TinyLFU: A Highly Efficient Cache Admission Policy

Gil Einziger and Roy Friedman
Computer Science Department
Technion
Haifa 32000, Israel

***Abstract*—This paper proposes to use a *frequency based cache admission policy* in order to boost the effectiveness of caches subject to skewed access distributions. Rather than deciding on which object to evict, TinyLFU decides, based on the recent access history, whether it is worth admitting an accessed object into the cache at the expense of the eviction candidate.**

Realizing this concept is enabled through a novel approximate LFU structure called *TinyLFU*, which maintains an approximate representation of the access frequency of recently accessed objects. TinyLFU is extremely compact and lightweight as it builds upon Bloom filter theory.

The paper shows an analysis of the properties of TinyLFU including simulations of both synthetic workloads as well as YouTube and Wikipedia traces.

I. Introduction

Caching is one of the most basic and effective methods in computer science for boosting system's performance in a multitude of domains. It is obtained by keeping a small percent 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 why 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 noting that in many interesting domains of 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 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-rate*. 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-rate [1].

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, which items should be *evicted* from the cache. Obviously, this should be done in an efficient manner, in order to avoid a situation in which the computation and storage overheads required to answer these questions surpasses the benefit of using the cache. The storage 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 *meta-data* of the cache. In many caching schemes, the time complexity of manipulating the meta-data as well as the size of the meta-data 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) policy yields the highest cache hit ratio [2, 3]. 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 meta-data. 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 last W accesses both in order to limit the size of the meta-data 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) [4], as elaborated below. Another alternative that relies on the "time locality" property is known as the *Least Recently Used* (LRU) [1] scheme, 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 LFU[1] and automatically adapts to temporal changes in the data access patterns. Yet, under many workloads, LRU requires much larger caches than LFU in order to obtain the same hit-rate.

In this paper, we make two main contributions. First, we present a novel highly efficient probabilistic data structure, nicknamed TinyLFU, that enables approximating the tempo-

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

ral probability distribution of skewed access patterns, such as Zipf-like distributions, which were shown to represent Web accesses and Internet based video accesses [2, 3, 5]. Using TinyLFU, we generate an approximated version of WLFU called TLFU. We show that TLFU achieves similar performance to WLFU with significantly lower meta-data costs. Moreover, the meta-data 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 extremely fast manipulation.

The second contribution stems from an observation that most caching schemes focus on *eviction policies*, i.e., deciding which item should be evicted, rather than on *admission policies*, which determine what items should be inserted to the cache. In this paper we show that at least for skewed probability distributions, the admission policy can be even more important than the eviction policy. Specifically, since TinyLFU enables us to approximate the temporal access distribution of all objects ever encountered in a fairly accurate yet compact manner, TinyLFU can be added to caches of arbitrary eviction policies, and augment them with a frequency based admission policy. That is, we take an arbitrary eviction policy and ask it to find an eviction candidate. We then compare the approximate popularity of the newly accessed item, as recorded by TinyLFU, with the popularity of the eviction candidate. If the newly accessed item is more popular, then the candidate is indeed evicted. Otherwise, it remains in the cache. Surprisingly, we have found that for skewed access distributions, (i) adding the TinyLFU admission policy dramatically increases the hit-rate obtained by the system, and (ii) once the TinyLFU admission policy is in place, the difference between different eviction policies becomes marginal.

Notice 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 [6]. Hence, we distinguish between them by referring to the former as *Perfect LFU* (PLFU) and the latter as *In-Memory LFU*. Since WLFU outperforms In-Memory LFU [2] (at the cost of larger meta-data), we compare our work only against WLFU and PLFU.

For deductive reasons, we first present the TinyLFU admission policy architecture and only then explain and explore its storage optimizations and accuracy. The rest of this paper is organized as follows: Related work is discussed in Section II. We present TinyLFU in Section III. The performance results are shown in Section IV and we conclude with a discussion in Section V.

