAhead even for the worst 5% of experiments, whereas HillClimb and Static achieve only 40-50% of LookAhead's OHR*. A common challenge in content delivery is that a small set of items becomes very popular (flash crowd). Size-aware admission can lead to a low OHR, if the size-aware admission policy is configured with a threshold parameter below the flash crowd objects. In some cases, the flash crowd objects can require adjusting



(a) Single flash crowd event.



(b) 50 randomized flash crowd events.

Figure 16: Comparison of the OHR of cache tuning methods under frequent traffic mix changes (every three million requests). Each traffic mix requires a vastly different threshold parameter. (a) OHR over time for a single flash crowd event. AdaptSize consistently performs close to LookAhead, whereas the performance of HillClimb and Static varies between traffic mixes. (b) OHR box plot across 25 experiments involving frequent traffic mix changes. The boxes show the range of OHR from the 25-percentile to the 75-percentile among the 50 experiments. The whiskers show the 5-percentile to the 95-percentile.

the threshold parameter by an order of magnitude (e.g., when a video with 2 MB chunks abruptly becomes popular).

We first explain how we emulate a single flash crowd event. The flash crowd is a random number of objects (between 200 and 1000), which are randomly sampled from the trace. Every caching system runs on two trace parts: first on an unchanged production trace, and second on a trace with the flash crowd. The first part is five million requests long, and allows each caching system to converge to a stable configuration. The second part is ten million requests long, and consists of 50% production-trace requests and 50% flash crowd requests. Figure 15a shows the OHR over time for a typical example where the optimal c changes from 20 KB to 512 KB. Before five million requests, all systems have a near-optimal OHR, which shows that all systems have converged and that the offline parameters of HillClimb and Static are optimally chosen. After five million requests, LookAhead immediately admits the flash crowd objects and consistently achieves an OHR around 0.8. AdaptSize reacts shortly after, adapting c to around $550KB$ and achieves an OHR around 0.77. HillClimb is able to admit some of the flash crowd objects but gets stuck in a local optimum for c around 64 KB. HillClimb achieves an OHR around 0.53. Static (by definition) does not adjust c, and achieves an OHR around 0.30.

We create 50 different flash crowd event experiments. Figure 15b shows a boxplot of the OHR for each caching tuning method across all 50 experiments. The boxes indicate the 25-percentile and 75-percentile, the whiskers indicate the 5-percentile and 95-percentile. As shown in Figure 15b, AdaptSize is superior compared to HillClimb and Static across every method – the median OHR, the worst 25% OHR and the worst 5% OHR.

5.3.2 Is AdaptSize robust against frequent traffic mix changes?

Quick answer: *AdaptSize performs within 81% of LookAhead even for the worst 5% of experiments, whereas HillClimb and Static achieve only 10-23% of LookAhead's OHR*. Another common challenge in content delivery is that a CDN server gets repurposed to serve a different traffic mix. Typical traffic mixes are web traffic, videos, and software downloads. Each of these traffic mixes has a significantly request size distribution, which requires very different threshold parameters.

We challenge the tuning method by changing the traffic mix very frequently (every three million requests), and by requiring extreme adaptions to the threshold parameter. Figure 16a shows the OHR over time for each of the caching systems for one example scenario. The optimal threshold in this scenario changes from about 1 MB, to 32 KB (at time 3 million request), and then to 16 MB (at time 6 million requests), back to 1 MB (at time 9 million requests). In this example, LookAhead achieves an OHR of 0.81. AdaptSize achieves an OHR of 0.74, and typically reacts within 500 thousand requests to adjust its threshold parameter. HillClimb achieves an OHR of 0.5, because it generally adapts more slowly (e.g., between 3 and 6 million requests), and does not adjust to the extreme 16 MB threshold (between 6 and 9 million requests). Static achieves an OHR of 0.38, because it can only be tuned to one of the three thresholds (1MB).

We create 25 different traffic mix changes, where each scenario is 60 million requests long and requires twenty adjustments to the threshold parameter. Figure 16b shows a boxplot of the OHR for each caching tuning method across all experiments. As can be seen, AdaptSize is very close to LookAhead across all percentiles and vastly superior to HillClimb and Static.

6 Conclusions and Implications

This work proposes a new caching system, AdaptSize, for the hot object cache (HOC) in a CDN server. AdaptSize is the first system to implement a size-aware admission policy that continuously optimizes its size threshold parameter over time. The power of AdaptSize stems from a new statistical model that predicts the optimal threshold parameter based on aggregated request statistics.

