

Adaptive Software Cache Management

Gil Einziger

Computer Science Department

Ben Gurion University

gilga1983@gmail.com

Roy Friedman

Computer Science Department

Technion

roy@cs.technion.ac.il

Ohad Eytan

Computer Science Department

Technion

ohadey@cs.technion.ac.il

Ben Manes

Independent

ben.manes@gmail.com

ABSTRACT

Developing a silver bullet software cache management policy is a daunting task due to the variety of potential workloads. In this paper, we investigate an adaptivity mechanism for software cache management schemes which offer tuning parameters targeted at the frequency vs. recency bias in the workload. The goal is automatic tuning of the parameters for best performance based on the workload without any manual intervention. We study two approaches for this problem, a *hill climbing* solution and an *indicator* based solution. In hill climbing, we repeatedly reconfigure the system hoping to find its best setting. In the indicator approach, we estimate the workloads' frequency vs. recency bias and adjust the parameters accordingly in a single swoop.

We apply these adaptive mechanisms to two recent software management schemes. We perform an extensive evaluation of the schemes and adaptation mechanisms over a large selection of workloads with varying characteristics. With these, we derive a parameterless software cache management policy that is competitive for all tested workloads.

ACM Reference Format:

Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. 2018. Adaptive Software Cache Management. In *19th International Middleware Conference (Middleware '18), December 10–14, 2018, Rennes, France*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3274808.3274816>

1 INTRODUCTION

Caching is a well-known technique for boosting system's performance, by maintaining a relatively small part of the data in a fast nearby memory known as a *cache*. In this work we are interested in *software caches*, i.e., caches that are maintained by software systems such as middleware, operating systems, file systems, storage systems, and databases, rather than *hardware caches* that are implemented in hardware, such as the CPU's L1, L2, and L3 caches. Repeated access to a data item that is already stored in the cache,

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.

Middleware '18, December 10–14, 2018, Rennes, France

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-5702-9/18/12...\$15.00

<https://doi.org/10.1145/3274808.3274816>

known as a *cache hit*, is served much faster than fetching the data from its actual storage. All other accesses are called *misses*. Deciding which items should be placed in the cache is the role of the *cache management policy*. Obviously, the holy grail of cache management is guessing which items would yield the highest *hit ratio*, i.e., the ratio between cache hits and the overall number of accesses. Typically, such schemes try to identify patterns in the workload that would serve as signals of how to obtain the highest possible hit ratio.

Recency and *frequency* are the two most common signals used by software cache management policies. Recency captures the likelihood that an item that was *recently* accessed will be accessed again in the near future. In contrast, frequency captures the likelihood that an item that has been accessed *frequently* will be accessed again in the near future. Since in most workloads items' popularity changes over time, usually, the frequency is measured w.r.t. some aging mechanism such as a *sliding window* [17] or an *exponential decay* [13, 19].

Empirically, different workloads exhibit varying levels of recency vs. frequency, which is why designing a single "best" cache management scheme is an elusive goal. Hence, system designers are faced with the non-trivial task of understanding the characteristics of their workloads and then investigating which known cache management policy would provide the highest hit ratios for these workloads. Further, some cache management policies have several tuning parameters, which requires systems designers to understand how to configure them. Worse yet, when designing a new system, its future workloads might not be known *a priori*, so its designers cannot even tell which cache management policy to choose. Alternatively, in a general caching library, to relieve users from dealing with setup parameters, these parameters might be set to default values which provide "best" results for "most workloads". For other systems' workloads, such settings may imply far from optimal results.

In summary, there is a need for an adaptive software cache management policy that would obtain competitive hit ratios on as many varying workloads as possible. Exploring such adaptivity mechanisms for software caching is our focus.

Contributions

In this paper, we identify two adaptivity mechanisms for software cache management schemes that expose tuning parameters that impact their obtained hit ratio. The first mechanism is based on the

well-known *hill climbing* method [26] that we adjust to cache management tuning. Specifically, we periodically adjust the parameters in a certain *direction* in terms of configurations that work better for recency biased workloads vs. frequency biased ones. After some time, the newly obtained hit ratio is compared with the previously measured one. If a noticeable improvement is obtained, then the tuning parameters are further adjusted in the same direction. Otherwise, the tuning parameters are adjusted in the opposite direction, and this goes on repeatedly. The main advantage of hill climbing is that it can be implemented without introducing any meta-data. Yet, it runs the risk of getting stuck in a local maximum, and it never stops oscillating.

The second approach we examine is a novel *indicator* based solution. Here, we periodically calculate an estimation of the workloads' frequency vs. recency bias and adjust the tuning parameters accordingly in a single swoop. This approach requires some meta-data to calculate the above estimation but converges quickly.

We apply these two approaches to the recently introduced *FRD* [25] and *W-TinyLFU* schemes [13]¹. Both schemes split the overall cache area into two sub-caches, one for “newly arriving items” and the other for more “frequent items”. Both schemes offer the division between these sub-caches sizes as a tuning parameter but differ in the exact rule for deciding which item should go into which sub-area. In *W-TinyLFU*, this decision is based on an *admission filter* implemented through a space-efficient sketch [11], whose frequency aging mechanism serves as another potential area for adaptation.

