

TinyLFU: A Highly Efficient Cache Admission Policy

GIL EINZIGER, Nokia Bell Labs

ROY FRIEDMAN, Technion

BEN MANES, Independent

This article proposes to use a *frequency-based cache admission policy* in order to boost the effectiveness of caches subject to skewed access distributions. Given a newly accessed item and an eviction candidate from the cache, our scheme decides, based on the recent access history, whether it is worth admitting the new item into the cache at the expense of the eviction candidate.

This concept is enabled through a novel approximate LFU structure called *TinyLFU*, which maintains an approximate representation of the access frequency of a *large sample* of recently accessed items. TinyLFU is very compact and lightweight as it builds upon Bloom filter theory.

We study the properties of TinyLFU through simulations of both synthetic workloads and multiple real traces from several sources. These simulations demonstrate the performance boost obtained by enhancing various replacement policies with the TinyLFU admission policy. Also, a new combined replacement and eviction policy scheme nicknamed *W-TinyLFU* is presented. *W-TinyLFU* is demonstrated to obtain equal or better hit ratios than other state-of-the-art replacement policies on these traces. It is the only scheme to obtain such good results on all traces.

CCS Concepts: • **Information systems** → **Information storage systems**; **Hierarchical storage management**;

Additional Key Words and Phrases: Caching, eviction policy, admission policy, counting bloom filter, sketches

ACM Reference format:

Gil Einziger, Roy Friedman, and Ben Manes. 2017. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Trans. Storage* 13, 4, Article 35 (November 2017), 31 pages.

<https://doi.org/10.1145/3149371>

1 INTRODUCTION

Caching is one of the most basic and effective methods in computer science for boosting a system's performance in a multitude of domains. It is obtained by keeping a small percentage of the data items in a memory that is faster and/or closer to the application in settings where the entire data domain does not fit into this fast nearby memory. The intuitive reason caching works is that data accesses in many domains of computer science exhibit a considerable degree of "*locality*." A more formal way to capture this "locality" is to characterize the access frequency of all possible data items through a probability distribution and note that in many interesting domains of

Part of this work was supported by the Israeli Ministry of Science and Technology under grant #10886.

Authors' addresses: G. Einziger, Nokia Bell Labs, 16 Atir Yeda St., Kfar Saba 4464321, Israel; email: gilga1983@gmail.com (much of the work was done while this author was with the Technion); R. Friedman, Computer Science Department, Technion, Haifa 32000, Israel; email: roy@cs.technion.ac.il; B. Manes, Independent; email: Ben.Manes@gmail.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

© 2017 ACM 1553-3077/2017/11-ART35 \$15.00

<https://doi.org/10.1145/3149371>

computer science, the probability distribution is highly skewed, meaning that a small number of objects are much more likely to be accessed than other objects. Further, in many workloads, the access pattern and consequently the corresponding probability distribution change over time. This phenomenon is also known as “*time locality*.”

When a data item is accessed, if it already appears in the cache, we say that there is a *cache hit*; otherwise, it is a *cache miss*. The ratio between the number of cache hits and the total number of data accesses is known as the *cache hit ratio*. Hence, if the items kept in the cache correspond to the most frequently accessed items, then the cache is likely to yield a higher hit ratio (Hennessy and Patterson 2012).

Given that cache sizes are often limited, cache designers face the dilemma of how to choose the items that are stored in the cache. In particular, when the memory reserved for the cache becomes full, one needs to decide which items should be *evicted* from the cache. Intuitively, eviction decisions should be done in an efficient manner, in order to avoid a situation in which the computation and space overheads required to answer these questions surpass the benefit of using the cache. The space used by the caching mechanism in order to decide which items should be inserted into the cache and which items should be evicted is called the *metadata* of the cache. In many caching schemes, the time complexity of manipulating the metadata and the size of the metadata are proportional to the number of items stored in the cache.

When the probability distribution of the data access pattern is constant over time, it is easy to show that the *Least Frequently Used* (LFU) yields the highest cache hit ratio (Breslau et al. 1999; Serpanos and Wolf 2000). According to LFU, in a cache of size n items, at each moment the n most frequently used items thus far are kept in the cache. Yet, LFU has two significant limitations. First, known implementations of LFU require maintaining large and complex metadata. Second, in most practical workloads, the access frequency changes radically over time. For example, consider a video caching service; a video clip that is extremely popular on a given day might not be accessed at all a few days later. Hence, there is no point in keeping that item in cache once its popularity has faded just because it was once very popular.

Consequently, various alternatives to LFU have been developed. Many of these include aging mechanisms and/or focus on a limited size *window* of the last W accesses. Such aging is both in order to limit the size of the metadata and in order to adapt the caching and eviction decisions to the more recent popularity of items. A prominent example of such a scheme is called *Window LFU* (WLFU) (Karakostas and Serpanos 2002), as elaborated later. Further, in the vast majority of cases, a newly accessed item is always inserted into the cache and the caching scheme focuses solely on its *eviction policy*, that is, deciding which item should be evicted. This is because maintaining metadata for objects not currently in the cache is deemed impractical.

Let us note that due to the prohibitively high cost of maintaining a frequency histogram for all objects ever encountered, published works that implement the LFU scheme only maintain the frequency histogram w.r.t. items that are in the cache (Podlipnig and Böszörményi 2003). Hence, we distinguish between them by referring to the latter as *In-Memory LFU* and to the previously discussed variant as *Perfect LFU* (PLFU). Since WLFU outperforms In-Memory LFU (Breslau et al. 1999) (at the cost of larger metadata), here we only address WLFU and PLFU.

A popular alternative to LFU that relies on the “time locality” property is known as *Least Recently Used* (LRU) (Hennessy and Patterson 2012), by which the last item accessed is always inserted into the cache and the least recently accessed item is evicted when the cache is full. LRU can be implemented much more efficiently than LFU¹ and automatically adapts to temporal changes in

¹Yet, LRU is still considered too slow for hardware caches and operating system page caching, and therefore in these highly demanding situations, we find approximations of LRU rather than exact LRU implementations.

the data access patterns and to bursts on the workloads. Yet, under many workloads, LRU requires much larger caches than LFU in order to obtain the same hit ratio.

Contributions. The main contribution of this article is in showing the feasibility and effectiveness of augmenting caches with an approximate LFU-based *admission policy*. This consists of the following aspects: First, we present a caching architecture in which an accessed item is only inserted into the cache if an admission policy decides that the cache hit ratio is likely to benefit from replacing it with the *cache victim* (as chosen by the cache's replacement policy). Second, we develop a novel highly space-efficient data structure, nicknamed TinyLFU, that can represent an adaptable approximate LFU statistics for very large access *sample*.² For example, the metadata required to implement TinyLFU, even for a domain of millions of items, can fit in a single memory page and thus can remain pinned in physical memory, allowing for fast manipulation. We show through both a formal analysis and simulations that despite being approximated, the frequency ordering obtained by TinyLFU mimics closely the corresponding ordering that would appear in a true access frequency histogram for these items.

Third, we report on the integration of TinyLFU into the Caffeine high-performance Java cache open-source project (Manes 2016). In particular, this part includes another optimization of TinyLFU called *W-TinyLFU* to better handle bursty workloads. The *W-TinyLFU* scheme places a relatively small *window cache* in front of the main cache. New items are always inserted into the window cache, and TinyLFU serves as an admission filter for inserting removed items from the window cache into the main cache. In addition to improving hit ratios in such workloads, it may also better fit systems in which new items must always be loaded into the cache.

Fourth, we perform an extensive performance study of TinyLFU-enhanced eviction policies and compare them to multiple other cache management policies including state-of-the-art ones like LIRS (Jiang and Zhang 2002) and ARC (Megiddo and Modha 2003, 2006). We exhibit that for skewed workloads, with both synthetic and real-world traces, the TinyLFU admission policy dramatically boosts the performance of several known eviction policies. In the case of static distributions, after applying the TinyLFU admission policy, the difference between the various eviction policies becomes marginal. That is, even the most naive eviction policy yields almost the same performance as true LFU. On the other hand, in the case of real (dynamic) distributions, a smart eviction policy still has an impact, but much less profound than without the admission policy. Further, on the various traces we have tested, *W-TinyLFU*'s hit ratio tops or equals all other cache management policies we have experimented with, and it is the only policy that performed well consistently. Interestingly, for most traces, even the basic (non-*W-TinyLFU*) configuration in which the eviction policy was LRU enhanced with a TinyLFU admission policy obtained the same (best) results as *W-TinyLFU*.

The rest of this article is organized as follows: Related work is discussed in Section 2. We present TinyLFU in Section 3. Section 4 describes the adaptation of TinyLFU into Caffeine (Manes 2016) and our novel *W-TinyLFU* scheme. The performance measurements are reported in Section 5, and we conclude with a discussion in Section 6.

2 RELATED WORK

2.1 Cache Replacement

As indicated in the introduction, while PLFU is an optimal policy when the access distribution is static, the cost of maintaining a complete frequency histogram for all data items ever accessed is

²We use the term “sample” here rather than a “window” to highlight the role it plays in TinyLFU as well as to distinguish it from other references to “windows” in this work.

prohibitively high, and PLFU does not adapt to dynamic changes in the distribution (Karakostas and Serpanos 2002; Arlitt et al. 2000, 1999). Consequently, several alternatives have been suggested.

In-Memory LFU only maintains the access frequency of data items already in the cache, and always inserts the most recently accessed item into the cache. When needed, In-Memory LFU evicts the least frequently accessed item among those that are in the cache (Podlipnig and Böszörmenyi 2003). In-Memory LFU is usually maintained using a heap. The time complexity of managing LFU was thought to be $O(\log(N))$ until recently, when an $O(1)$ construction was shown in Ketan Shah and Matani (2010). Yet, even with this improvement, In-Memory LFU still suffers from slow reaction to changes in the frequency distribution. Also, its performance lags considerably compared to PLFU in static distributions, since it does not maintain any frequency statistic for items that are no longer in the cache (Podlipnig and Böszörmenyi 2003).

Aging was introduced in Arlitt et al. (2000) to improve the ability of In-Memory LFU to react to changes. It is obtained by limiting the maximum frequency count of cached items as well as occasionally dividing the frequency count of cached items by a given factor. Determining when to divide the counters and by how much is tricky and requires fine-tuning (Arlitt et al. 1999).

As mentioned earlier, WLFU only maintains the access frequency for a window of the last W requests (Karakostas and Serpanos 2002). In order to maintain this window, the mechanism needs to keep track of the order of the requests, which adds an overhead. Yet, WLFU adapts much better to dynamic changes in the access distribution than PLFU does, because its frequencies are maintained only w.r.t. the last W accesses.

ARC (Megiddo and Modha 2003, 2006) combines recency and frequency by maintaining meta-data in two LRU lists. The first reflects items that were only accessed once, while the second corresponds to items that were accessed at least twice in the recent history. ARC adjusts itself to the characteristics of the workload by balancing its cache content source from both lists according to their hit ratio. In order to do so, ARC monitors access data for items that were recently evicted from the cache. If items evicted from the LRU list are hit again, then ARC extends this list and behaves more like LRU. If an item that was recently evicted from the second LFU list is hit, then ARC extends the size of the LFU list and behaves more similar to LFU. The technique of monitoring items that were recently evicted from the cache is called *ghost entries* and has become very common in state-of-the-art cache policies.

LIRS (Jiang and Zhang 2002) (*Low Interference Recency Set*) is a page replacement algorithm that attempts to directly predict the next occurrence of an item using a metric named *reuse distance*. To do so, LIRS monitors previous arrival times of many past items both in cache and in ones that were evicted from the cache. Thus, LIRS also maintains a large number of ghost entries.

The *Segmented LRU (SLRU)* (Karedla et al. 1994) policy captures recent popularity by distinguishing between temporally popular items that are accessed at least twice in a short window versus items accessed only once during that period. This distinction is done by managing the cache through two fixed-size LRU segments, a *probation* segment (A1) and a *protected* segment (A2). New items are always inserted into the probation segment. When an item already in the probation segment is being accessed again, it is moved into the protected segment. If the probation segment becomes full, an insertion to it results in evicting (and forgetting) one of its items according to the LRU policy. Similarly, any insertion to a full protected segment results in evicting its LRU victim. However, in this case, the victim is inserted back into the probation segment rather than being completely forgotten. This way, temporally popular items that reach the protected segment will stay longer in the cache than less popular items.