II. Related Work

A. 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 prohibitively high, and PLFU does not adapt to dynamic changes in the distribution [4, 7, 8]. 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, evicting, if needed, the least frequently accessed item among those that are in the cache [6]. In-Memory LFU is usually maintained using a heap. The time complexity of managing LFU heaps was thought to be $O(\log(N))$ until recently, when an $O(1)$ construction was shown in [9]. Yet, even with this improvement, In-Memory LFU still suffers from slow reaction to changes in the frequency distribution, and 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 [6].

Aging was introduced in [7] to improve the ability of In-Memory LFU to react to changes. It is obtained by limiting the maximum frequency count of cache 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 [8].

As mentioned above, WLFU only maintains the access frequency for a window of the last W requests [4]. In order to maintain the 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.

ARC [10] combines recency and frequency by maintaining meta data in two LRU lists. The first contains data on items that were only accessed once, while the second contains data on 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-rate. If the first buffer incurs a higher hit-rate, ARC is closer to LRU, while if the second one yields a higher hit-rate, ARC behaves closer to LFU.

2Q [11] is a page replacement policy for operating systems. This policy uses two queues, A1 and A2. At first, a page is entered into A1 that is ordered as a FIFO queue. New pages replace pages in A1. Yet, when a page in A1 is referenced it moves to A2 and treated as a hot page. Hence, A2 is populated with the most frequently used pages.

LRU-K [12] 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 [13] offers to simply remove the largest item first, while same size items are ordered by LRU. LRU-SP [14] weighs both the size and the frequency of an item when picking a cache victim.

GDSF [15] 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 [16] that only factors size and cost in its decisions. In [17], the latency is also taken into account in workloads when the data is hierarchical, i.e., in order to fetch a child item one must first fetch its predecessor item.

A related approach to ours is introduced in [18], which suggests to augment 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 whether to evict an item in the cache does not depend on the relative frequency of this item vs. the frequency of the currently accessed item and an explicit list of n items need to be maintained. This method was shown in [18] to somewhat improve the hit-rate of various caching suggestions at the cost of significant meta-data overhead.

In summary, TinyLFU relates to the above suggestions by being a mechanism that can be used to augment other caching policies by enhancing them with approximated statistics on a large history while providing both quick access time and low storage overhead.

B. Approximated Counting Architectures

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

Sampling methods [19, 20, 21] 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 [22] reduce the meta data size but require a long decode operation and are therefore not applicable to caching. Another approach is to compress the counter representation itself [23, 24, 25]. In TinyLFU we manage to represent the histogram using short counters, and thus these methods do not help us.

Multi hash sketches such as *Spectral Bloom Filters* (SBF) [26] and the *Count Min Sketch* (CM-Sketch) [27] are appealing in our context. Since they 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 suggestion aimed to achieve variable sized counters. Such a complex implementation is not needed in our case, since we only require small counters. The CM-Sketch on the other hand, offers a simple implementation, but is relatively inaccurate and will therefore take more storage to implement.

For TinyLFU, we chose to implement an SBF using a simple Counting Bloom Filter [28]. We used an alternative add operation that is referred to as *Minimal Increment* or *Conservative Update*. This method was shown to increase the accuracy of the filter in cases where all the increments are always positive [26, 29].

C. Interesting caching applications

Application of data caching includes using it as cloud services [30, 31].MemcacheD [32] 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 the routers themselves using a technique called *in network caching* [33, 34]. This technique allows content to be named explicitly, and routers in the data center to cache data in order to increase the network capacity.

TinyLFU can be integrated in all the examples above. 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 [31, 32] is a variant of LRU with no admission policy. As we show in this paper, adding a TinyLFU based admission filter to an LRU eviction policy greatly boosts its performance.

III. TinyLFU Architecture

A. TinyLFU Overview

In the literature, the term LFU refers to two different things: *LFU cache eviction policy* is a policy that picks the victim of the cache according to LFU and the *Frequency based cache admission policy* is a policy that admits items to the cache only if they are more frequent than the cache victim. TinyLFU architecture is ilustrated in figure 1, in our architecture, 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-rate.

To do so, TinyLFU maintains statistics of items frequency over the recent history. Storing these statistics is considered prohibitively expensive for practical implementation and therefore TinyLFU approximates this statistics 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 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 in order to be considered a practical cache management technique.