Here, we explore ways of dynamically adapting the parameters of *FRD* and *W-TinyLFU* to the online workload using the hill climbing and indicator methods mentioned above. Specifically, we examine adjusting the relative size of the two sub-cache areas for both *FRD* and *W-TinyLFU*. For *W-TinyLFU*, we also examine manipulating the aging parameters of the sketch employed by its admission filter.

Finally, we evaluate our proposed schemes against a large set of traces from various sources, some proposed by authors of previous works and some offered by Caffeine users [23]. In our evaluation the best of our adaptive schemes are always competitive with the best alternative.

2 RELATED WORK

Belady's optimal cache management policy [5] looks into the future and evicts from the cache the entry whose next access is furthest away. This policy is impractical in most domains as we cannot predict the future. It serves as a useful upper bound for cache policies' performance. In practice, cache management policies involve heuristics and optimizations for typical access patterns.

Least Recently Used (LRU) [14] is based on the assumption that the least recently used item is also the least likely to be used in the future. Thus, once the cache is full, it evicts the LRU item to make room for the newly arriving item. Alternatively, the *Least Frequently Used (LFU)* algorithms assume that access frequency is a good estimator of future behavior [2, 3, 13, 17, 18]. Realizing LFU requires monitoring a large number of items that are no longer in

¹ W-TinyLFU is implemented as part of the open source Caffeine Java 8 cache [23]. W-TinyLFU was adopted, either directly or through Caffeine in multiple storage systems, including Cassandra, LinkedIn's feed, VMWare's Corfu, RedHat's InfiniSpan, Apache Accumulo, Allegro, Amplitude, ScalaCache, druid.io, neo4j, Mangocache, and others.

the cache, which incurs significant overheads. The work of [13] minimizes these overheads by monitoring past frequencies using an approximate sketch. Moreover, LFU policies often employ aging, where the frequency is calculated with respect to a specific sliding window [17] or an exponentially decaying sample [2, 3]. Other heuristics include reuse distance [1, 16, 20] to account for the time between subsequent accesses to the same entry. Alas, the latter requires remembering a large number of *ghost* entries.

Size is another dimension for optimization. In some cases, items significantly differ in size, and thus size should be taken into account [6, 9, 32]. E.g., *AdaptSize* dynamically adjusts the cache admission probability to maximize the object hit ratio [6]. In contrast, many cache policies maintain a fixed amount of items regardless of size. This is effective when cached items have equal or similar size, such as in the case of block caches and page caches. Further, most popular video-on-demand and streaming systems break files into (nearly) equal sized chunks (or stripes), each of which can be cached independently. In this work, we target fixed-sized items.

Hyperbolic caching [7] is a recent proposal for an overall best cache management policy. Hyperbolic caching can mimic multiple eviction policies with the same internal data structures thereby adapting its actual behavior to the workload. Another adaptive policy is *Adaptive Replacement Cache (ARC)* [24]. The drawback of both these policies is that they do not competitively capture frequency biased traces (as we show in our evaluation). Also, ARC requires maintaining a large number of ghost entries.

LRFU [19] rates each item by combining its recency and frequency, whose balance is controlled by a parameter λ . While instantiating LRFU with the “right” λ value for a given workload may yield high hit ratios, choosing a “wrong” value results in poor performance. Automatically adjusting λ to the workload is an open issue. Another issue that prevents LRFU's broad adoption is its inherent high implementation and runtime costs.

The recently introduced Mini-Sim [31] approach suggests simulating multiple possible cache configurations concurrently. For each simulation, a random sample of all accesses is processed to reduce computation overheads. The challenge with this approach is that naïvely sampling 1 out of r accesses is unlikely to capture the characteristics of recency biased workloads as it is likely to miss multiple accesses within a short period (here $1/r$ is the *sampling ratio*). Therefore, Mini-Sim samples random $1/r$ keys and then feeds all accesses for a sampled item to the simulated configurations. To save space, the simulated cache size is set to be c/r where c is the original cache size. Recall that the goal here is to find the best configuration parameters rather than determining which items should be in the actual cache.

Mini-Sim has the following drawbacks. The simulated configurations must hold ghost entries for their stored items, which consume storage space and execution time proportional to the number of configurations simulated concurrently and the sampling ratio. We experimented with applying the Mini-Sim approach to W-TinyLFU and discovered that on caches of up to several thousands of items as well as on frequency biased workloads, Mini-Sim requires a large sampling ratio to yield accurate results. In particular, sampling based on items' IDs is less accurate on frequency biased workloads since if none of the frequent items is sampled, the results of Mini-Sim will mispredict the actual workloads' behavior. Also, because

of sampling, Mini-Sim takes a relatively long time to adapt to significant changes in the workload. In contrast, our approach is more risk-taking in the sense that we change the configuration without first simulating the performance of the new configuration. Our risk-taking approach avoids most of Mini-Sim's overheads.

The recent LeCaR work studied utilizing machine learning techniques for combining LRU and LFU replacement in an agile way [30]. The resulting method was shown to be superior to ARC and reacted well to changes in the workload.