We build AdaptSize on top of Varnish and we evaluate it on extensive Akamai production CDN traces. We find that AdaptSize vastly improves the OHR over state-of-the-art production systems by 30-90% while maintaining a comparable throughput. We also shows that AdaptSize outperforms representative research-based caching policies by 25-30% with respect to the OHR. Finally, we compare AdaptSize's parameter tuning mechanism with two other widely-used tuning methods from the literature, and we find that it is the only tuning method that consistently achieves an OHR close to that of offline parameter tuning.

While this work focuses on improving the OHR for the HOC, a change to a key CDN component such as the HOC requires other considerations. A key issue is the effect on the performance of the second-level disk cache (DC), which is accessed after each cache miss from the HOC. While optimizing the DC is outside the scope of this paper, we outline the intuition on the expected effects of AdaptSize. The DC can be affected in two ways: 1) via its byte hit ratio BHR[5] and 2) via the type

[5]Unlike the HOC, the DC's goal is to optimize the fraction of *bytes* that are served from the DC, which is called BHR.

of requests the disks have to serve. AdaptSize can actually lead to an increase in the BHR, because the DC gets to serve more large objects[6]. With respect to disk requests, we find that AdaptSize leads to a 60% reduction in the number of requests to the DC, but to a 45% increase in the number of bytes the disks need to serve. However, because the DC requests under AdaptSize have a $3\times$ larger size on average, this actually leads to the same or a smaller number of IOPS than for a HOC without an admission policy.

While this brief consideration shows that the DC can be positively affected when deploying AdaptSize, it is clear that conclusive evidence requires further experiments. We plan to achieve this goal by making AdaptSize open source and by working alongside industrial partners in further examining our proposal.

References

[1] Marc Abrams, C. R. Standridge, Ghaleb Abdulla, S. Williams, and Edward A. Fox. Caching Proxies: Limitations and Potentials. In *WWW*. 1995.

[2] Marc Abrams, Charles R Standridge, Ghaleb Abdulla, Edward A Fox, and Stephen Williams. Removal policies in network caches for world-wide web documents. In *ACM SIGCOMM*, pages 293–305, 1996.

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

[4] Alexa. Alexa top sites. `http:///www.alexa.com/topsites` accessed 03/16/16.

[5] George Almasi, Calin Caşcaval, and David A Padua. Calculating stack distances efficiently. In *ACM SIGPLAN Notices*, volume 38, pages 37–43, 2002.

[6] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating content management techniques for web proxy caches. *Performance Evaluation Review*, 27(4):3–11, 2000.

[7] Hyokyung Bahn, Kern Koh, Sam H Noh, and SM Lyul. Efficient replacement of nonuniform objects in web caches. *IEEE Computer*, 35(6):65–73, 2002.

[8] Sorav Bansal and Dharmendra S Modha. Car: Clock with adaptive replacement. In *Usenix FAST*, volume 4, pages 187–200, 2004.

[9] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella. Changes in web client access patterns: Characteristics and caching implications. *World Wide Web*, 2:15–28, 1999.

[6] Specifically, we simulate the BHR of a 1 TB DC with a HOC that either uses no admission policy or AdaptSize. The DC's BHR is 0.78 without HOC admission policy and 0.88 with AdaptSize on the US trace, and 0.53 without HOC admission policy and 0.72 with AdaptSize on the HK trace.

[10] Daniel S. Berger, Philipp Gland, Sahil Singla, and Florin Ciucu. Exact analysis of TTL cache networks. *Perform. Eval.*, 79:2 – 23, 2014. Special Issue: Performance 2014.

[11] 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.

[12] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE INFOCOM*, pages 126–134, 1999.

[13] PJ Burville and JFC Kingman. On a model for storage and search. *Journal of Applied Probability*, pages 697–701, 1973.

[14] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Usenix symposium on internet technologies and systems*, volume 12, pages 193–206, 1997.

[15] Hao Che, Ye Tung, and Zhijun Wang. Hierarchical web caching systems: Modeling, design and experimental results. *IEEE JSAC*, 20:1305–1314, 2002.

[16] Ludmila Cherkasova and Gianfranco Ciardo. Role of aging, frequency, and size in web cache replacement policies. In *High-Performance Computing and Networking*, pages 114–123, 2001.