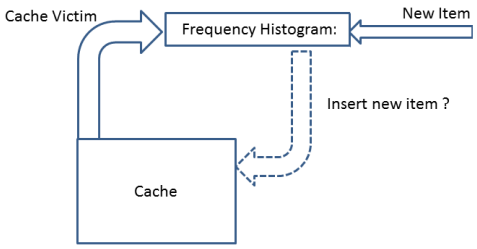


Figure 1. A general cache augmented with TinyLFU

B. Approximate Counting Overview

As indicated in Section II above, SBF and Minimal Increment CM-Sketch are roughly equivalent very popular approximate counting schemes. Thus, for simplicity, we limit our discussion to SBF. 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.

An SBF 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. For example, if we use 3 hash functions, upon item arrival, 3 counters are read. Assuming we read $\{2, 2, 5\}$, the `Add` operation increments only the two left counters from 2 to 3 while the third counter remains untouched.

Intuitively, the `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.

C. Freshness Mechanism

To the best of our knowledge, keeping approximation sketches fresh has only been studied in [35], where a sliding window 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. This way, old events are forgotten.

The method of [35] 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 memory consumption, since we need to store the same items in m different sketches and each item is allocated more counters.

Instead, we have invented a novel method for keeping the sketch fresh, the *Reset* method described below. The Reset operation is simple. Every time we add an item to the approximation sketch, we increment a counter. Once this counter reaches the window size (W), we divide this counter and all other counters in the approximation sketch by 2. This 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. Since the accuracy of an approximation sketch can always be increased by using more memory, we show that the Reset method in fact reduces the total memory cost since we get a significantly more accurate sketch for the same memory cost.

The downside of this operation is an expensive infrequent operation that goes over all the counters in the approximation scheme and divides them by 2. Yet, division by 2 can be implemented efficiently in hardware using shift registers. Further, 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 the truncation error it adds to the sketch. This error is caused since we use an integer division and therefor a counter that shows 3 will be reset to 1 and not to 1.5.

1) Reset Correctness:

Definition Denote W the window size, f_i the frequency of item i, and h_i the height of i in the histogram.

Lemma 3.1: Under constant distribution, at the end of each window (immediately before each Reset operation), the expected hight of i in the histogram is $E(h_i) = f_i * W$.

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 $E(h_i) = f_i * W$.

Step: Assume correctness for $r < j$ and prove it for $r = j$. From the induction hypothesis, until the $j-1$ Reset operation occurs, $E(h_i) = f_i * W$. Therefore, immediately after the $j-1$ Reset operation, $E(h_i) = \frac{f_i * w}{2}$. There are exactly $W/2$ samples until the next Reset operation and therefore $E(h_i)$ is expected to be incremented $\frac{f_i * W}{2}$ times. Since the expectation is additive we conclude that right before the $j'th$ Reset operation: $E(h_i) = f_i * W$. ■

2) Reset Truncation Error: Each time we perform a Reset, we execute an integer division. This caps the size of the histogram by eliminating entries that reach zero. Yet, this also introduces a truncation error on each division by two. Therefore, we may forget a single occupance of the counter. When we read a counter after a single Reset operation, the value we read can be as much as 0.5 lower than the value 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 truncation error worst case 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 $2/W$ right after a Reset operation, since a Reset operation is performed once per $W/2$ operations.

D. Storage Size Reduction

Our storage cost 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 amount of counters allocated by the sketch. These storage saving optimizations are detailed below.

1) Using 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 window), 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 window W, we can safely cap the counters by W/C.

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 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 window is 8 times larger than the cache itself. For comparison, if we consider a small 2K items cache with a history window of 16K items and we do not employ the small counters optimization, then the required counter size is 14 bits.

2) Doorkeeper: In many workloads, and especially in heavy tailed workloads, unpopular items account for a considerable portion of all accesses. This implies that if we count how many times each unique item in the window appeared, the majority of the counters are assigned to items that are not likely to appear more than once in the sliding window. Hence, to further reduce the size of our counters, we have developed the *Doorkeeper* mechanism that enables us to avoid allocating counters to most tail objects.