2Q (Johnson and Shasha 1994) is a page replacement policy for operating systems, which manages two FIFO queues, A1 and A2. A1 is split into two subqueues, A1in (*resident*) and A1out (*non-resident*). When an item is first accessed, it is entered into A1in and stays there until it is evicted

under the FIFO policy. However, when an item is being evicted from either A1in or A2, it is inserted into A1out. If an accessed item is neither in A1in nor in A2, but it appears in A1out, then it is inserted into A2.

LRU-K (O'Neil et al. 1993) also combines ideas from LRU and LFU. In this policy, the last K occurrences of each element are remembered. Using this data, LRU-K statistically estimates the momentary frequency of items in order to keep the most frequent pages in memory.

Web caching schemes often take into account also the size of objects. For example, SIZE (Williams et al. 1996) offers to simply remove the largest item first, while same-size items are ordered by LRU. LRU-SP (Cheng and Kambayashi 2000) weighs both the size and the frequency of an item when picking a cache victim.

Greedy-Dual-Frequency-Size (GDFS) (Cherkasova 1998) is a hybrid web caching policy that combines in its decisions the recency of last accesses, the cost of bringing the object to the cache, the object size, and its frequency. It is an improvement of the Greedy Dual algorithm (Young 1991) that only factors size and cost. In Atzmon et al. (2002), latency is also taken into account in workloads when the data is hierarchical; that is, in order to fetch a child item, one must first fetch its predecessor item.

A related approach to ours is introduced in Tewari and Hazelwood (2004), which suggests augmenting known caching algorithms using a *Hot List*. This list indicates what the most popular items are using some decay mechanism. Items in the hot list are given priority over other items in the eviction policy. However, the decision of whether to evict an item in the cache does not depend on the relative frequency of this item versus the frequency of the currently accessed item. Further, an explicit list of n items must be maintained. This method was shown in Tewari and Hazelwood (2004) to somewhat improve the hit ratio of various caching suggestions at the cost of significant metadata overhead.

Briefly, TinyLFU relates to the previous suggestions by being a mechanism that can be used to augment other caching policies by enhancing them with approximate statistics on a large history while providing both quick access time and low metadata overhead.

2.2 Cache Placement in SANs

The work of Ari, Gottwals, and Henze (Ari et al. 2006) studied SAN-based storage systems in which the disks are enhanced with a limited capacity SSD-based cache. They have suggested automatically migrating large chunks of randomly accessed hot data from the disks to the SSD cache, where the migration decision is based on identifying regions whose *access density* is above a given threshold. In particular, the work in Ari et al. (2006) proposed the *Most Recently Frequently Used* (MRFU) criteria to identify such regions, that is, maintaining access frequency counters for randomly accessed regions while periodically aging (or *cooling* in their terminology) the counters by decrementing them by some constant. The aging/cooling period and the aging/cooling constant were left for future exploration. The replacement policy to make room for newly migrated regions was also not explored. Their results exhibit similar or higher cache hit ratios compared to alternatives with much lower migration bandwidth.

2.3 Approximate Counting Architectures

Approximate counting techniques are widely employed in many networking applications. Approximate counting was originally developed in order to maintain per-stream network statistics. These constructions can be appealing to caching as well since they offer very fast updates and consume a compact size.

Sampling methods (Choi et al. 2002; Hu et al. 2008; Estan et al. 2004) offer a small memory footprint but require explicit representation of keys. Also, they usually encounter relatively large

error bounds. We therefore chose not to use them, as the size of the keys in our context is a significant part of the overall costs.

Other methods such as Counter Braids (Lu et al. 2008) reduce the metadata size but require a long decode operation and are therefore not applicable to caching. Another approach is to compress the counterrepresentation itself (Stanojevic 2007; Hu et al. 2010; Tsidon et al. 2012; Einziger et al. 2015). In TinyLFU, we manage to represent the histogram using short counters, and thus these methods do not help us.

Multihash sketches such as *Spectral Bloom Filters* (SBF) (Cohen and Matias 2003) and the *Count Min Sketch* (CM-Sketch) (Cormode and Muthukrishnan 2004) are appealing in our context. As these sketches implicitly associate keys and counters, there is no need to store keys in the frequency histogram. Yet, they are not optimal for our case. In particular, SBF includes a complex implementation aimed at achieving variable-sized counters. Such a complex implementation is not needed for our usage since we only require small counters. The CM-Sketch, on the other hand, offers a simple implementation, but is relatively inaccurate and therefore takes more space than *counting Bloom filters* (Fan et al. 2000).

The *Minimal Increment* scheme, also known as *conservative update*, has been proposed as a way to boost the accuracy of counting Bloom filters when all the increments are positive (Cohen and Matias 2003; Estan and Varghese 2002; Einziger and Friedman 2015a). This technique was successfully used in several fields (Narayanasamy et al. 2003; Raspall and Sallent 2008; Bianchi et al. 2010; Goyal et al. 2010; Bianchi et al. 2011, 2010; Kolaczowski 2007; Goyal and Daumeaé 2011; Goyal et al. 2012).

2.4 Interesting Caching Applications

Data caching can be used as a cloud service (Chockler et al. 2010, 2011). Memcached (Fitzpatrick 2004) allows in-memory caching of database queries and the items associated with them and is widely deployed by many real-life services including Facebook and Wikipedia.

Caching is also employed at the network level of data centers inside network devices. This is realized by using a technique called *in-network caching* (Psaras et al. 2012; Chai et al. 2012). The technique enables content to be named explicitly so network devices can serve it from their caches. In peer-to-peer systems, TinyLFU was also used to expedite the lookup process in Kademia (Einziger et al. 2016), effectively increasing capacity and reducing delays.

TinyLFU can be integrated in all the aforementioned examples. For in-network caching, TinyLFU is appealing since it requires very small memory overhead and can efficiently be implemented in hardware. As for Memcached and cloud cache services, the eviction policy of Fitzpatrick (2004) and Chockler et al. (2011) is a variant of LRU with no admission policy. As we show in this article, adding a TinyLFU-based admission policy to an LRU eviction policy greatly boosts its performance.

3 TINYLFU ARCHITECTURE

3.1 TinyLFU Overview

TinyLFU's architecture is illustrated in Figure 1. Here the cache eviction policy picks a cache victim, while TinyLFU decides if replacing the cache victim with the new item is expected to increase the hit ratio.

To do so, TinyLFU maintains statistics of item frequencies over a sizable recent history. Storing these statistics is considered prohibitively expensive for a practical implementation and therefore TinyLFU approximates them in a highly efficient manner. In the following, we describe the techniques we employ for TinyLFU, some of which are adaptations of known *sketching* techniques

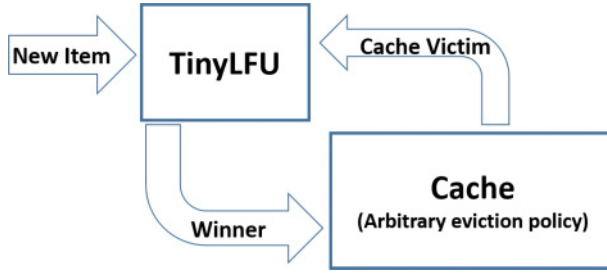


Fig. 1. A general cache augmented with TinyLFU.

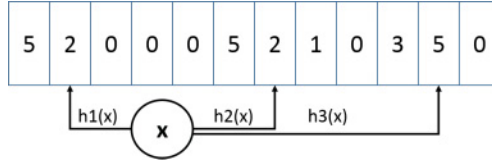


Fig. 2. A counting Bloom filter example.

for approximate counting, whereas others are novel ideas created especially for the context of caching.

Let us emphasize that we face two main challenges. The first is to maintain a freshness mechanism in order to keep the history recent and remove old events. The second is the memory consumption overhead that should significantly be improved upon in order to be considered a practical cache management technique.

3.2 Approximate Counting Overview

A *counting Bloom filter* is a *Bloom filter* in which each entry in the vector is a counter rather than a single bit. Hence, rather than setting bits at the indexes corresponding to the filter's hash functions, these entries are incremented on an insert/add operation.

A *Minimal Increment CBF* (Cohen and Matias 2003; Estan and Varghese 2002) is an augmented counting Bloom filter that supports two methods: Add and Estimate. The Estimate method is performed by calculating k different hash values for the key. Each hash value is treated as an index, and the counter at that index is read. The minimal value of these counters is the returned value. The Add method also calculates k different hash values for the key. However, it reads all k counters and only increments the minimal counters (all of them if there are several minimal counters). If a counter reaches the maximal value it can represent (given its size in bits), additional increments do not increase it any further in order to avoid overflows. For example, suppose we use three hash functions as illustrated in Figure 2. Upon item arrival, three counters are read. Assuming we read $\{2, 2, 5\}$, the Add operation increments only the two left counters from two to three, while the third counter remains untouched. Intuitively, this Add operation prevents unnecessary increments to large counters and yields a better estimation for high-frequency items, as their counters are less likely to be incremented by the majority of low-frequency items. It does not support decrements but is shown to empirically reduce the error for high-frequency counts (Cohen and Matias 2003).

Other Approximate Counting Schemes. As mentioned earlier, CM-Sketch is another popular approximate counting scheme (Cormode and Muthukrishnan 2004). It provides a somewhat lower-accuracy-per-space tradeoff. However, it is largely believed to be faster (Cormode and Muthukrishnan 2004). The space optimizations and aging mechanism we describe next can be

applied to CM-Sketch in the same manner. The description of TinyLFU that follows is oblivious to the choice between counting Bloom filter and CM-Sketch.

3.3 Freshness Mechanism

To the best of our knowledge, keeping approximation sketches fresh has only been studied in Dimitropoulos et al. (2008), where a sliding sample is obtained by maintaining an ordered list of m different sketches. New items are inserted to the first sketch, and after a constant amount of insertions, the last sketch is cleared and is moved to the head of the list. In this way, old events are forgotten.

The method of Dimitropoulos et al. (2008) is not very appealing as a frequency histogram for two reasons: First, in order to evaluate the frequency of an item, m distinct approximation sketches are read, resulting in many memory accesses and hash calculations. Second, this method increases the space consumption, since we need to store the same items in m different sketches and each item is allocated more counters.

Instead, we propose a novel method for keeping the sketch fresh, the *reset* method described later. Every time we add an item to the approximation sketch, we increment a global counter S . Once the value of this counter S reaches the sample size (W), we divide S and all other counters in the approximation sketch by two. Notice that immediately after the reset, $S = W/2$. This division has two interesting merits. First, it does not require much extra space as its only added memory cost is a single counter of $\log(W)$ bits. Second, this method increases the accuracy of high-frequency items as we show both analytically and experimentally. Typically, the accuracy of an approximation sketch can be increased by using more space. We show that the reset method allows increasing sketch accuracy without additional total space cost.

The downside of this operation is an infrequent operation that goes over all the counters in the approximation scheme and divides them by two. Yet, division by two can be implemented efficiently in hardware using shift registers. Similarly, shift and mask operations allow for performing this operation for multiple (small) counters at once. Finally, its amortized complexity is constant, making it feasible for many applications.

In the following subsections, we prove the correctness of the reset operation and evaluate its truncation error. The truncation error is caused since we use an integer division and therefore a counter that shows three will be reset to one and not to 1.5.

3.3.1 Reset Correctness. In this section, we analyze TinyLFU operations under the assumption that the counting infrastructure works accurately. In the result section, we explore how much space is required to keep the performance identical to that of an accurate counting technique.

Definition 3.1. Denote by W the sample size, f_i the frequency of item i , and h_i the estimation of i in TinyLFU.

LEMMA 3.2. *Under constant distribution, at the end of each sample (immediately before each reset operation), the expected frequency of i in the histogram is $E(h_i) = f_i \cdot W$.*

PROOF. By induction on the number of reset operations performed r .

Base: For $r = 0$, the condition holds trivially. We sample W items one after another under constant distribution. By definition, item i has a frequency of f_i . Therefore, the expected height of the histogram is $E(h_i) = f_i \cdot W$.