Several works have considered the unique aspects of SSD caching. Most importantly, the work in [8] describes why Belady's notion of optimality is insufficient for SSDs and proposes a new optimal offline algorithm whose goal is not only maximal hit ratio but also taking into account container placement issues to reduce write amplification and device wear. Pannier [21] focuses on the technical details of implementing a container cache for SSD. Pannier utilizes an adaptive credit system that limits the amount of SSD writes to a predefined quota. Closest to our work is [15], which maintains an explicit frequency statistics of objects using ghost entries and only admits popular objects into the SSD cache, with ARC as the cache replacement mechanism. This approach is similar to TinyLFU in essence but utilizes ghost entries and not sketches.

In the area of hardware CPU caches, closest to our work is [28], which proposes an adaptive mechanism for switching between different replacement policies (LRU, LFU, FIFO, Random), but not for tuning any configuration parameters.

3 AN OVERVIEW OF W-TINYLFU AND FRD

As hinted above, W-TinyLFU includes three components: the *Main cache*, an approximated LFU based *admission filter* called *TinyLFU*, and a *Window cache*, as illustrated in Figure 1. Newly arriving items are inserted into the Window cache. In principle, it can be maintained using any known policy, but in the Caffeine implementation, the Window cache uses LRU eviction [23]. Similarly, the Main cache may employ any cache management scheme, but in Caffeine it is implemented with SLRU eviction. The cache victim of the Window cache as well as the would be cache victim of the Main cache are then presented to the TinyLFU filter. The latter estimates the frequency of both and chooses to insert/keep the item with the highest estimate in the Main cache.

The size of the Window cache in W-TinyLFU can be fixed to any relative portion of the total cache size from 0 – 100%. The authors of W-TinyLFU [13] report that for the vast number of workloads they have experimented with, up to 1% offered the best hit-ratios, which is why this is the default configuration of the Caffeine cache library [23]. Yet, in some significantly recency biased traces, the size of the Window cache had to be increased to as much as 20 – 40% to match the best alternative scheme, typically either LRU or ARC.

The frequency estimation held by the *TinyLFU* admission filter is implemented through a sketch, either a *minimal increment counting Bloom filter* [10], or *count min sketch* [11]. The frequency statistics are counted over a long sample whose length S is a multiple of the cache size C . To age its entries, all counters are divided by an *aging factor* once every S accesses, an operation called *Reset* in [13]. By default, the aging factor is 2, i.e., all counters are halved on each Reset operation. Further, since items whose frequency is at

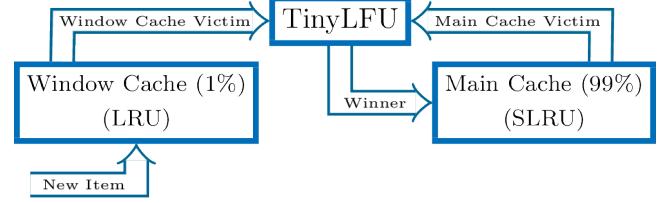


Figure 1: W-TinyLFU scheme: Items are first always admitted to the Window cache and the victim of the Window cache is offered to the Main cache, which employs TinyLFU as its admission filter. We preserve here the terminology of [13].

least S/C are always frequent enough to remain in the cache, the counters' maximal values are capped at S/C .

FRD [25] also divides the total cache area into two sections called *Temporal cache* and *Actual cache*, each managed by LRU. FRD maintains a long history of accesses, somewhat similar to ghost entries in other modern schemes. On a cache miss, if the accessed item appears in the stored history, it is inserted into the Actual cache. Otherwise, the new item is inserted into the Temporal cache. Similarly to W-TinyLFU, the ratio between the Temporal cache and Actual cache is a tuning parameter. Based on measurements against multiple traces, the authors of [25] recommend a default setting of 10% for the Temporal cache and leave dynamically adapting this value is left for future work. Unlike W-TinyLU, FRD requires to maintain an extended access history.

4 ADAPTIVE CACHE POLICY

In this section, we suggest a couple of methods for deriving adaptive cache management policies. Specifically, we discuss an adaptation of the well-known *hill climbing* algorithm [26] to the context of caching as well as our novel *indicator* based adaptive scheme. To make the presentation concrete, we first present the tuning parameters these schemes adjust in both W-TinyLFU and FRD in Section 4.1. Yet, one can carry over the ideas to any scheme that exposes tuning parameters for better handling of recency vs. frequency biased workloads. The adaptive schemes are then discussed in Section 4.2.

4.1 The Tuning Parameters

For both W-TinyLFU and FRD, we explore dynamically adjusting the relative size of the window (or Temporal) cache between 0–100% of the total cache size. In the case of W-TinyLFU, we also examine dynamically calibrating the sketch parameters used by the TinyLFU admission filter itself that affect the speed of the frequency aging process. We elaborate on these below.

4.1.1 The Window Cache Size. When the eviction policy of the window (Temporal) cache is LRU, as in Caffeine and FRD, a large window (Temporal) cache will yield an overall behavior closer to LRU. In particular, when the size of the window (Temporal) cache is 100%, we obtain exactly LRU. In contrast, since the W-TinyLFU admission filter is based on the aged frequency of items, a small

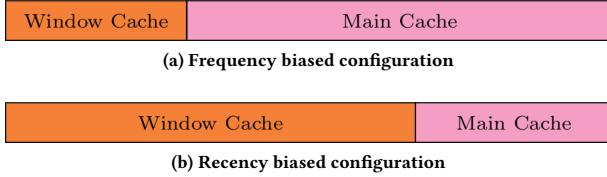


Figure 2: Tuning W-TinyLFU and FRD: the partition between window (Temporal) cache and main (Actual) cache implies a trade-off between recency and frequency.