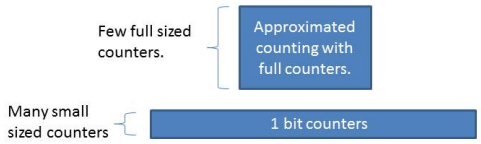


Figure 2. TinyLFU structure

The Doorkeeper is a regular Bloom filter placed in front of the approximate counting scheme. First timers are inserted only to the Doorkeeper and are cleared upon Reset. Only items that are already contained in the Doorkeeper are inserted to the approximate counting scheme. Although the Doorkeeper adds its own memory space to store its Bloom filter, it obviates allocating counters to tail items. Hence, in many cacheable workloads, this optimization significantly reduces the memory consumption of TinyLFU.

TinyLFU and the Doorkeeper are illustrated in Figure 2. We note that a similar technique was previously suggested by [36] in the context of network security.

E. Test Case: TinyLFU vs. 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 [35]. 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 items cache augmented by a 9K items frequency histogram. For this workload, TinyLFU requires its counters to count up to 9. This is obtained using 3 bits full counters that can count up to 8, in addition to the Doorkeeper that can count up to 1. 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 characteristics of many interesting real-world workloads.

We summarize the storage requirements of both frequency histograms in Figure 3. As can be observed in this workload, TinyLFU is required to store approximately 10% less unique values due to the fact that 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 window were allocated an additional 3 bits 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%.

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

Figure 3. TinyLFU vs unoptimized approximate frequency histogram

IV. Experimental Results

A. Methodology

In order to test the feasibility of our approach, we have implemented a Java based prototype in which we connected TinyLFU to the following cache eviction schemes: LRU, Random and LFU. This implementation 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 above 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 a hit or a miss as well as overall statistics.

We compared the hit-rate using both constant distributions of Zipf 0.7 and Zipf 0.9[2], a workload trace generated from YouTube [37], and a Wikipedia workload [38].

The YouTube trace includes a weekly summary of the number of accesses to each video rather than a continuous time-line of requests. Hence, for each reported week, we have calculated the corresponding approximate access distribution, and have generated synthetic accesses that follow this distribution on a week by week basis.

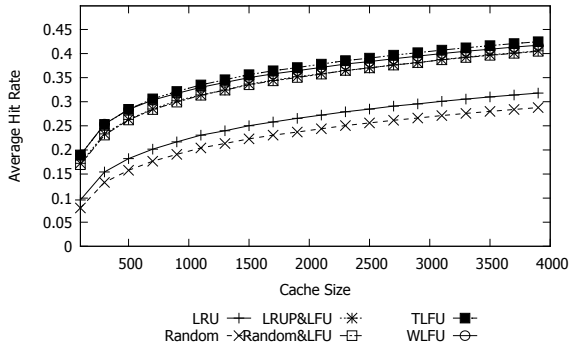
The Wikipedia workload contains 10% of the traffic to Wikipedia during two months starting in September 2007. Due to the enormous size of this trace, we sampled different starting points from which we played 100 million consecutive requests and measured the obtained hit-rate.

In the synthetic workloads, 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 windows) and we present them at their highest hit rate. In the Wikipedia and YouTube workloads, caches are not warmed up since as the distribution gradually changes over time, no warmup is necessary.

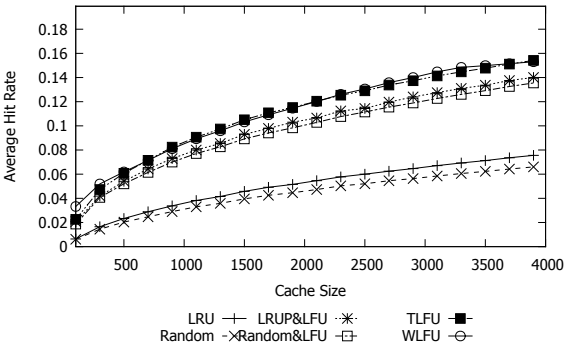
We have also used the YouTube workload to measure the impact of the distribution change rate on the performance. To do so, instead of playing the entire amount of views in that week, we only played a few windows from that week. We believe these tests simulate the behavior of the caches when the workload is very dynamic.