Step: Assume correctness for $r < j$ and prove it for $r = j$. From the induction hypothesis, until the $j - 1$ reset operation occurs, $E(h_i) = f_i \cdot W$. Therefore, immediately after the $j - 1$ reset operation, $E(h_i) = \frac{f_i \cdot W}{2}$ (at which point $S = \frac{W}{2}$). There are exactly $\frac{W}{2}$ samples until the next reset operation (until $S = W$ again) and therefore $E(h_i)$ is expected to be incremented $\frac{f_i \cdot W}{2}$ times.

Since the expectation is additive, we conclude that right before the j' th reset operation, $E(h_i) = f_i \cdot W$. \square

The previous lemma proves that TinyLFU converges to the right frequency under constant distribution. To show that TinyLFU adjusts to changes, we prove a slightly stronger lemma that says that under constant distribution, eventually TinyLFU converges to the right frequency regardless of its initial value.

LEMMA 3.3. *Under constant distribution, eventually $E(h_i) = f_i \cdot W$.*

PROOF. Consider the initial value of h_i right before a reset operation. We write this value as $f_i \cdot W + \sigma$, where σ is the error. For example, if $h_i = 10$ but $f_i \cdot W = 8$, then $\sigma = 2$. After performing a reset operation, h_i will be $h_i = \frac{f_i \cdot W}{2} + \frac{\sigma}{2}$. The next reset operation is scheduled to happen within $\frac{W}{2}$ events and on average h_i before the next reset operation will increase by $\frac{f_i \cdot W}{2}$. Its new value will therefore be $h_i = \frac{f_i \cdot W}{2} + \frac{\sigma}{2} + \frac{f_i \cdot W}{2} = f_i \cdot W + \frac{\sigma}{2}$. We can repeat this process k times and get that after k more samples, the value of h_i is going to be $h_i = \frac{f_i \cdot W}{2} + \frac{\sigma}{2^k} + \frac{f_i \cdot W}{2} = f_i \cdot W + \frac{\sigma}{2^k}$. Therefore, if we take k to infinity, we get that

$$\lim_{k \rightarrow \infty} \left(f_i \cdot W + \frac{\sigma}{2^k} \right) = f_i \cdot W.$$

In practice, since we use integer division, the initial error is eliminated after $\log_2(\sigma)$ samples. However, in that case, there is also a truncation error as detailed next. \square

3.3.2 Reset Truncation Error. Since our reset operation uses integer division, it introduces truncation error. That is, after a reset operation, the value of a counter can be as much as 0.5 lower than that of a floating-point counter. If we have to reset again, after the reset, the truncation error of the previous reset operation is divided to 0.25, but we accumulated a new truncation error of 0.5, resulting in a total error of 0.75. It is easy to see that the worst-case truncation error converges to at most one point lower than the accurate rate of the item. Therefore, the truncation error affects the recorded occurrence rate of an item by as much as $\frac{2}{W}$ right after a reset operation. This means that the larger the sample size, the smaller the truncation error.

3.4 Space Reduction

Our space reduction is obtained over two separate axes: First, we reduce the size of each of the counters in the approximation sketch. Second, we reduce the total number of counters allocated by the sketch. These space-saving optimizations are detailed next.

3.4.1 Small Counters. Naively implementing an approximation sketch requires using long counters. If a sketch holds W unique requests, it is required to allow counters to count until W (since in principle an item could be accessed W times in such a sample), resulting in $\log(W)$ bits per counter, which can be prohibitively costly. Luckily, the combination of the reset operation and the following observation significantly reduces the size of the counters.

Specifically, a frequency histogram only needs to know whether a potential cache replacement victim that is already in the cache is more popular than the item currently being accessed. However, the frequency histogram need not determine the exact ordering between all items in the cache. Moreover, for a cache of size C , all items whose access frequency is above $1/C$ belong in the cache (under the reasonable assumption that the total number of items being accessed is larger than C). Hence, for a given sample size W , we can safely cap the counters by W/C .



Fig. 3. TinyLFU structure.

Notice that this optimization is possible since our “aging” mechanism is based on the reset operation rather than a sliding window. With a sliding window, in an access pattern in which some item i alternates between W/C consecutive accesses followed by $W/C + 1$ accesses to other items, it could happen that i would be evicted as soon as the sliding window shifts beyond the least recent W/C accesses to i even though i is the most frequently accessed item in the cache.

To get a feel for counter sizes, when $W/C = 8$, the counters require only 3 bits each, as the sample is 8 times larger than the cache itself. For comparison, if we consider a small 2K-item cache with a sample size of 16K items and we do not employ the small counters optimization, then the required counter size is 14 bits.

3.4.2 Doorkeeper. In many workloads, and especially in heavy-tailed workloads, unpopular items account for a considerable portion of all accesses. This phenomenon implies that if we count how many times each unique item in the sample appeared, the majority of the counters are assigned to items that are not likely to appear more than once inside the sample. Hence, to further reduce the size of our counters, we have developed the *Doorkeeper* mechanism that enables us to avoid allocating multiple-bit counters to most *tail* items, that is, to the infrequent items whose counters are likely to remain at most one, which are the vast majority of items in skewed distributions.

The Doorkeeper is a regular Bloom filter placed in front of the approximate counting scheme. Upon item arrival, we first check if the item is contained in the Doorkeeper. If it is not contained in the Doorkeeper (as is expected with first timers and tail items), the item is inserted to the Doorkeeper, and otherwise, it is inserted to the main structure. When querying items, we use both the Doorkeeper and the main structures. That is, if the item is included in the Doorkeeper, TinyLFU estimates the frequency of this item as its estimation in the main structure plus one. Otherwise, TinyLFU returns just the estimation from the main structure.

When performing a reset operation, we clear the Doorkeeper in addition to halving all counters in the main structure. Doing so enables us to remove all the information about first timers. Unfortunately, clearing the Doorkeeper also lowers the estimation of every item by one, which also increases the truncation error by one.

Memorywise, although the Doorkeeper requires additional space, it allows the main structure to be smaller since it limits the amount of unique items that are inserted to the main structure. In particular, most tail items are only allocated 1-bit counters (in the Doorkeeper). Hence, in many skewed workloads, this optimization significantly reduces the memory consumption of TinyLFU. TinyLFU and the Doorkeeper are illustrated in Figure 3. We note that a similar technique was previously suggested in the context of network security (Cao et al. 2009).

	#Unique Items	#2nd Timers	Full Size	Small Size	Average Size (bits)
TinyLFU	7239	416	3	1	1.22
Strawman	8020	-	10	-	10

Fig. 4. TinyLFU versus unoptimized approximate frequency histogram.

3.5 Test Case: TinyLFU Versus a Strawman

To motivate the design of TinyLFU, we compare it to a strawman. The strawman is equivalent to building a frequency histogram using only existing approximate counting suggestions. In order for the strawman to keep items fresh, it uses the sliding-window approximation proposed in Dimitropoulos et al. (2008). That is, the strawman uses 10 different approximate counting sketches in order to mimic a sliding window. Moreover, the strawman does not have a Doorkeeper or a cap on its counters and is therefore required to allow its counters to grow even to the maximal window size.

Our test case is a 1K-item cache augmented by a 9K-item frequency histogram. For this workload, TinyLFU requires its counters to count up to nine. This counting is obtained using 3-bit full counters that can count up to eight, in addition to the Doorkeeper that can count up to one. The strawman uses 10 approximate counting sketches of 900 items each, and its counters are of size 10 bits. In this example, we consider a Zipf 0.9 distribution, which is characteristic of many interesting real-world workloads (e.g., Breslau et al. (1999), Serpanos and Wolf (2000), and Gill et al. (2007)).

We summarize the storage requirements of both frequency histograms in Figure 4. As can be observed, in this workload, TinyLFU is required to store approximately 10% fewer unique values because it uses a single big sketch instead of 10 small sketches. Moreover, for this experiment the vast majority of items consumed only a small counter of 1 bit. The frequent items that appeared more than once in this sample were allocated additional 3-bit counters due to the small counters optimization. In total, for this workload, we notice that TinyLFU reduces the memory consumption of the strawman by $\approx 89\%$.

3.6 Connecting TinyLFU to Caches

Connecting TinyLFU to both LRU and Random caches while treating the cache as a black box is trivial. However, since TinyLFU uses a reset that halves the frequencies of items, we were forced to modify the implementation of the LFU cache in order to properly connect it with TinyLFU. In particular, we had to synchronize the reset operation of TinyLFU with the cache and reset the frequencies of cached items as well.

4 THE W-TINYLFU OPTIMIZATION

4.1 On TinyLFU's Limitations

Despite its benefits, the approach of only adding a TinyLFU admission policy has certain limitations, which motivate the W-TinyLFU architecture presented in Section 4.2. These include:

- (1) TinyLFU was integrated into the open-source Caffeine Java high-performance caching library (Manes 2016). During extensive benchmarking performed with this library, we have discovered that while TinyLFU performs well on traces originating from Internet services and artificial Zipf-like traces, there are a few workloads in which TinyLFU did not perform as well when compared to state-of-the-art caching policies. We explain this issue shortly.
- (2) From a theoretical standpoint, as we show in the appendix, TinyLFU has a worse competitive ratio than LRU.

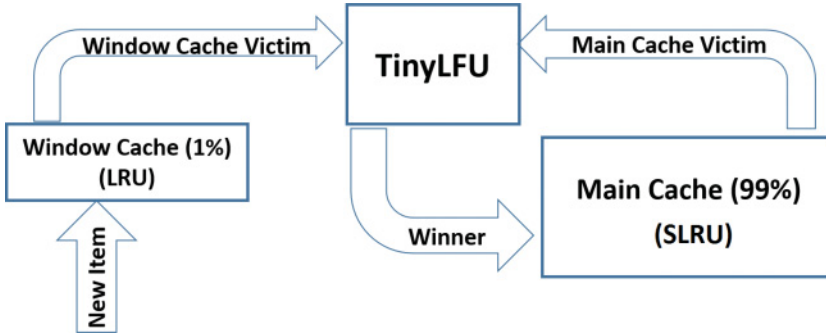


Fig. 5. Window TinyLFU scheme: Items are first always admitted to the window cache and the victim of the window cache is offered to the main cache, which employs TinyLFU as its admission filter. In the particular case of the Caffeine implementation, the window cache is allocated 1% of the total space and it is managed by the LRU policy, while the main cache follows the SLRU eviction policy and is allocated 99% of the total cache space.

- (3) In some systems, missed items always have to be brought into the cache in order to be read. In such systems, the cache needs at least one slot for the most recently accessed item.

To understand limitation 1, let us introduce the concept of *recency bias* in a workload. Consider an access pattern in a workload in which occasionally there is a sequence of accesses such that the vast majority of accesses inside this sequence are to a very small number of objects (in the extreme, all accesses are to a single object). We refer to such a sequence as a *burst of requests*. Note that during a burst, the most recently accessed objects are the most likely to be accessed next regardless of their overall frequency. Hence, when a workload includes many bursts of requests, we say that it has a strong recency bias.

Any kind of cache management policy that is based on frequency, including the TinyLFU admission policy, is likely to yield suboptimal hit ratios on workloads with a strong recency bias. Intuitively, when a burst begins, it may take a frequency-based scheme several accesses until the frequency of the corresponding objects becomes high enough to admit them into the cache. Then, once the burst ends, it would take some time for the aging process of the frequency estimation for the corresponding objects to drop enough to evacuate them. During this time, they occupy space in the cache that cannot be allocated to more relevant objects. In other words, both at the beginning of a burst and after it ends, a frequency-based scheme will underperform compared to an LRU cache. The smaller the cache is and the stronger the recency bias is, the worse the frequency-based scheme will do. Yet, if a burst is very long, then the detrimental impact of its beginning is amortized by its many accesses, meaning that many short bursts are the worst for frequency-based schemes. The workloads where TinyLFU underperformed were these types of traces, which originated from a few storage systems, as reported in Section 5.

4.2 The W-TinyLFU Architecture

The aforementioned issues were remedied by devising an architecture called *Window TinyLFU (W-TinyLFU)*, which consists of two cache areas as illustrated in Figure 5: a *main cache* that employs a TinyLFU admission policy and a *window cache* that has no admission filter. Every time there is a cache miss, the accessed item is inserted into the window cache. The victim of the window cache is then passed to TinyLFU to decide whether it should be inserted into the main cache depending on whether its frequency estimation is higher or lower than the main cache's victim.