Window cache tilts the emphasis towards frequency. Similarly, in FRD the Actual cache is used to store frequent items. That is, if we scale the main (Actual) cache closer to 100% of the total cache size, we get a frequency oriented policy in both W-TinyLFU and FRD. Figure 2 exemplifies this.

4.1.2 The W-TinyLFU Sketch Parameters. The W-TinyLFU admission filter has three configuration parameters: (i) the aging sample length S , (ii) the aging factor, which by default is 2 on each Reset operation but can be made larger or smaller, and (iii) the frequency increment on each item's access, which by default is 1 but can also be adjusted.

We start with the sample length S . By shortening S , we limit the maximal frequency count of items. Such limitation reduces the maximal frequency based difference between items and also expedites their aging process since their maximal value is lowered. Both reduce the impact of frequency and make the filter more recency biased.

Next, increasing the aging factor used to divide counters in Reset operations yields faster aging and vice versa. I.e., increasing this factor tilts the admission filter towards recency while decreasing it favors frequency.

Finally, we claim that enlarging the counters increment on each item's access from 1 to a larger value also favors recency. This is because a large increment quickly brings the items counters' to their maximal value, meaning that the counters' values reflect more how recently each item was accessed than how many times it was accessed². Moreover, we invoke Reset once the total increments reaches the sample size so Reset operations are executed per fewer items and aging happens quicker. Hence, the impact of historical frequency counts rapidly diminishes in favor of recent activity.

For example, consider an increment value of 8. The counters representing an arriving item are increased by 8, reflecting a frequency estimation of 8. After a single Reset operation, the frequency estimation is halved to 4, after a second Reset these counters are halved to 2, etc. At the extreme, when the increment is set to S/C , W-TinyLFU becomes very recency biased. The impact of these manipulations is explained in Figure 3.

The motivation for changing the increment rather than the sample length or reset value comes from an engineering perspective. That is, a division by two is efficiently implemented with shift operations and dividing by another factor would require floating point arithmetics. Additionally, changing the size of the sketch would

² Notice that in counting Bloom filters and count-min sketch an item's value is represented by multiple counters.

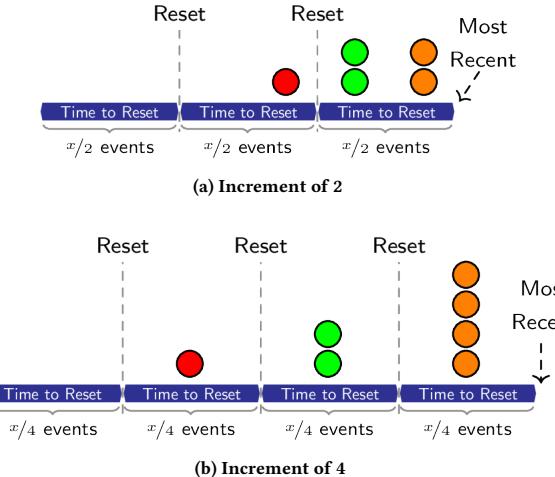


Figure 3: A scenario where 3 items appear only once at different times. The default W-TinyLFU with increment 1 grades all 3 items either as 1 if they happened after the last Reset or as 0 if they happened before the last Reset. Thus, it cannot determine which is more recent. In Figure 3a we use an increment value of 2, which becomes 1 following a Reset operation. Reset operations are also twice as frequent, which enables one to determine that the orange and green items appeared more recently than the red one. As there is no Reset operation between the orange and green, we cannot determine their order. Figure 3b increases the increment to 4, which again doubles the number of Resets. In this case, there is an additional Reset between the orange and the green so we can detect that orange is more recent than green.

either change the accuracy of the sketch or require dynamic memory allocation. In contrast, changing the increment value is less intrusive and is easier to implement in practice.

4.2 Potential Adaptivity Schemes

4.2.1 Hill climbing. Hill Climbing is a simple optimization technique for searching a local maximum of a function. In our context, we first change the configuration in a certain direction, e.g., enlarge the Window cache size. Then we compare the hit ratio obtained under the new configuration to the previously recorded hit ratio. If the hit ratio has improved we make an additional step in the same direction, e.g., continue increasing the window size. Otherwise, we flip directions and make a step backward. The hill climber algorithm is visually explained in Figure 4.

The main benefit of this approach is that we do not need additional meta-data to reconfigure the cache. In contrast, previous adaptive algorithms require meta-data and ghost entries [4, 24]. Hill climbing is a general meta-algorithm and can be used with many cache policies.