In our figures, LRU and Random refer to eviction policies, while LRU&LFU and Random&LFU refers to LRU/Random caches augmented by TinyLFU. For LFU cache eviction we tested two options, WLFU that uses both 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.

[2]We have experimented with several other skewed distributions and obtained very similar qualitative results.



(a) Window size is 32 times the cache size, Zipf 0.9



(b) Window size is 32 times the cache size, Zipf 0.7

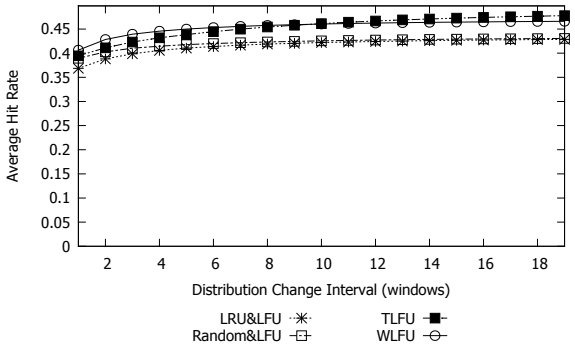
Figure 4. Augmenting arbitrary caches with a TinyLFU admission policy

B. Results of Augmenting Caches with TinyLFU

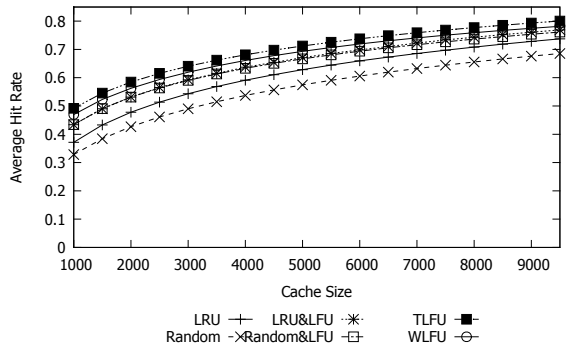
Figure 4 shows the results of TinyLFU admission policy on the performance of the above mentioned eviction policies under constant skewed distributions. As can be seen, under constant distributions, all caches that are augmented with TinyLFU behave in a similar way. Surprisingly, LFU cache eviction yields only a marginal benefit over the TinyLFU augmented LRU and Random techniques. Let us note that in such skewed distributions, the maximal cache hit-rate 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, TinyLFU cache admission policy is an attractive enhancement. While augmenting In-memory LFU yields slightly higher hit-rate, the overheads of In-memory LFU may justify using a simpler cache eviction policy. Particularly, LRU and even Random offer low overheads with comparable hit-rate.

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 at Apr. 16th, 2008 [37]. 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

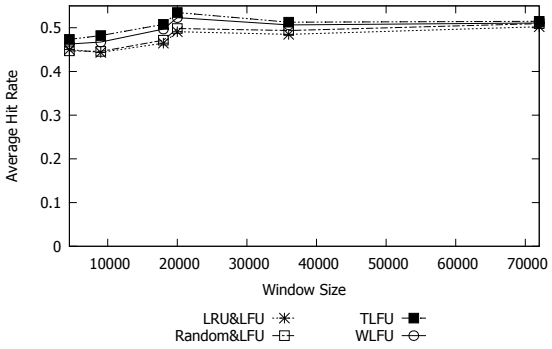


(a) Effect of the change speed on the hit-rate for 1000 items cache, and window size of 9000

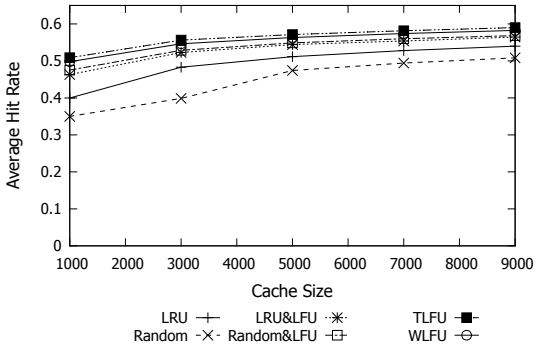


(b) Cache size vs. hit-rate for window size of 9*CacheSize

Figure 5. YouTube dataset



(a) Window size vs. hit-rate for a 1000 items cache



(b) Cache size vs. hit-rate for window size of 20*CacheSize

Figure 6. Evaluation over the Wikipedia trace

distributions every given amount of requests.