In principle, this architecture is generic and one is free to choose any eviction policy for the window cache and for the main cache, as well as to decide on their relative sizes. Yet, the purpose of the window cache in this architecture is to hold recently accessed items, thereby helping the entire scheme handle workloads with a strong recency bias, without hurting the cache's performance in other workloads. As mentioned earlier, workloads with a strong recency bias incur many bursts during which LRU gives the best performance. This is why in this article, as well as in Caffeine's release, we limit ourselves to configurations in which the eviction policy of the window cache is LRU. It also explains why the window cache has no admission policy at all.

As for the main cache, we employ the SLRU eviction policy (Karedla et al. 1994). The A1 and A2 regions of the SLRU policy in the main cache are divided such that 80% of the space is allocated to hot items (A2) and the victim is picked from the 20% nonhot items (A1), as recommended by (Karedla et al. 1994).

In the current release of Caffeine (2.0), the size of the window cache is 1% of the total cache size and that of the main cache is 99%. The motivation behind W-TinyLFU is to have the scheme behave like TinyLFU for LFU workloads while still being able to exploit LRU patterns such as bursts. Because 99% of the cache is allocated to the main cache (with TinyLFU), the performance impact on LFU workloads is negligible. On the other hand, some workloads allow for exploitation of LRU-friendly patterns. In these workloads, W-TinyLFU is better than TinyLFU. As we report in Section 5, for Caffeine's needs, W-TinyLFU is a top alternative for a wider variety of workloads and thus the added complexity is justified.

TinyLFU was added into the Caffeine project after most other schemes were already implemented in its simulator. Our experience doing this exposed several qualitative benefits for TinyLFU compared to other state-of-the-art schemes. As we mentioned previously, the latter typically maintain ghost entries, which adds complexity to the implementations. Further, they involve a higher memory overhead. The overall space overhead of W-TinyLFU in Caffeine is 8 bytes per cache entry.³ This is significantly lower than the overhead of any scheme that requires explicit management of ghost entries, such as ARC and LIRS, since there are usually more ghost entries than elements in the cache. In particular, the identifier of each such entry can be at least 8 bytes plus the space overhead of associated data structures. Also, the development time was faster in the case of TinyLFU (and W-TinyLFU) because the policy is less invasive, as there is no need to rework the main eviction scheme to account for ghost entries polluting the other data structure.

Finally, in the appendix, we show that for any workload, W-TinyLFU always offers a hit ratio that is at least as high as the one obtained by an LRU cache whose size is the same as the window cache of W-TinyLFU, which addresses limitation 2 of TinyLFU. Additionally, limitation 3 is addressed whenever the window cache size is at least one.

5 EXPERIMENTAL RESULTS

5.1 Methodology

In this section, we examine the utility of TinyLFU as a practical cache admission policy. Our evaluation is divided into three parts: First, in Section 5.2, we examine the impact of augmenting three baseline eviction policies, namely, LRU, Random, and LFU, with a TinyLFU admission policy. This part is performed using a Java prototype that can be instantiated with a given cache size (in terms of number of items it can store), a file that contains a trace of accesses, and one of the aforementioned cache management schemes. The prototype then consumes the file, returning for each access whether according to the chosen management scheme and cache size it would have been

³The total metadata size is larger since the LRU policy is maintained as a queue with forward and backward pointers adding two pointers for each in-memory item.

a hit or a miss as well as overall statistics. The prototype used a counting Bloom filter with all optimizations described in Section 1.

Our second set of evaluations, reported in Section 5.3, is performed using the actual Caffeine implementation (Manes 2016). In the current release of Caffeine (2.0), the TinyLFU histogram is maintained for a sample that is 10 times the cache size. Also, Caffeine uses a CM-Sketch with four hash functions and 4 bits per counter. For best performance, each set of 16 counters is stored with a 64-bit alignment. This way, the Reset operation halves 16 counters at once with simple bit manipulation techniques.

We have evaluated TinyLFU both with the generic CBF implementation and with the aforementioned CM-Sketch configuration and could not find any empirically meaningful hit ratio difference. Yet, since the Caffeine CM-Sketch realization is precoded to a specific highly efficient configuration, it is significantly faster than the generic CBF implementation. Consequently, we have used the Caffeine implementation in our comparative study.

We have compared the obtained hit ratios of a large number of cache management policies.⁴ Yet, for brevity, we report only the best-performing ones. In the third set of measurements, reported in Section 5.4, we explore the various types of errors accumulated by TinyLFU's various mechanisms as well as the total error.

For the evaluations, we employ both constant distributions of Zipf 0.7 and Zipf 0.9⁵ and an SPC1-like synthetic trace (Megiddo and Modha 2003) (denoted **SPC1-like** later) as well as real traces. In the synthetic Zipf traces, items are picked according to the corresponding distribution from a set of 1 million objects. Further, caches are given a long warm-up time (20 times the sample size) and we present them at their highest hit ratio (in percentage). The SPC1-like trace contains long sequential scans in addition to random accesses and has a page size of 4Kbytes (Megiddo and Modha 2003). In the real traces, the caches are not warmed up; since the distribution gradually changes over time, no warmup is necessary.

Our real workloads come from various sources and represent multiple types of applications, as detailed here:

- **YouTube**: A YouTube trace is generated from a YouTube dataset (Cheng et al. 2008). Specifically, the trace in Cheng et al. (2008) includes a weekly summary of the number of accesses to each video rather than a continuous timeline of requests. Hence, for each reported week, we have calculated the corresponding approximate access distribution and generated synthetic accesses that follow this distribution on a week-by-week basis. Note that there is no recency information in this trace. Moreover, in some experiments, we varied the number of requests played from each week's distribution as a parameter to demonstrate how quickly algorithms adjust to changes in the workload.
- **Wikipedia**: A Wikipedia trace containing 10% of the traffic to Wikipedia during 2 months starting in September 2007 (Urdaneta et al. 2009).
- **DS1**: A database trace taken from Megiddo and Modha (2003).
- **S3**: A search engine trace taken from Megiddo and Modha (2003).

⁴The Caffeine simulator implements the following schemes: Belady's optimal (Belady 1966), an unbounded cache, LRU, MRU, LFU, MFU, FIFO, Random, TinyLFU, W-TinyLFU, Clock (Corbato 1968), S4LRU (Huang et al. 2013), SLRU (Karedla et al. 1994), MultiQueue (Zhou et al. 2001), Sampled LRU (Psounis and Prabhakar 2002), Sampled MRU (Psounis and Prabhakar 2002), Sampled LFU (Psounis and Prabhakar 2002), Sampled MFU (Psounis and Prabhakar 2002), Sampled FIFO (Psounis and Prabhakar 2002), 2Q (Johnson and Shasha 1994), TuQueue (OpenBSD 2014), LIRS (Jiang and Zhang 2002), Clock-Pro (Jiang et al. 2005), ARC (Megiddo and Modha 2003, 2006), CAR (Bansal and Modha 2004), and CART (Bansal and Modha 2004).

⁵We have experimented with several other skewed distributions and obtained very similar qualitative results.

- **OLTP**: A trace of a file system of an OLTP server (Megiddo and Modha 2003). It is important to note that in a typical OLTP server, most operations are performed on objects already in memory and thus have no direct reflection on disk accesses. Hence, the majority of disk accesses are the results of writes to a transaction log. That is, the trace mostly includes ascending lists of sequential block accesses sprinkled with a few random accesses due to an occasional write replay or in-memory cache misses.
- **P8 and P12**: Two Windows server traces from Megiddo and Modha (2003).
- **Glimpse**: A trace of the Glimpse system describing an execution of an analytic query (Jiang and Zhang 2002). The main characteristic of this trace is an underlying loop, along with other types of accesses.
- **F1 and F2**: Traces from applications running at two large financial institutions, taken from the UMass trace repository (Liberatore and Shenoy 2016). These are fairly similar in structure to the OLTP trace for the same reasons mentioned previously.
- **WS1, WS2, and WS3**: Three additional search engine traces, also taken from the UMass repository (Liberatore and Shenoy 2016).

Notice that the DS1, OLTP, P8, P12, and S3 traces were provided by the authors of ARC (Megiddo and Modha 2003) and represent potential ARC applications. The SPC1-like synthetic trace is also by the authors of ARC (Megiddo and Modha 2003). Similarly, the Glimpse trace is provided by the authors of LIRS (Jiang and Zhang 2002).

We have also used the YouTube workload to measure the impact of the dynamics of the distribution on the performance. To do so, instead of playing the entire amount of views in a week, we only played a random subset from each week. These tests simulate the behavior of the caches when the workload is very dynamic.

In our figures, LRU and Random refer to eviction policies, while TLRU and TRandom refer to LRU and Random caches augmented by TinyLFU, respectively. For LFU cache eviction, we tested two options: WLFU that uses both the LFU eviction policy and LFU admission policy implemented using an accurate sliding window. TLFU is the name we gave an LFU cache augmented with TinyLFU. Further, ARC represents the ARC policy and LIRS represents the LIRS policy, while W-TinyLFU stands for the W-TinyLFU scheme (1% LRU window cache and 99% main cache managed with TinyLFU admission policy and an SLRU eviction policy). In graphs where we explore different window cache sizes for W-TinyLFU, the window size is written in parentheses; for example, W-TinyLFU(20%) represents a window cache that consumes 20% of the total cache size.

5.2 Results of Augmenting Caches with TinyLFU

Figure 6 shows the results of the TinyLFU admission policy on the performance of several eviction policies mentioned earlier under constant skewed distributions. As can be seen, under constant distributions, all caches that are augmented with TinyLFU behave in a similar way. Surprisingly, LFU cache eviction yields only a marginal benefit over the TinyLFU-augmented LRU and Random techniques. Let us note that in such skewed distributions, the maximal cache hit ratio is theoretically bounded regardless of its size. Intuitively, for a distribution function f_i , this bound can be roughly computed by the integral over $\max(0, f_i - 1)$ (since the first occurrence is always a miss) divided by the integral over f_i .

We conclude that for constant skewed distributions, the TinyLFU cache admission policy is an attractive enhancement—for example, the augmented policies with a 1,000-item cache obtain better or similar hit ratios than the corresponding nonaugmented ones with a 4,000-items cache. While augmenting LFU yields a slightly higher hit ratio, the overheads of LFU may justify using a simpler

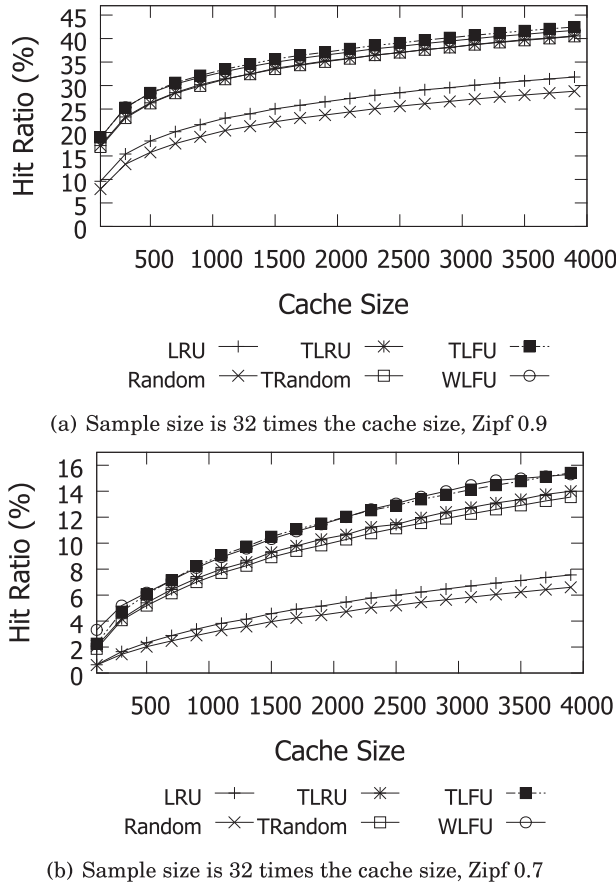


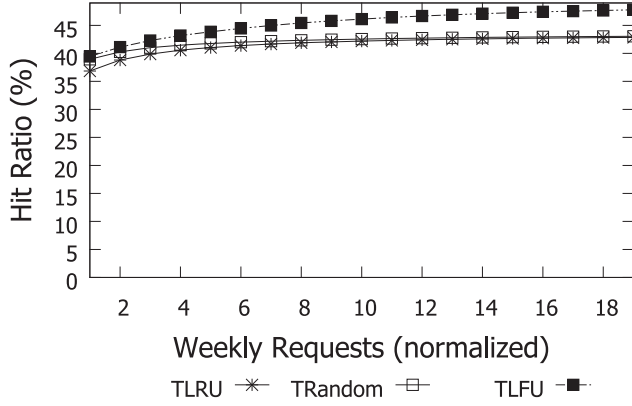
Fig. 6. Augmenting arbitrary caches with a TinyLFU admission policy.

cache eviction policy. Particularly, LRU and even Random offer low overheads with comparable hit ratios.