In this algorithm, we first change the cache configuration in a certain direction and then evaluate the impact on performance. That is, we take a risk as we do not know if the change benefits the hit ratio or not. The difficulty in realizing this method is determining

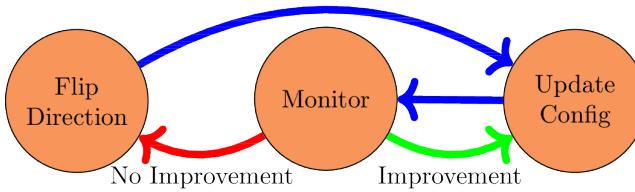


Figure 4: The hill climber algorithm: In the monitor state, compare the current hit ratio to the previously obtained one. If the hit ratio improves then we update the configuration in the same direction. Otherwise, we flip directions and update the configuration accordingly.

how large each step should be and how frequently to take such a step, which turns out to be a balancing act.

At first glance, frequent small steps seem appealing. This is because the penalty for a wrong step is small and the reaction to changes is quick. The problem with such an approach is that measuring the hit ratio over a short duration is a noisy process. Hence, with frequent steps, it is difficult to distinguish between a change in the hit ratio that was caused by the new configuration and noise.

This observation led us to make infrequent and relatively large changes to the policy. Infrequent changes provide us with enough time to evaluate their effectiveness, and making them relatively large makes the change in hit ratio more noticeable. In our implementation, we chose to make steps of 5% to the Window cache size or of ± 1 to the increment size. This means that we alternate between 21 possible configurations when adapting the window size (i.e., 0%, 1%, 5%, 10%, etc.) and 15 possible configurations when adapting the sketch parameter (note that the maximal value of counters in our W-TinyLFU sketch is 15). Also, we chose a decision interval of once every 10 times the cache size; this works well empirically as it allows enough time to evaluate the performance change of the new configuration.

4.2.2 Frequency-Recency Indicator. Our goal here is to obtain an indicator that reflects the recency-frequency balance of the trace at any point in time. Such an indicator selects the proper configuration directly, rather than gradually taking incremental steps.

The indicator uses the same sketch as the W-TinyLFU filter [13] for estimating items frequency with an increment value fixed at 1. Thus, when the tuning parameter is the Window cache size of W-TinyLFU rather than the sketch increment, the indicator method shares exactly the same sketch with the W-TinyLFU filter.

For each newly arriving item, we sum the sketch estimations and use the average of these estimations as a *hint*, which provides an indication for the bias of the trace. Specifically, we empirically discovered that low values indicate a frequency biased workload, higher values indicate a recency bias, while very high values once again indicate a frequency bias of a very frequency skewed workload. Intuitively, if the workload is frequency biased, accesses to the same item are scattered, leading to relatively low counter values. In contrast, when the workload is recency biased, accesses to the

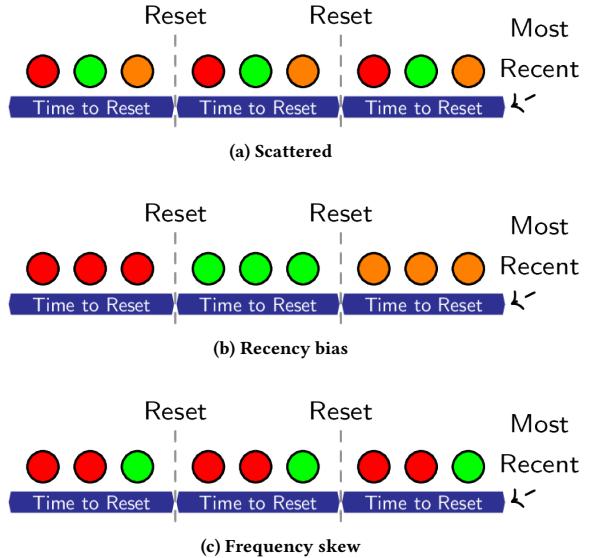


Figure 5: A scenario where each period contains 3 accesses. In Figure 5a each item appears once in each period. Due to Reset operations, the frequency estimations are always 0, so the hint equals 0. In Figure 5b the accesses are reordered in a recency manner. Now the estimations are 0, 1, 2 for each item. Hence, the hint equals 1. In Figure 5c we changed the number of times each item appears to illustrate frequency skewness. The estimations become 0, 1, 0 for the first period and 1, 2, 0 for the others and the hint becomes 0.77.

same item tend to appear near each other, leading to higher average counter values for accessed items. Finally, when the workload is frequency biased with a large frequency skew, a small subset of items are accessed very frequently, so the hint becomes very high. Figure 5 exemplifies this intuition.

To distinguish the large frequency skew effect from the high recency bias effect, we apply another mechanism to estimate the distribution skew. It is well known that when plotting Zipf distributed items in a log-log plot whose axes are an item's rank and its frequency, the result is a straight line whose slope is the frequency skew parameter. During each configuration period, we gather the k most frequent items and compute the skew estimation by performing a linear regression over the log-log values of these items. This calculation is performed infrequently (e.g., once every hundred multiples of the cache size), which makes it practical.