[17] Marek Chrobak, Gerhard J Woeginger, Kazuhisa Makino, and Haifeng Xu. Caching is hard—even in the fault model. In *Algorithmica*, volume 63, pages 781–794, 2012.

[18] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Dynacache: dynamic cloud caching. In *USENIX HotCloud*, 2015.

[19] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *USENIX NSDI*, pages 379–392, 2016.

[20] CISCO. VNI global IP traffic forecast: The zettabyte era—trends and analysis, May 2015. available `http://goo.gl/wxuvVk` accessed 11/16/15.

[21] Edward G. Coffman and Predrag Jelenković. Performance of the move-to-front algorithm with markov-modulated request sequences. *Operations Research Letters*, 25:109–118, 1999.

[22] Edward Grady Coffman and Peter J Denning. *Operating systems theory*, volume 973. 1973.

[23] Asit Dan and Don Towsley. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *ACM SIGMETRICS*, pages 143–152, 1990.

[24] John Dilley, Bruce M. Maggs, Jay Parikh, Harald Prokop, Ramesh K. Sitaraman, and William E. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[25] Robert P Dobrow and James Allen Fill. The move-to-front rule for self-organizing lists with markov dependent requests. In *Discrete Probability and Algorithms*, pages 57–80. Springer, 1995.

[26] Gil Einziger and Roy Friedman. Tinylfu: A highly efficient cache admission policy. In *IEE Euromicro PDP*, pages 146–153, 2014.

[27] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *USENIX NSDI*, pages 371–384, 2013.

[28] Seyed Kaveh Fayazbakhsh, Yin Lin, Amin Tootoonchian, Ali Ghodsi, Teemu Koponen, Bruce Maggs, KC Ng, Vyas Sekar, and Scott Shenker. Less pain, most of the gain: Incrementally deployable icn. In *ACM SIGCOMM*, volume 43, pages 147–158, 2013.

[29] James Allen Fill and Lars Holst. On the distribution of search cost for the move-to-front rule. *Random Structures & Algorithms*, 8:179–186, 1996.

[30] Philippe Flajolet, Daniele Gardy, and Loÿs Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics*, 39:207–229, 1992.

[31] N Choungmo Fofack, Mostafa Dehghan, Don Towsley, Misha Badov, and Dennis L Goeckel. On the performance of general cache networks. In *VALUETOOLS*, pages 106–113, 2014.

[32] Christine Fricker, Philippe Robert, and James Roberts. A versatile and accurate approximation for lru cache performance. In *ITC*, page 8, 2012.

[33] Massimo Gallo, Bruno Kauffmann, Luca Muscariello, Alain Simonian, and Christian Tanguy. Performance evaluation of the random replacement policy for networks of caches. In *ACM SIGMETRICS/ PERFORMANCE*, pages 395–396, 2012.

[34] Nicolas Gast and Benny Van Houdt. Transient and steady-state regime of a family of list-based cache replacement algorithms. In *ACM SIGMETRICS*, pages 123–136, 2015.

[35] Erol Gelenbe. A unified approach to the evaluation of a class of replacement algorithms. *IEEE Transactions on Computers*, 100:611–618, 1973.

[36] WJ Hendricks. The stationary distribution of an interesting markov chain. *Journal of Applied Probability*, pages 231–233, 1972.

[37] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. Lama: Optimized locality-aware memory allocation for key-value cache. In *USENIX ATC*, pages 57–69, 2015.

[38] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. An analysis of facebook photo caching. In *ACM SOSP*, pages 167–181, 2013.

[39] Sunghwan Ihm and Vivek S Pai. Towards understanding modern web traffic. In *ACM IMC*, pages 295–312, 2011.

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

[41] Predrag R Jelenković and Ana Radovanović. Least-recently-used caching with dependent requests. *Theoretical computer science*, 326:293–327, 2004.

[42] Song Jiang, Feng Chen, and Xiaodong Zhang. Clock-pro: An effective improvement of the clock replacement. In *USENIX ATC*, pages 323–336, 2005.

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

[44] Shudong Jin and Azer Bestavros. Greedydual* web caching algorithm: exploiting the two sources of temporal locality in web request streams. *Computer Communications*, 24:174–183, 2001.

[45] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *VLDB*, pages 439–450, 1994.

[46] Poul-Henning Kamp. Varnish notes from the architect, 2006. available `https://www.varnish-cache.org/docs/trunk/phk/notes.html` accessed 11/16/15.