We measured two metrics: The first is how fast can we change the distributions, i.e, what is the effect of the number of samples we perform from each week's distribution on the hit-rate of our TinyLFU augmented caches. The second is the impact of the cache size on the achieved hit-rate when the distribution change speed is taken form the trace.

The result of this experiment are shown in Figure 5. As can be observed, TinyLFU is effective in augmenting arbitrary caches even in dynamic workloads. Further, the benefit is greater 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 cache is more significant. Therefore, picking the correct cache victim seems to be more important on dynamic workloads than in static workloads.

The third measurement we made was running the workloads on the Wikipedia access trace. We first studied the required ratio between window size and cache size on samples 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 6. Unlike static workloads, real life workloads gradually change over time, therefore using a very large window can even reduce the obtain hit rate as it slows the paste by which the cache adjusts to the workload.

We note that although WLFU and TLFU achieved nearly identical hit rate, the main difference between WLFU and TLFU is in their meta-data costs. For example, in the YouTube workload we used only 0.57 bytes per window element. Since there are 9 window elements per cache entry, the total meta-data cost of TinyLFU is 5.13 bytes per cache entry. If we consider that each cache entry should contain a video ID that requires 11 bytes, we conclude that TinyLFU is able to approximately remember a history 9 times bigger than the cache with less storage 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 memory 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 overheads, WLFU's admission policy still requires almost 20 times more memory than TinyLFU.

V. Conclusion

We have introduce TinyLFU, a 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 footprint.

For many cache sizes and workloads, TinyLFU requires up to 1 byte per window element in order to function with negligible approximation error, rendering the total cost of the admission policy practical. The performance of TinyLFU has been explored and validated through simulations using both synthetic traces as well as YouTube and Wikipedia traces.

We hope that our results will convince cache designers to shift some of the focus from the replacement policies to considering admission policies. Looking into the future we would like to adopt TinyLFU to domain specific applications such as cloud caching services and in network caching.

REFERENCES

[1] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.

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

[3] Dimitrios N. Serpanos and Wayne H. Wolf. Caching web objects using zipf's law. pages 320–326, 1998.

[4] G. Karakostas and D. N. Serpanos. Exploitation of different types of locality for web caches. In *Proc. of the 7th Int. Symposium on Computers and Communications (ISCC)*, 2002.

[5] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. Youtube traffic characterization: a view from the edge. In *Proc. of the 7th ACM SIGCOMM Conf. on Internet measurement (IMC)*, pages 15–28, 2007.

[6] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, December 2003.

[7] Martin Arlitt, Rich Friedrich, and Tai Jin. Performance evaluation of web proxy cache replacement policies. *Perform. Eval.*, 39(1-4):149–164, February 2000.

[8] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating content management techniques for web proxy caches. In *In Proc. of the 2nd Workshop on Internet Server Performance*, 1999.

[9] Anirban Ketan Shah and Mitra Dhruv Matani. An o(1) algorithm for implementing the lfu cache eviction scheme. Technical report, 2010. "http://dhruvbird.com/lfu.pdf".

[10] Nimrod Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proc. of the 2nd USENIX Conf. on File and Storage Technologies (FAST)*, pages 115–130, 2003.

[11] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proc. of the 20th Int. Conf. on Very Large Data Bases (VLDB)*, pages 439–450, 1994.

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

[13] Stephen Williams, Marc Abrams, and Charles R. Standridge. Removal policies in network caches for world-wide web documents, 1996.

[14] Kai Cheng and Yahiko Kambayashi. Lru-sp: A size-adjusted and popularity-aware lru replacement algorithm for web caching. In *24th IEEE Int. Computer Software and Applications Conf. (COMPSAC)*, pages 48–53, 2000.

[15] Ludmila Cherkasova. Improving www proxies performance with greedy-dual-size-frequency caching policy. Technical report, In HP Tech. Report, 1998.