We combine the hint and the frequency skew estimation (*skew*) to forge our desired indicator as follows:

$$\text{indicator} \triangleq \frac{\text{hint} \cdot (1 - \min\{1, \text{skew}^3\})}{\text{maxFreq}}$$

where *maxFreq* is the maximum estimation we can get from the frequency sketch (15 in our settings). Our goal is to end up with a value close to 1 only if a high hint value appeared as a result of a recency bias. Thus, high values of the hint will be canceled out by high values of the skew if there is a high-frequency bias. As the skew is between [0,1], raising it by a power of 3 drastically separates

Adaptivity	Parameter		
	Window Cache Size		Sketch Increment
Hill Climber	WC-W-TinyLFU	WC-FRD	SC-W-TinyLFU
Indicator	WI-TinyLFU	WI-FRD	SI-W-TinyLFU

Table 1: Algorithm variants suggested in this work

skew values close to 1 from lower values. The normalization by \maxFreq gives us a value in $[0, 1]$, where 0 indicates a frequency bias while 1 indicates a recency bias.

5 EVALUATION

In this section, we perform an extensive evaluation of our adaptivity mechanisms (hill climber and indicator) and the adaptation parameters (window cache size and sketch management). We also evaluate two underlying caching policies W-TinyLFU and FRD, which were presented in Section 3. Overall, most of our work spans the 6 different possibilities that are summarized in Table 1: Adapting the Window (W) cache of W-TinyLFU based on the hill climber (C) and indicator (I) approaches, yielding **WC-W-TinyLFU** and **WI-W-TinyLFU** respectively. Adapting the Window (W) cache of FRD based on the hill climber (C) and indicator (I) approaches, yielding **WC-FRD** and **WI-FRD** respectively. Last, adapting the W-TinyLFU sketch (S) parameters based on the hill climber (C) and indicator (I) approaches, yielding **SC-W-TinyLFU** and **SI-W-TinyLFU** respectively. We also report on experimenting with applying the Mini-Sim approach [31] to adapting the window size of W-TinyLFU, yielding **MS-W-TinyLFU**.

Our experiments are performed in Caffeine’s simulator [23]. The implementation of the competitor policies is taken from that repository. In each experiment, each policy gets an instance and we feed that instance the entire workload without warm-up periods.

5.1 Adaptivity Configurations

Below we describe the different configurations considered by the hill climber and the indicator techniques.

Hill Climber. Hill climber performs an adaptation step once every $10 \cdot \text{cacheSize}$ accesses. It starts from 1% Window (or Temporal) cache and considers all configurations in $[0\%, 1\%, 5\%, 10\%, \dots, 80\%]$ for the Window cache, and in $[0\%, 1\%, 5\%, 10\%, \dots, 100\%]$ in the Temporal cache. For adjusting the W-TinyLFU sketch, we consider increment values of $[1 - 15]$, i.e., 15 configurations overall. Recall that 1% is the initial configuration of [13]. Yet, the authors of [25] recommend setting the Temporal cache size to 10%; it is interesting to see how our adaptive schemes that start from a different configuration fare compared to the recommended one.

Indicator. Indicator takes an adaptation decision once every 50,000 accesses. For configuring the sketch, we use $[30 \cdot \text{indicator}]$ at a time with a maximum value of 15. Since the increment of the sketch has 15 possible values, by multiplying the indicator by

30, an indicator value of 0.5 already sets the increment to 15. The reason for this is that the affect of the sketch parameter change is limited compared to changing the window size, so we multiply the indicator value by 2 when it is used to adjust the sketch increment. When configuring the Window and Temporal cache size, we use $80 \cdot \text{indicator}$ and $100 \cdot \text{indicator}$ respectively. Particularly, we allocate at most 80% of the total cache size to the Window cache and at most 100% to the Temporal cache at a time.

5.2 Traces

Our evaluation focuses on 14 real life workloads from diverse domains that include databases, analytic systems, transaction processing, search engines, Windows servers and more. The underlying characteristics of these vary considerably. Some are very frequency biased while some contain a mix of frequency and recency and some are almost entirely recency biased. Clearly, with such a diverse selection, no single configuration is suitable for all workloads. We now list the traces used in this work:

- **OLTP:** A trace of a file system of an OLTP server [24]. 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 blocks accesses sprinkled with a few random accesses due to an occasional write replay or in-memory cache misses.
- **F1 and F2:** Traces of transaction processing taken from two large financial institutions. The traces are provided by the UMass trace repository [22]. These are fairly similar in structure to the OLTP trace for the same reasons mentioned above.
- **Wikipedia:** A Wikipedia trace containing 10% of the traffic to Wikipedia during two months starting in September 2007 [29].
- **DS1:** A database trace taken from [24].
- **S3:** A search engine trace taken from [24].
- **WS1, WS2, and WS3:** Three additional search engine traces taken from the UMass repository [22].
- **P8 and P12:** Windows server disc accesses [24].
- **SPC1-like** A synthetic trace created by the authors of ARC [24] to benchmark storage systems.
- **Scarab:** A one-hour trace from Scarab Research of product recommendation lookups for several e-commerce sites of varying sizes worldwide.
- **Gradle:** A trace from a distributed build cache that holds the compiled output so that subsequent builds on different machines can fetch the results instead of building anew. Due to machines leveraging local build caches, the distributed cache is recency-biased as only the latest changes are requested. This trace was provided by the Gradle project.