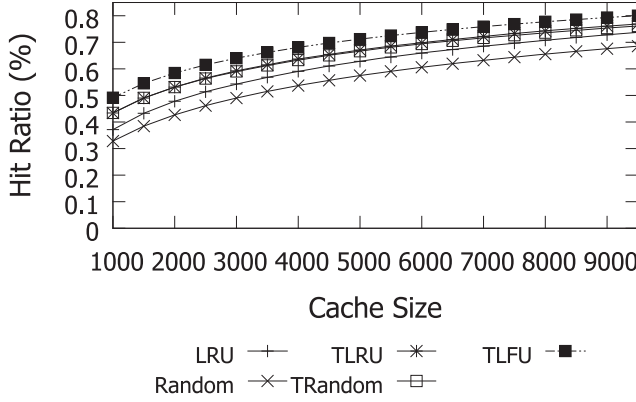
The second experiment we did was testing the augmented caches on a dynamic distribution. To do so, we used a dataset that describes the popularity of 161K newly created YouTube videos over 21 weeks starting on April 16, 2008 (Cheng et al. 2008). We evaluated the approximated frequencies of each of the videos each week and created a distribution that represents each week. Our experiments therefore swap between these distributions every given amount of requests as explained later.

We measured two metrics: The first is how fast we can change the distributions, that is, what is the effect of the number of requests we play from each week's distribution on the hit ratio of our TinyLFU augmented caches. The second is the impact of the cache size on the achieved hit ratio when the distribution change speed is taken from the trace.

The results of this experiment are shown in Figure 7. As can be observed, TinyLFU is effective in augmenting arbitrary caches even in dynamic workloads—for Random it adds roughly 10% to its hit ratio and for LRU it adds about 5%. Further, the augmented policies obtain better hit ratios when the distribution changes more slowly, as expected from all LFU caches. Yet, in this workload, the difference between an augmented Random cache and an augmented LRU cache to a true LFU



(a) Effect of the change speed on the hit-ratio for 1000 items cache and a TinyLFU sample size of 9000



(b) Cache size vs. hit-ratio for TinyLFU sample size of $9 \cdot \text{Cache-Size}$

Fig. 7. YouTube dataset.

cache is more significant than in the static distribution case. Therefore, picking the correct cache victim seems to be more important in dynamic workloads than in static workloads.

The third measurement we made was running the experiment on the Wikipedia access trace. We first studied the required ratio between the TinyLFU sample size and cache size on arbitrary sequences of 100 million consecutive requests from different points in the trace. Second, we used the best ratio we found and tested it on different cache sizes. These results are shown in Figure 8. Unlike static workloads, real-life workloads gradually change over time. Therefore, using a very large TinyLFU sample can even reduce the obtained hit ratio as it slows the pace by which the cache adjusts to the workload.

We note that although WLFU and TLFU achieved nearly identical hit ratios, the main difference between WLFU and TLFU is in their metadata costs. For example, in the YouTube workload, we used only 0.57 bytes per sample element. Since the TinyLFU statistics are maintained for a sample 9 times the size of the cache, the total metadata cost of TinyLFU is $9 \cdot 0.57 = 5.13$ bytes per cache entry. If we consider that each cache entry should contain a video ID that requires 11 bytes,

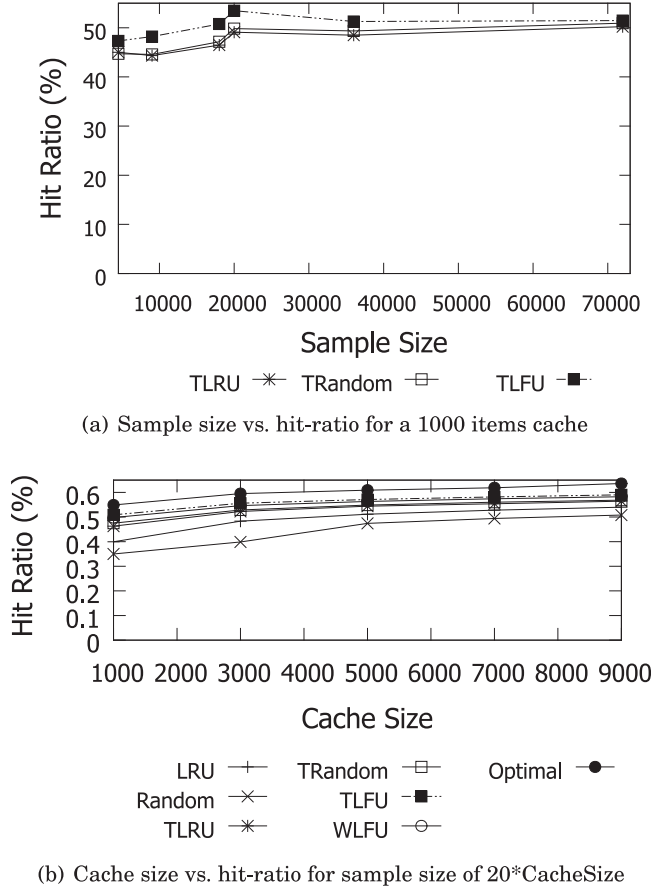


Fig. 8. Evaluation over the Wikipedia trace.

we conclude that TinyLFU is able to approximately remember a history 9 times bigger than the cache with less space overhead than what is required to store just the keys of all cached items.

For comparison, WLFU is required to remember an explicit history 9 times bigger than the cache content. Maintaining this history is expected to cost, even in the most space-efficient implementation, 99 bytes per cache entry. In addition, to operate quickly, it is required to maintain an explicit summary of these items, since iterating over the window and counting the frequency of cached items and replacement candidates is very slow. Even if we neglect this additional memory overhead, WLFU's admission policy still requires almost 20 times more space than TinyLFU.

5.3 Experiments with Caffeine

5.3.1 Comparative Analysis.

Databases. Figures 9 and 10 present the hit ratios of TLRU and W-TinyLFU compared to the state-of-the-art schemes ARC and LIRS on the Glimpse (Jiang and Zhang 2002) and DS1 (Megiddo and Modha 2003) traces. In both traces, TLRU and W-TinyLFU provide very good results. In the Glimpse trace, they are on par with LIRS, while on the DS1 trace, both TLRU and W-TinyLFU outperform all other policies. This indicates that the TinyLFU approach is a viable option for various types of databases.

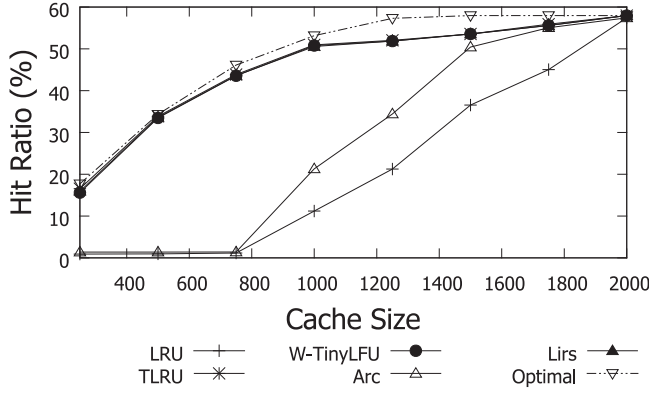


Fig. 9. Glimpse trace.

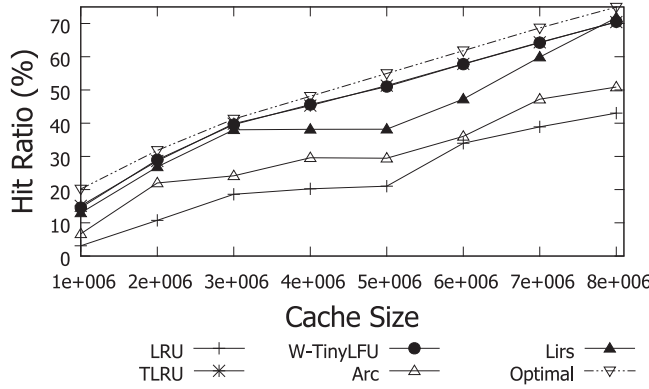


Fig. 10. DS1 database trace.

Windows File System. Figures 11 and 12 show results for the window workstation traces P8 and P12 (Megiddo and Modha 2003). In these traces, we evaluate W-TinyLFU with both 1% window cache and with a 20% window cache. As shown, both achieve very attractive hit ratios and outperform the other policies for all but the largest data points. In Figure 12, the graphs for the 20% window cache and 1% window cache intersect each other twice. This indicates a potential benefit for being able to adaptively change the size of the window cache.

Disk Accesses in OLTP Systems. In the OLTP trace shown in Figure 13, TinyLFU behaves poorly due to the peculiarity of this trace as reported in Section 5.1. That is, it consists of ascending lists of sequential block accesses sprinkled with a few random accesses due to an occasional write replay or in-memory cache misses. Therefore, TLRU is inferior to the rest. Figure 14 and Figure 15 show results for two additional OLTP traces from servers running financial applications (Liberatore and Shenoy 2016), exposing similar phenomena. Yet, in these two last figures, the differences between all schemes are smaller.

As can be seen, in all OLTP traces, W-TinyLFU's performance is comparable with the state of the art. W-TinyLFU outperforms LRU and LIRS in all three traces. Compared to ARC, W-TinyLFU is slightly better in the OLTP and F2 traces, while in the F1 trace, it is only better than ARC in smaller cache sizes with marginal differences between them.

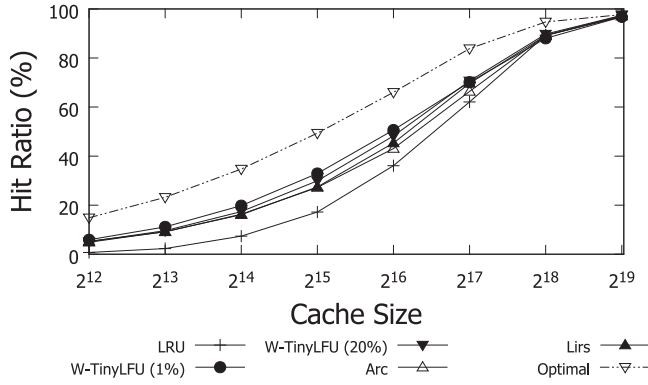


Fig. 11. P8 windows trace.

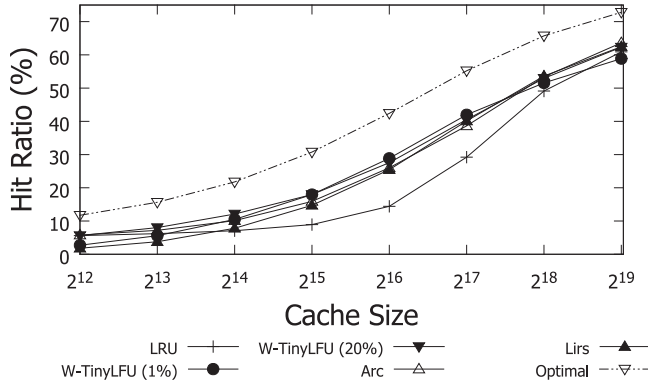


Fig. 12. P12 windows trace.