[16] Neal Young. On-line caching as cache size varies. In *Proc. of the second annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 241–250, 1991.

[17] Hilla Atzmon, Roy Friedman, and Roman Vitenberg. Replacement policies for a distributed object caching service. In *Confederated Int. Conf.s DOA, CoopIS and ODBASE*, pages 661–674, 2002.

[18] Geetika Tewari and Kim Hazelwood. Adaptive web proxy caching algorithms. Technical Report TR-13-04, "Harvard University".

[19] Baek-Young Choi, Jaesung Park, and Zhi-Li Zhang. Adaptive random sampling for load change detection. *SIGMETRICS Perform. Eval. Rev.*, 30(1):272–273, June 2002.

[20] Cristian Estan, Ken Keys, David Moore, and George Varghese. Building a better netflow. *SIGCOMM Comput. Commun. Rev.*, 34(4), August 2004.

[21] Chengchen Hu, Sheng Wang, Jia Tian, Bin Liu 0001, Yu Cheng, and Yan Chen. Accurate and efficient traffic monitoring using adaptive non-linear sampling method. In *INFOCOM*, pages 26–30. IEEE, 2008.

[22] Yi Lu, Andrea Montanari, Balaji Prabhakar, Sarang Dharmapurikar, and Abdul Kabbani. Counter braids: a novel counter architecture for per-flow measurement. In *Proc. of the ACM SIGMETRICS Int. Conf. on Measurement and modeling of computer systems*, pages 121–132. ACM, 2008.

[23] Chengchen Hu, Bin Liu, Hongbo Zhao, Kai Chen, Yan Chen, Chunming Wu, and Yu Cheng. Disco: Memory efficient and accurate flow statistics for network measurement. In *Proc. of the IEEE 30th Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 665–674, 2010.

[24] Rade Stanojevic. Small active counters. In *26th IEEE Int. Conf. on Computer Communications (INFOCOM)*, pages 2153–2161, 2007.

[25] Erez Tsidon, Iddo Hanniel, and Isaac Keslassy. Estimators also need shared values to grow together. In *INFOCOM*, pages 1889–1897, 2012.

[26] Saar Cohen and Yossi Matias. Spectral bloom filters. In *Proc. of the ACM SIGMOD Int. Conf. on Management of data*, pages 241–252, 2003.

[27] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55:29–38, 2004.

[28] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, June 2000.

[29] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. *SIGCOMM Comput. Commun. Rev.*, 32(4):323–336, August 2002.

[30] Gregory Chockler, Guy Laden, and Ymir Vigfusson. Data caching as a cloud service. In *Proceedings of the 4th ACM International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, LADIS '10, pages 18–21. ACM, 2010.

[31] Gregory Chockler, Guy Laden, and Ymir Vigfusson. Design and implementation of caching services in the cloud. *IBM Journal of Research and Development*, 55(6):9:1–9:11, 2011.

[32] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, August 2004.

[33] Ioannis Psaras, Wei Koong Chai, and George Pavlou. Probabilistic in-network caching for information-centric networks. In *Proceedings of the second edition of the ICN workshop on Information-centric networking*, pages 55–60. ACM, 2012.

[34] Wei Koong Chai, Diliang He, Ioannis Psaras, and George Pavlou. Cache "less for more" in information-centric networks. In *Proceedings of the 11th international IFIP TC 6 conference on Networking*, pages 27–40. Springer-Verlag, 2012.

[35] Xenofontas Dimitropoulos, Marc Stoecklin, Paul Hurley, and Andreas Kind. The eternal sunshine of the sketch data structure. *Comput. Netw.*, 52(17):3248–3257, December 2008.

[36] Jing Cao, Yu Jin 0001, Aiyou Chen, Tian Bu, and Zhi-Li Zhang. Identifying high cardinality internet hosts. In *INFOCOM*, pages 810–818. IEEE, 2009.

[37] Xu Cheng, C. Dale, and Jiangchuan Liu. Statistics and social network of youtube videos. In *16th Int. Workshop on Quality of Service (IWQoS)*, pages 229–238, june 2008.

[38] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009.