5.3 Motivation

We start by showing that for the FRD and W-TinyLFU policies, there is no single static configuration of its tuning parameters that is attractive for all traces. Specifically, Figure 6 illustrates the obtained hit ratios with FRD and W-TinyLFU when varying the

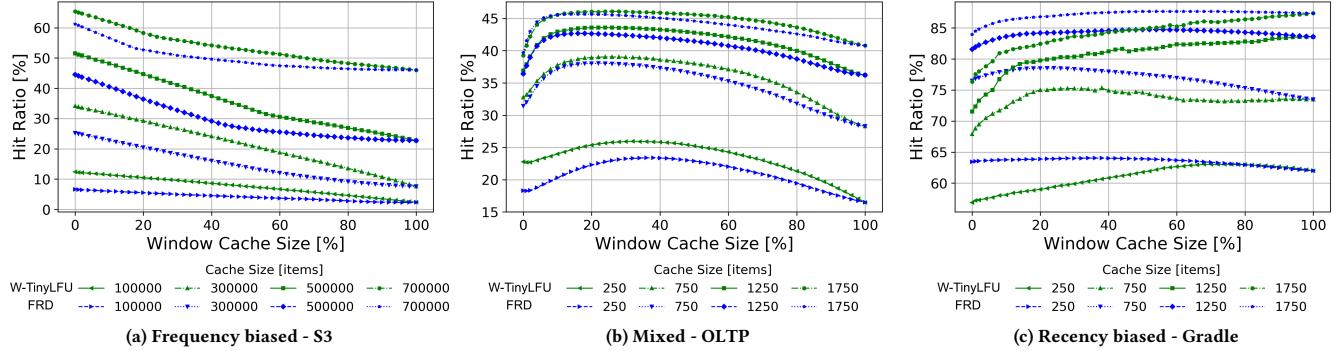
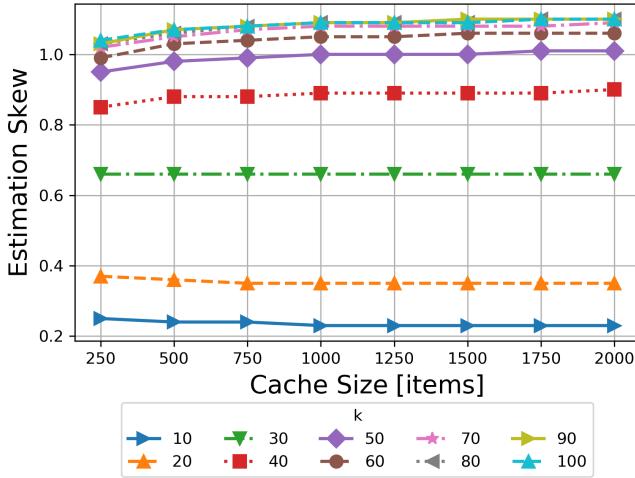


Figure 6: Evaluation of W-TinyLFU and FRD for different Window (Temporal) cache sizes

Figure 7: Skew estimation for different k 's on the Wikipedia trace (as in 5.2). Empirically, the estimation seems to converge and the benefit from taking $k > 70$ is insignificant.

total cache size and the relative size of the Window (Temporal) cache. As can be observed, for the search engine queries the graphs are monotonically decreasing. That is, for these traces, the smaller the Window (Temporal) cache size the better. In contrast, the Gradle graph favors large Window (Temporal) cache sizes and the larger the better. In the middle, there is the OLTP trace. In this trace, there is a maximum which is different for each total cache size (around 15%-30%). Thus, every static selection of parameters favors some traces over the others.

5.4 Indicator configuration

As mentioned above, estimating the skew parameter requires performing linear regression for the k most frequent items. Intuitively, large k values mean more accurate skew estimations but impose increased computations. The default value for k was selected empirically so that it provides reasonable estimations in all tested traces. Figure 7 shows the results for different K values in the Wikipedia

trace. We selected $k = 70$ since increasing k beyond 70 does not yield significantly better estimations.

5.5 Evaluation of the six algorithms

We evaluate here our six proposed algorithms to determine which is the best. We split the algorithms according to the configuration parameter they adapt. That is, we compare algorithms that dynamically adjust the Window (Temporal) cache size (WC-W-TinyLFU, WI-W-TinyLFU, WC-FRD, and WI-FRD) with those that dynamically change the sketch increment (SC-W-TinyLFU, SI-W-TinyLFU). For each of these parameters, we calculate the “offline” optimal value as a reference benchmark. This calculation is obtained by selecting the best static configuration for each data point separately, i.e., independently for each workload and for each cache size.

5.5.1 Dynamically configured Window cache size. Figure 8 shows the results for the dynamically configured Window cache size. The line labeled “offline” is the point-by-point best static configuration for the given trace, W-TinyLFU(1%) is the default configuration of Caffeine [13] and FRD the recommended configuration (10%) in [25].

As shown, in the Windows server (Figure 8a) and in the search engine trace (Figure 8b), all W-TinyLFU based schemes are nearly optimal. The adaptive FRD schemes match the recommended one, but all under-perform compared to the W-TinyLFU based ones. In the Wikipedia trace (Figure 8c) all schemes are almost identical.

The database trace (Figure 8d) shows a similar story, i.e., W-TinyLFU based schemes obtain better hit ratios than FRD based ones. Among W-TinyLFU schemes, WC-W-TinyLFU is slightly worse for large caches and WI-W-TinyLFU is slightly worse for small caches. The adaptive FRD schemes manage to meet the performance of its recommended configuration.