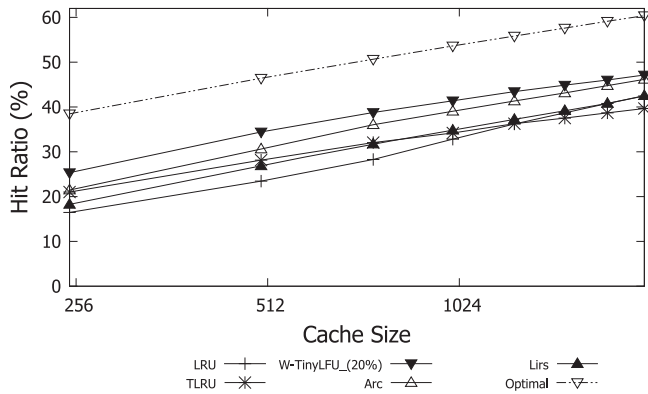


Fig. 13. OLTP trace.

We present here only W-TinyLFU(20%) since it is slightly better than W-TinyLFU(1%) on these traces and doing so helps in distinguishing between the various lines in the graphs. Section 5.3.2 explores the impact of the window cache size in W-TinyLFU on the hit ratio, where it is shown that a 20% window cache is needed for the best results with OLTP traces. Unfortunately, increasing the window cache size reduces W-TinyLFU's performance with the other traces. Hence, in the default

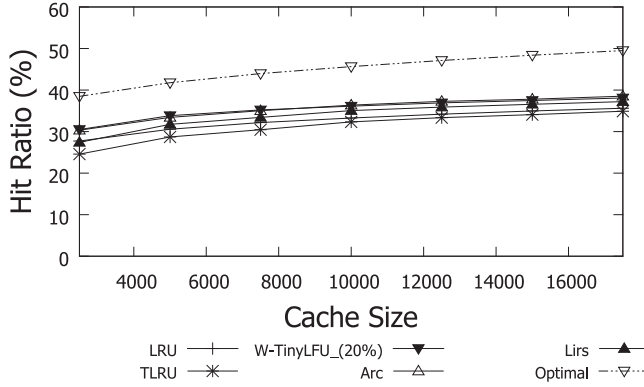


Fig. 14. Financial application (F1) trace.

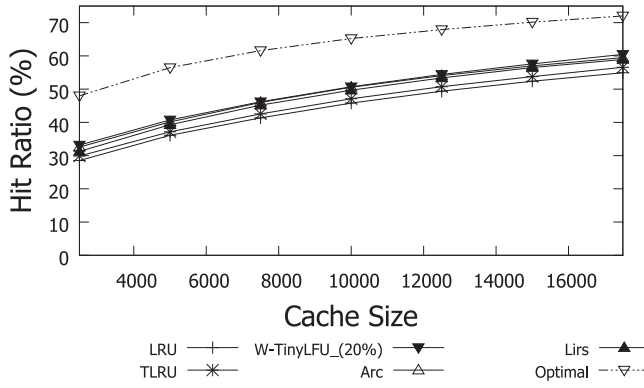


Fig. 15. Financial application (F2) trace.

configuration of the current version of Caffeine (2.0), the window cache size is fixed at 1% of the total cache size in order not to burden the user with extra configuration parameters.

SPC1-like. Figure 16 shows our result on the SPC1-like trace taken from Megiddo and Modha (2003). As can be observed, TLRU and W-TinyLFU outperform all other policies on that trace as well.

Search Engine Queries. Figures 17, 18, 19, and 20 demonstrate the behavior of W-TinyLFU and TLRU with a variety of search engine traces from Megiddo and Modha (2003) and Liberatore and Shenoy (2016). As can be observed, between ARC and LIRS, ARC seems better for small caches and LIRS for large ones. Yet, both TLRU and W-TinyLFU outperform them throughout.

5.3.2 Tuning the Window Cache Size in W-TinyLFU. As reported earlier, for the majority of traces, W-TinyLFU with a window cache whose size is 1% of the total cache outperformed (or tied) all other schemes. In contrast, in OLTP, F1, and F2, TinyLFU typically underperformed, and while a window cache size of 1% improved the hit ratio, it was still not the top performer. In these traces, a larger window cache is needed to better handle their recency bias, as mentioned previously. Figure 21 explores the impact of the cache allocation between the window cache and the main cache on the hit ratio of W-TinyLFU under these three traces: OLTP, F1, and F2. For OLTP we tested a total cache size of 1,000 items, while for F1 and F2 the total cache size is 2,500 items. As can be observed, window cache sizes of 20% to 40% seem to perform best on these traces since,

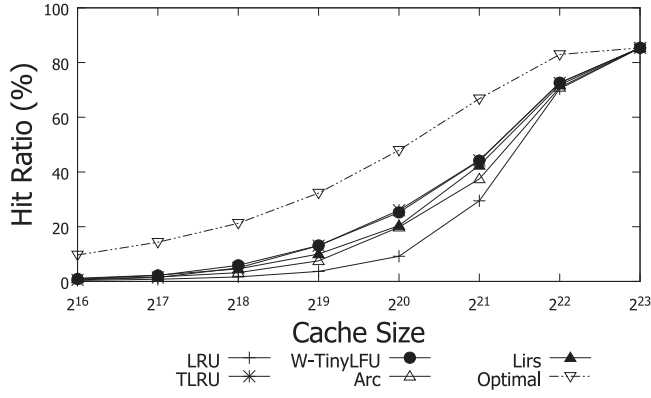


Fig. 16. SPC1-like trace.

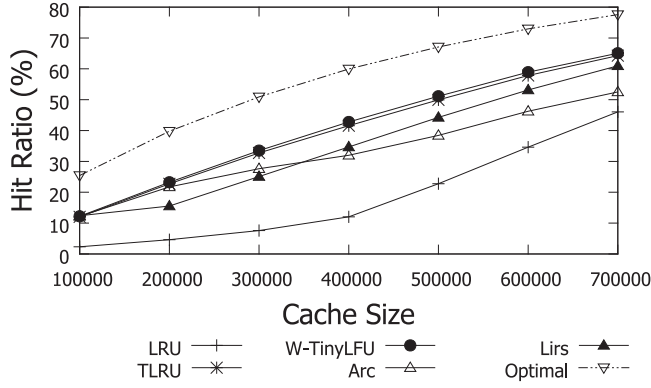


Fig. 17. Search engine trace (S3).

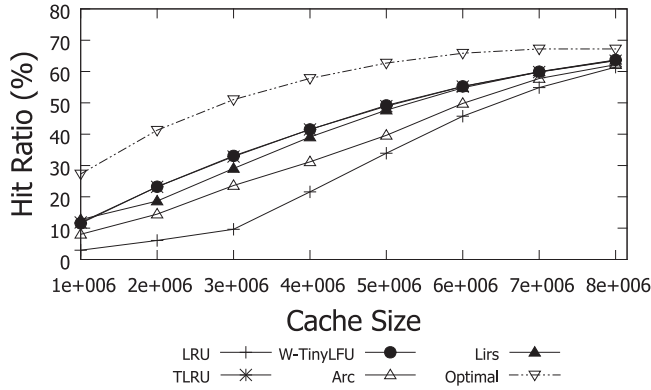


Fig. 18. Search engine trace (WS1).

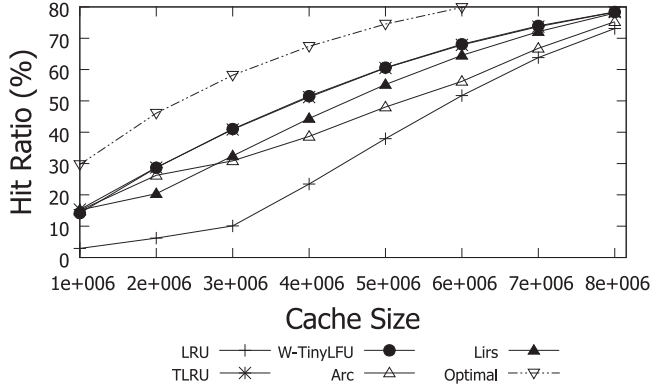


Fig. 19. Search engine trace (WS2).

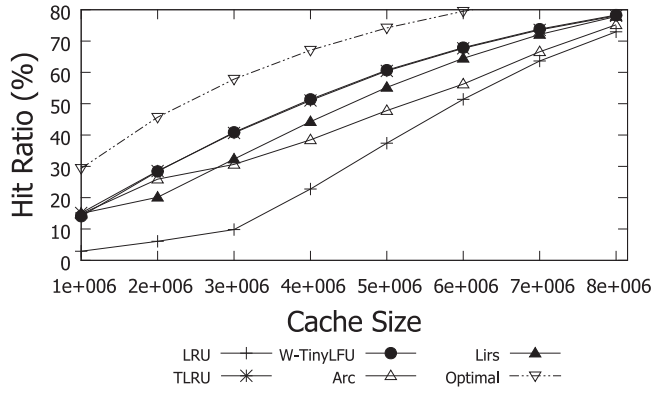


Fig. 20. Search engine trace (WS3).

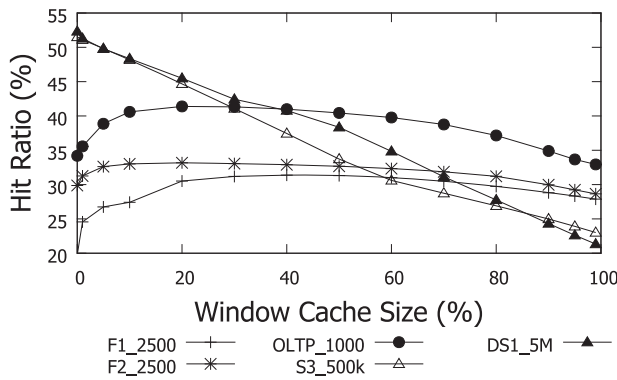


Fig. 21. Window/main cache balance.

as discussed earlier, a large window cache enables better handling of significant recency bias in the workload. In contrast, we can see that for the DS1 and S3 traces, the hit ratio degrades almost linearly as the window cache size is increased, since these traces exhibit little or no recency bias. Automatic dynamic tuning of the window size is left for future work.

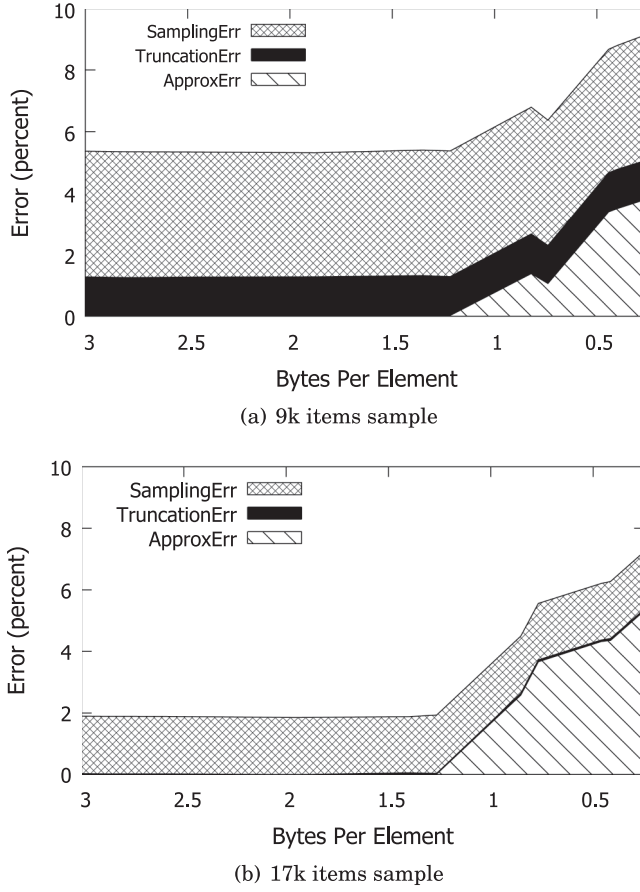


Fig. 22. Different errors for 1K items cache under Zipf 0.9 distribution, for different sample sizes.

5.4 Sources of Inaccuracy

The most obvious source of inaccuracy in TinyLFU is the approximation error introduced by false positives. However, there is also the truncation error that is caused from performing integer division rather than a real (floating-point) division in the reset operation. For static distributions it is also feasible to measure the sampling error, that is, how well TinyLFU was able to understand the underlining distributions. All these factors together can explain the (in)accuracy of TinyLFU.