[47] Poul-Henning Kamp. Varnish lru architecture, June 2007. available `https://www.varnish-cache.org/trac/wiki/ArchitectureLRU` accessed 11/16/15.

[48] W. Frank King. Analysis of demand paging algorithms. In *IFIP Congress (1)*, pages 485–490, 1971.

[49] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies. In *ACM SIGMETRICS*, volume 27, pages 134–143, 1999.

[50] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G Andersen, O Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *ACM ISCA*, pages 476–488, 2015.

[51] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *USENIX NSDI*, pages 429–444, 2014.

[52] Bruce M Maggs and Ramesh K Sitaraman. Algorithmic nuggets in content delivery. *ACM SIGCOMM CCR*, 45:52–66, 2015.

[53] Valentina Martina, Michele Garetto, and Emilio Leonardi. A unified approach to the performance analysis of caching systems. In *IEEE INFOCOM*, 2014.

[54] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[55] John McCabe. On serial files with relocatable records. *Operations Research*, 13:609–618, 1965.

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

[57] Nicolas Niclausse, Zhen Liu, and Philippe Nain. A new efficient caching policy for the world wide web. In *Workshop on Internet Server Performance*, pages 119–128, 1998.

[58] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *USENIX NSDI*, pages 385–398, 2013.

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

[60] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *ACM SIGMOD*, 22(2):297–306, 1993.

[61] Elizabeth J O'neil, Patrick E O'Neil, and Gerhard Weikum. An optimality proof of the lru-k page replacement algorithm. *JACM*, 46:92–112, 1999.

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

[63] Antonis Panagakis, Athanasios Vaios, and Ioannis Stavrakakis. Approximate analysis of lru in the case of short term correlations. *Computer Networks*, 52:1142–1152, 2008.

[64] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. CFLRU: a replacement algorithm for flash memory. In *ACM/IEEE CASES*, pages 234–241, 2006.

[65] Penco Petrov Petrushev and Vasil Atanasov Popov. *Rational approximation of real functions*, volume 28. Cambridge University Press, 2011.

[66] Petr Pošík, Waltraud Huyer, and László Pál. A comparison of global search algorithms for continuous black box optimization. *Evolutionary computation*, 20(4):509–541, 2012.

[67] William H Press. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[68] Konstantinos Psounis, An Zhu, Balaji Prabhakar, and Rajeev Motwani. Modeling correlations in web traces and implications for designing replacement policies. *Computer Networks*, 45:379–398, 2004.

[69] Luigi Rizzo and Lorenzo Vicisano. Replacement policies for a proxy cache. *IEEE/ACM TON*, 8:158–170, 2000.

[70] Eliane R Rodrigues. The performance of the move-to-front scheme under some particular forms of markov requests. *Journal of applied probability*, pages 1089–1102, 1995.

[71] Trausti Saemundsson, Hjortur Bjornsson, Gregory Chockler, and Ymir Vigfusson. Dynamic performance profiling of cloud caches. In *ACM SoCC*, pages 1–14, 2014.

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

[73] Ramesh K. Sitaraman, Mangesh Kasbekar, Woody Lichtenstein, and Manish Jain. Overlay networks: An Akamai perspective. In *Advanced Content Delivery, Streaming, and Cloud Services*. John Wiley & Sons, 2014.

[74] David Starobinski and David Tse. Probabilistic methods for web caching. *Perform. Eval.*, 46:125–137, 2001.

[75] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. Ripq: advanced photo caching on flash for facebook. In *USENIX FAST*, pages 373–386, 2015.

[76] Naoki Tsukada, Ryo Hirade, and Naoto Miyoshi. Fluid limit analysis of FIFO and RR caching for independent reference model. *Perform. Eval.*, 69:403–412, September 2012.

[77] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient mrc construction with shards. In *USENIX FAST*, pages 95–110, 2015.

[78] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, and Andrew Warfield. Characterizing storage workloads with counter stacks. In *USENIX OSDI*, pages 335–349, 2014.

[79] Roland P Wooster and Marc Abrams. Proxy caching that estimates page load delays. *Computer Networks and ISDN Systems*, 29(8):977–986, 1997.

[80] Neal E Young. Online paging against adversarially biased random inputs. *Journal of Algorithms*, 37:218–235, 2000.

[81] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *USENIX ATC*, pages 91–104, 2001.