We define the *Sampling Error* to be the difference between the hit ratio of an accurate TinyLFU with floating-point counters and the ideal hit ratio achievable under a constant distribution. The sampling error can be lowered for static distributions by using a larger sample size.

Next, the *Truncation Error* is the change in hit ratio between an accurate TinyLFU (with no Doorkeeper and no false positives) and floating-point counters and that of an accurate TinyLFU with integer counters. The only difference between these two implementations is the type of division performed in the reset operation. The truncation error can be lowered by allowing higher counts by counters (more bits per counter).

Finally, the *Approximation Error* is the difference in hit ratio we obtain by using an approximated version of TinyLFU (as described in this article) instead of an accurate one. This means using a

counting Bloom filter instead of a hash table. The difference in hit ratio is due to false positives that may distort the sampling and make infrequent items appear frequent.

The results for a static Zipf 0.9 distribution are displayed in Figure 22. As can be observed, the approximation error does not appear at all until ≈ 1.25 bytes per sample item. That is, the approximate version of TinyLFU offers the same hit ratio as the accurate version. As expected, the truncation error is smaller for small sample sizes, and therefore, in a 9K-item sample, it is more dominant than in a 17K-item sample. In the 17K sample, the truncation error is almost negligible.

The sampling error is the most tricky one since increasing the sample size only helps with static distributions. The sampling error gets smaller the larger we set the sample size. This behavior is obviously not the case for real workloads, in which the sample size is to be chosen according to an empirical trial-and-error process.

Recall also that the TinyLFU version that was implemented in Caffeine employs CM-Sketch rather than a counting Bloom filter. In its default configuration, it allocates 8 bytes per cached entry and maintains a sample size that is 10 times the cache size. In other words, it uses 0.8 bytes per element. By doubling the allocated space to 16 bytes per entry, the obtained hit ratio increases by approximately $\approx 0.5\%$ on the various traces. In other words, the current configuration of Caffeine carries roughly a 0.5% approximation error.

6 CONCLUSION

We have introduced TinyLFU, an approximate frequency-based cache admission policy. We showed that TinyLFU can augment caches of arbitrary eviction policy and significantly improve their performance. We also optimized the memory consumption of TinyLFU using adaptation of known approximate counting techniques with novel techniques tailored specifically in order to achieve low memory footprints for caches.

Unlike previous schemes that combine ghost cache entries into the eviction policy considerations, in TinyLFU, the admission policy is orthogonal to the eviction policy. This decoupling follows the separation-of-concerns principle, simplifies design and implementation, and enables optimizing each independently. In particular, the use of approximate sketching techniques for the admission policy enables maintaining statistics for a relatively large sample while consuming only small amounts of metadata.

The performance of TinyLFU and its variant W-TinyLFU have been explored and validated through simulations using both synthetic traces and multiple real world traces from several different sources. As mentioned before, TinyLFU is freely available as part of the open-source Caffeine Java caching project (Manes 2016). In the future, we will look into an improved implementation based on a succinct hash table design (Einziger and Friedman 2015b, 2016). Also, we are currently working on dynamically adapting W-TinyLFU parameters to the traces, in order to obtain the best results in all traces without any manual tuning. The basic idea is to maintain a compound score that represents whether the trace is currently behaving in a more “frequency”- or “recency”-oriented manner and automatically adjust W-TinyLFU’s parameters accordingly.

APPENDIX

A THE COMPETITIVE RATIO OF TINYLFU

In this section, we analyze the competitive ratio of TinyLFU. The results that follow assume that the eviction policy is either LRU or FIFO. Yet, we believe that they can be adjusted to other deterministic eviction policies such as MRU and CLOCK. Further, for simplicity, we ignore the effects of false positives in the CBF/MI-CMS of TinyLFU on counters’ precision. The analysis itself follows

similar steps as the competitive analysis of LRU (Borodin and El-Yaniv 1998) and LRFU (Cohen et al. 2002).

THEOREM A.1. *Let CACHE be an LRU or FIFO cache of size C enhanced with a TinyLFU admission policy whose sample size is W . Assuming $W \geq 2C$, the competitive ratio of CACHE is at most W and at best $W/4 + C/2$.*

The theorem follows directly from the combination of the two lemmas that follow.

LEMMA A.2. *CACHE is W competitive.*

PROOF. For the proof, we break any sequence σ of requests into C -phases, where each C -phase consists of requests to C different objects. Thus, the first request of a C -phase (other than the first C -phase) accesses a different object than all other objects in the previous C -phase and it is the first such request in σ . We now show that CACHE cannot incur more than W cache misses during such a C -phase.

Since the size of the cache is C , during each C -phase, each cache replacement policy must incur at least one cache miss (Borodin and El-Yaniv 1998). Further, due to the LRU (or FIFO) replacement policy of CACHE, no item can be inserted into the cache more than once during the same C -phase and any item inserted into the cache or requested at least once while in the cache cannot be evicted during this C -phase.

In particular, denote the first miss in a given C -phase m_1 and let m_2 be any following miss during the same C -phase. If object x is the LRU (or FIFO) eviction candidate of CACHE during m_2 , it is guaranteed that x was not requested between m_1 and m_2 (inclusive). Hence, during m_2 , the TinyLFU estimation for x is bounded by its estimate during m_1 , which is bounded by W/C . Notice that any potential Reset operation occurring while an object y is outside the cache would halve the TinyLFU estimations for both y and any such x proposed due to a miss request to y . In other words, an object that is not inside the cache will enter the cache after at most W/C misses, unless the C -phase ends before.

Overall, during the same C -phase, at most C objects can be inserted into the cache at most once, and each such object may cause at most W/C misses. Hence, altogether, there can be at most W misses during any C -phase. \square

Next, denote that $L = \lceil \frac{W}{2} \log \frac{W}{C} \rceil$. Recall that the maximal value for any counter in TinyLFU is W/C and therefore the maximal possible estimation of each object is also W/C . Further, every $W/2$ requests, each counter is divided by two. Hence, the estimation associated with each object in TinyLFU reaches zero after at most L requests in which it is not accessed.

LEMMA A.3. *Assuming $W \geq 2C$, CACHE is at best $W/4 + C/2$ competitive.*

PROOF. We construct a sequence of requests σ such that the ratio between the number of misses occurred by CACHE while serving σ and the minimal possible number of misses by any cache serving σ approaches $W/4 + C/2$. Our desired sequence consists of accesses to $C + 1$ distinct objects, denoted x_1, x_2, \dots, x_{C+1} . Once again, we partition σ into C -phases, this time specially constructed such that CACHE will experience $W/4 + C/2$ misses when serving this C -phase, while an optimal caching policy could incur only a single miss.

Recall that by the definition of L , after L consecutive requests to the same object x , the TinyLFU estimation of x would be W/C , while all other estimations would be zero. Hence, the first C -phase in σ consists of requests to $x_2^L, x_3^L, x_4^{W/C}, x_5^{W/C}, x_6^{W/C}, \dots, x_C^{W/C}, x_{C+1}^{W/C}$, where x^i means i consecutive requests to x . At the end of this C -phase, the TinyLFU estimation for x_1 is zero, while the TinyLFU estimations of all even objects are W/C (or $W/2C$ depending on the relation between W and C)

and for the odd objects it is one. For simplicity, we assume here that C is an odd number, but the proof can be adjusted for even numbers as well. Further, in terms of eviction priority in the LRU or FIFO cache of *CACHE*, the objects are ordered in the order they were requested.

The second C -phase starts with requests to $x_1^{k_1}, x_2, x_3^{k_3}, x_4 \dots, x_C^{k_C}$, where k_i is the number of consecutive requests required until the estimation of x_i will surpass that of x_{i+1} and therefore it will be admitted into the cache instead of x_{i+1} . By the assumption about the eviction policy and TinyLFU, the requests to odd elements cause k_i misses before they are admitted. Notice that the requests of the first phase initially bring the value of each even element to W/C . Yet, as $C/2$ even elements are requested W/C times, plus $C/2$ odd elements that are requested once, the estimations of the *CACHE* cache victim is already halved by the time it becomes a victim. Hence, $k_i = W/2C$ and the total number of misses due to odd objects is $W/2C \times C/2 = W/4$.

In contrast, the $(C + 1)/2$ even objects cause a single miss each, or $\lceil C/2 \rceil$ misses combined. Hence, the total number of misses during the second phase is $W/4 + C/2$.

We finalize the second C -phase by adding the following requests: $x_1^L, x_2, x_3^{(W/C)}, x_4, x_5^{W/C}, \dots, x_{C-1}, x_C^{W/C}$. These cause no misses as all items are already in the cache. Further, at the end of the phase, the state of TinyLFU and the LRU or FIFO cache of *CACHE* is equivalent to what it was at the end of the first phase. Consequently, we can extend the sequence with additional such C -phases infinitely many times. \square

B ON THE COMPETITIVENESS OF W-TINYLFU VERSUS LRU

Next, we show that for any workload, the hit ratio of an entire W-TinyLFU cache with a window cache of size k is at least as high as the hit ratio of an LRU cache of size k . In particular, this means that the competitive ratio of such a W-TinyLFU cache is at least as good as that of an LRU cache of size k . In the proof, we do not rely on any particular admission policy into the main cache or on the replacement policy of the main cache, meaning that the result holds for any choice of them.

We start with a few notations. The order of requests in a workload is a total order. Given two requests r_1 and r_2 of the same workload σ , we denote by $r_1 < r_2$ the fact that r_1 appears before r_2 in σ and by $r_1 \leq r_2$ the fact that r_2 does not appear before r_1 in σ . When $r_1 \leq r_2$ and $r_2 \leq r_1$, then $r_1 = r_2$ (they are the same request). Given a cache C and a workload σ , we denote by $HR(C, \sigma)$ the hit ratio obtained by C when faced with σ .

THEOREM B.1. *Consider a W-TinyLFU-based cache T with a window cache of size k whose eviction policy is LRU and an LRU cache L of size k . For any workload σ , $HR(T, \sigma) \geq HR(L, \sigma)$.*

PROOF. Assume, by way of contradiction, that the theorem does not hold and there exists a workload σ such that $HR(T, \sigma) < HR(L, \sigma)$. Hence, there has to be at least one request in σ such that r is a hit in L but a miss in T . Choose an arbitrary such request r and denote the object it accesses x .

Since r generated a hit in L , x was accessed by a previous request r' . We continue by case analysis.

Suppose r' was a miss in L . In this case, we claim that r' must have been a hit in T . Otherwise, if r' was a miss in T , then x would have been inserted into the window cache of T , whose eviction policy is LRU and size is the same as the size of L . Hence, if r would have resulted in a miss in T , it means that x was evicted between r' and r from the window cache of T , and therefore there have been requests to at least k other distinct objects between r' and r . Yet, by the LRU replacement policy of L , in such a case, x should have been removed from L as well before r , a contradiction.

Next, suppose r' was a hit in L . From the same arguments as earlier, if r' was a miss in T , then x would have been inserted into the window cache of T at r' and remained there until r , a contradiction. Consequently, we establish that r' was a hit in T as well. Further, from the assumption

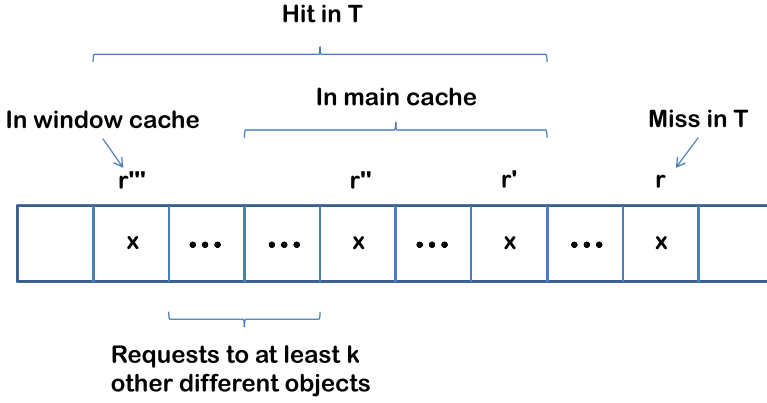


Fig. 23. Illustration of requests for x applied to cache T .

that r is a miss in T , then x must have been stored in the main cache of T during r' and evacuated between r' and r . See the illustration in Figure 23.

Consider the latest request $r'' \leq r'$ in σ at which x is found in T 's main cache, but in the previous request r''' to x in σ is found in T 's window cache. By definition, such an r''' must exist. Further, all requests to x between r''' and r' are a hit in T . From the same arguments as before, there are requests to at least k other distinct objects than x is between r''' and r'' . Thus, either r''' or r'' must be a miss in L .

In summary, we establish that for any object x accessed during σ , for any possible request that results in a miss in T and a hit in L , we can trace a unique previous request to x that results in a hit in T and a miss in L . Thus, $HR(L, \sigma)$ cannot be larger than $HR(T, \sigma)$, a contradiction. \square

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments, which greatly improved the presentation of this work.

REFERENCES

- Ismail Ari, Melanie Gottwals, and Dick Henze. 2006. Performance boosting and workload isolation in storage area networks with SANCache. In *Proceedings of the 23rd IEEE/14th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'06)*. 263–273.
- Martin Arlitt, Ludmila Cherkasova, John Dille, Rich Friedrich, and Tai Jin. 1999. Evaluating content management techniques for web proxy caches. In *Proceedings of the 2nd Workshop on Internet Server Performance*.
- Martin Arlitt, Rich Friedrich, and Tai Jin. 2000. Performance evaluation of web proxy cache replacement policies. *Perform. Eval.* 39, 1–4 (Feb. 2000), 149–164.
- Hilla Atzmon, Roy Friedman, and Roman Vitenberg. 2002. Replacement policies for a distributed object caching service. In *Confederated Int. Conf. DOA, CoopIS and ODBASE*. 661–674.
- Sorav Bansal and Dharmendra S. Modha. 2004. CAR: Clock with adaptive replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04)*. 187–200.
- László Belady. 1966. A study of replacement algorithms for a virtual storage computer. *IBM Syst. J.* 5, 2 (1966), 78–101.
- Giuseppe Bianchi, Elisa Boschi, Simone Teofili, and Brian Trammell. 2010. Measurement data reduction through variation rate metering. In *Proceedings IEEE INFOCOM*.
- Giuseppe Bianchi, Nico d'Heureuse, and Saverio Niccolini. 2011. On-demand time-decaying Bloom filters for telemarketer detection. *SIGCOMM Comput. Commun. Rev.* 41, 5 (Oct. 2011), 5–12.
- G. Bianchi, S. Teofili, E. Boschi, B. Trammell, and C. Greco. 2010. Scalable and fast approximate excess rate detection. In *Proceedings of the Future Network and Mobile Summit*.

- Allan Borodin and Ran El-Yaniv. 1998. *Online Computation and Competitive Analysis*. Cambridge University Press.
- Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. 1999. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM*. 126–134.
- Yu Jin Cao, Jing, Aiyu Chen, Tian Bu, and Zhi-Li Zhang. 2009. Identifying high cardinality internet hosts. In *IEEE INFOCOM*. 810–818.
- Wei Koong Chai, Diliang He, Ioannis Psaras, and George Pavlou. 2012. Cache “less for more” in information-centric networks. In *Proceedings of the 11th International IFIP TC 6 Conference on Networking*. Springer-Verlag, 27–40.
- Kai Cheng and Yahiko Kambayashi. 2000. LRU-SP: A size-adjusted and popularity-aware LRU replacement algorithm for web caching. In *Proceedings of the 24th IEEE International Computer Software and Applications Conference (COMPSAC’00)*. 48–53.
- Xu Cheng, C. Dale, and Jiangchuan Liu. 2008. Statistics and social network of youtube videos. In *Proceedings of the 16th International Workshop on Quality of Service (IWQoS’08)*. 229–238.
- Ludmila Cherkasova. 1998. *Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy*. HP Technical Report.
- Gregory Chockler, Guy Laden, and Ymir Vigfusson. 2010. Data caching as a cloud service. In *Proceedings of the 4th ACM International Workshop on Large Scale Distributed Systems and Middleware (LADIS’10)*. ACM, 18–21.
- Gregory Chockler, Guy Laden, and Ymir Vigfusson. 2011. Design and implementation of caching services in the cloud. *IBM J. Res. Devel.* 55, 6 (2011), 9:1–9:11.
- Baek-Young Choi, Jaesung Park, and Zhi-Li Zhang. 2002. Adaptive random sampling for load change detection. *SIGMETRICS Perform. Eval. Rev.* 30, 1 (June 2002), 272–273.
- Edith Cohen, Haim Kaplan, and Uri Zwick. 2002. Competitive analysis of the LRFU paging algorithm. *Algorithmica* 33, 4 (2002), 511–516.
- Saar Cohen and Yossi Matias. 2003. Spectral bloom filters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 241–252.
- F. J. Corbato. 1968. *A Paging Experiment with the Multics System*. Technical Report Project MAC Report MAC-M-384. MIT.
- Graham Cormode and S. Muthukrishnan. 2004. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms* 55 (2004), 29–38.
- Xenofontas Dimitropoulos, Marc Stoecklin, Paul Hurley, and Andreas Kind. 2008. The eternal sunshine of the sketch data structure. *Comput. Netw.* 52, 17 (Dec. 2008), 3248–3257.
- Gil Einziger, Benny Fellman, and Yaron Kassner. 2015. Independent counter estimation buckets. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM’15)*. 2560–2568.
- Gil Einziger and Roy Friedman. 2015a. A formal analysis of conservative update based approximate counting. In *Proceedings of the International Conference on Computing, Networking and Communications (ICNC’15)*. 255–259.
- Gil Einziger and Roy Friedman. 2015b. TinySet - An access efficient self adjusting bloom filter construction. In *Proceedings of the 24th International Conference on Computer Communication and Networks (ICCCN’15)*.
- Gil Einziger and Roy Friedman. 2016. Counting with TinyTable: Every bit counts! In *Proceedings of the 17th International Conference on Distributed Computing and Networking (ICDCN’16)*.
- Gil Einziger, Roy Friedman, and Yoav Kantor. 2016. Shades: Expediting Kademlia lookup process. *Comput. Netw.* 99 (April 2016), 37–50.
- Cristian Estan, Ken Keys, David Moore, and George Varghese. 2004. Building a better netflow. *SIGCOMM Comput. Commun. Rev.* 34, 4 (Aug. 2004), 323–336.
- Cristian Estan and George Varghese. 2002. New directions in traffic measurement and accounting. *SIGCOMM Comput. Commun. Rev.* 32, 4 (Aug. 2002), 323–336.
- Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. 2000. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.* 8, 3 (June 2000), 281–293.
- Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux J.* 2004, 124 (Aug. 2004), 5.
- Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. 2007. Youtube traffic characterization: A view from the edge. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement (IMC’07)*. 15–28.
- Amit Goyal, Hal Daumé, and Graham Cormode. 2012. Sketch algorithms for estimating point queries in NLP. In *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. 1093–1103.
- Amit Goyal and Hal Daumé. 2011. Lossy conservative update (LCU) sketch: Succinct approximate count storage. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence*.
- Amit Goyal, Jagadeesh Jagarlamudi, Hal Daumé III, and Suresh Venkatasubramanian. 2010. Sketching techniques for large scale NLP. In *Proceedings of the 6th NAACL HLT Web as Corpus Workshop (WAC’10)*. 17–25.
- John L. Hennessy and David A. Patterson. 2012. *Computer Architecture - A Quantitative Approach*. 5th ed. Morgan Kaufmann.

- Chengchen Hu, Bin Liu, Hongbo Zhao, Kai Chen, Yan Chen, Chunming Wu, and Yu Cheng. 2010. DISCO: Memory efficient and accurate flow statistics for network measurement. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems (ICDCS'10)*. 665–674.
- Chengchen Hu, Sheng Wang, Jia Tian, Bin Liu, Yu Cheng, and Yan Chen. 2008. Accurate and efficient traffic monitoring using adaptive non-linear sampling method. In *INFOCOM*. IEEE, 26–30.
- Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. 2013. An analysis of Facebook photo caching. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. 167–181.
- Song Jiang, Feng Chen, and Xiaodong Zhang. 2005. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC'05)*. 35–35.
- Song Jiang and Xiaodong Zhang. 2002. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS* (2002), 31–42.
- Theodore Johnson and Dennis Shasha. 1994. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*. 439–450.
- George Karakostas and Dimitrios Serpanos. 2002. Exploitation of different types of locality for web caches. In *Proceedings of the 7th International Symposium on Computers and Communications (ISCC'02)*.
- Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. 1994. Caching strategies to improve disk system performance. *IEEE Computer* 27, 3 (1994), 38–46.
- Anirban Ketan Shah and Mitra Dhruv Matani. 2010. *An $O(1)$ Algorithm for Implementing the LFU Cache Eviction Scheme*. Technical Report. Retrieved from <http://dhruvbird.com/lfu.pdf>.
- Piotr Kolaczowski. 2007. Memory efficient algorithm for mining recent frequent items in a stream. In *Proceedings of the International Conference on Rough Sets and Intelligent Systems Paradigms (RSEISP'07)*. 485–494.
- Marc Liberatore and Prashant Shenoy. 2016. UMass Trace Repository. Retrieved from <http://traces.cs.umass.edu/index.php/Main/About>.
- Yi Lu, Andrea Montanari, Balaji Prabhakar, Sarang Dharmapurikar, and Abdul Kabbani. 2008. Counter braids: A novel counter architecture for per-flow measurement. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. 121–132.
- Ben Manes. 2016. Caffeine: A High Performance Caching Library for Java 8. Retrieved from <https://github.com/benmanes/caffeine>.
- Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*. 115–130.
- Nimrod Megiddo and Dharmendra S. Modha. 2006. System and method for implementing an adaptive replacement cache policy. US Patent 6996676, February 7.
- Satish Narayanasamy, Timothy Sherwood, Suleyman Sair, Brad Calder, and George Varghese. 2003. Catching accurate profiles in hardware. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03)*. 269–280.
- Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Rec.* 22, 2 (June 1993), 297–306.
- OpenBSD. 2014. TuQueue. Retrieved from <https://www.tedunangst.com/flak/post/2Q-buffer-cache-algorithm>.
- Stefan Podlipnig and Laszlo Böszörményi. 2003. A survey of web cache replacement strategies. *ACM Comput. Surv.* 35, 4 (Dec. 2003), 374–398.
- Ioannis Psaras, Wei Koong Chai, and George Pavlou. 2012. Probabilistic in-network caching for information-centric networks. In *Proceedings of the 2nd Edition of the ICN Workshop on Information-Centric Networking*. ACM, 55–60.
- Konstantinos Psounis and Balaji Prabhakar. 2002. Efficient randomized web-cache replacement schemes using samples from past eviction times. *IEEE/ACM Trans. Netw.* 10, 4 (Aug. 2002), 441–455.
- Frederic Raspall and Sebastia Sallent. 2008. Adaptive shared-state sampling. In *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement (IMC'08)*. 271–284.
- Dimitrios N. Serpanos and Wayne H. Wolf. 2000. Caching web objects using zipf's law. In *Proceedings of the IEEE International Conference on Multimedia and Expo*. 727–730.
- Rade Stanojevic. 2007. Small active counters. In *Proceedings of the 26th IEEE International Conference on Computer Communications (INFOCOM'07)*. 2153–2161.
- Geetika Tewari and Kim Hazelwood. 2004. *Adaptive Web Proxy Caching Algorithms*. Technical Report TR-13-04. Harvard University.
- Erez Tsidon, Iddo Hanniel, and Isaac Keslassy. 2012. Estimators also need shared values to grow together. In *INFOCOM*. 1889–1897.
- Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. 2009. Wikipedia workload analysis for decentralized hosting. *Elsevier Comput. Netw.* 53, 11 (July 2009), 1830–1845.

- Stephen Williams, Marc Abrams, and Charles R. Standridge. 1996. Removal Policies in Network Caches for World-Wide Web Documents. In *Proceedings of the ACM SIGCOMM Conference*. 293–305.
- Neal Young. 1991. On-line caching as cache size varies. In *Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '91)*. 241–250.
- Yuanyuan Zhou, James Philbin, and Kai Li. 2001. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the USENIX Annual Technical Conference*. 91–104.

Received December 2015; revised September 2017; accepted September 2017