In the transaction processing trace (Figure 8e) the default configuration of W-TinyLFU is worse than the others. However, WI-W-TinyLFU is marginally better than the rest, meeting the “offline” results. A similar improvement is seen for the Gradle trace (Figure 8f), although here the dynamic FRD schemes are marginally better.

5.5.2 Dynamically configured sketch parameter. Figure 9 evaluates algorithms that modify the W-TinyLFU sketch parameters, namely

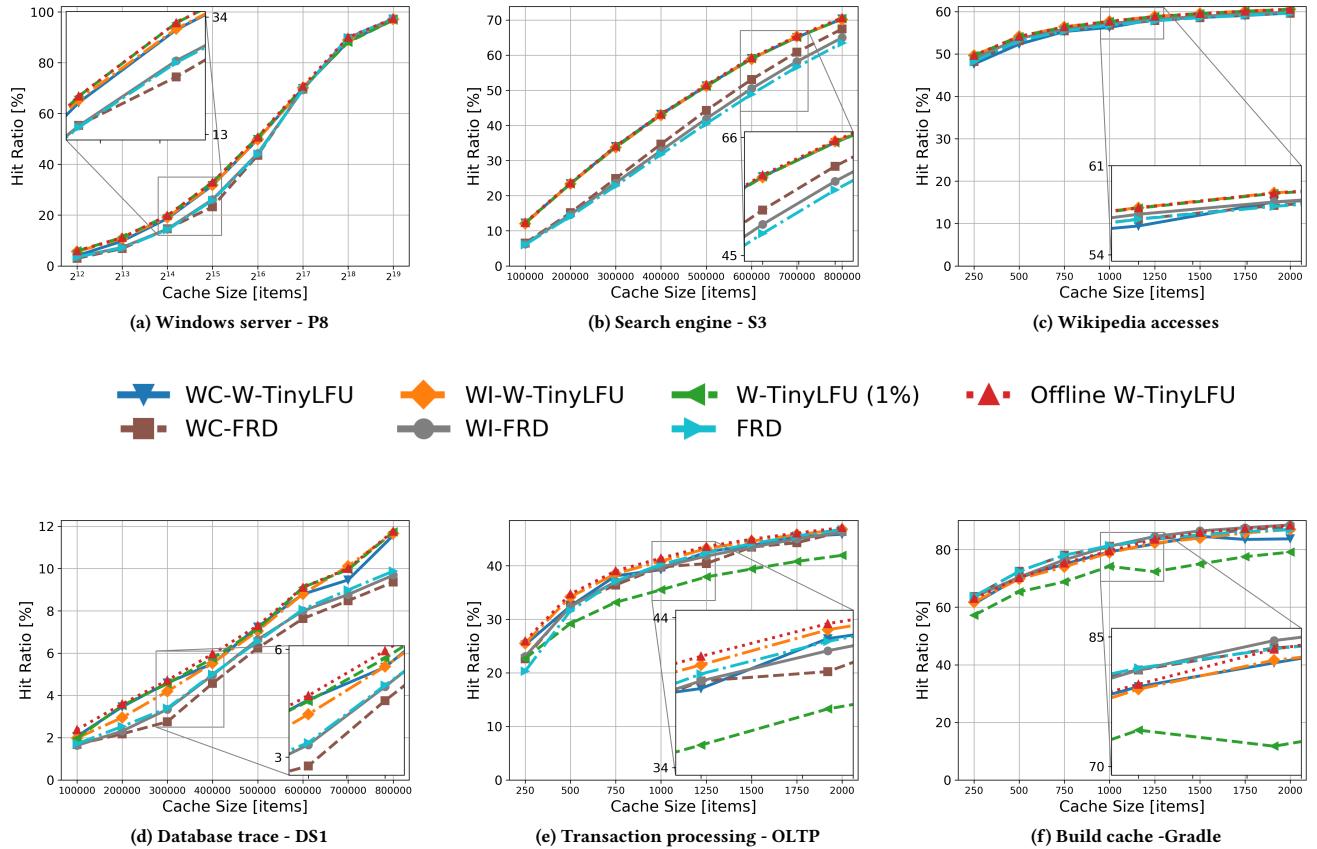


Figure 8: Evaluation of Window (Temporal) cache adjustments schemes compared to W-TinyLFU’s optimal static configuration

SI-W-TinyLFU and SC-W-TinyLFU. We compare these to their best offline configuration “Offline W-TinyLFU”.

For Windows servers (Figure 9a) and in the search engine trace (Figure 9b), all obtain nearly identical results. In the database trace (Figure 9c) SC-W-TinyLFU is slightly worse. In OLTP (Figure 9d), Wikipedia (Figure 9e), and Gradle (Figure 9f), all schemes are very similar.

In summary, it is unclear which adaptation policy works best: hill climbing or indicator. Yet, it seems that for W-TinyLFU adapting the Window cache size is always at least as good as adapting the sketch parameters. Thus we continue with the two algorithms WC-W-TinyLFU and WI-W-TinyLFU. The adaptive W-TinyLFU based schemes are either at par with the FRD based schemes or noticeably better, depending on the trace and maintain much smaller meta-data due to the sketch. The adaptive FRD schemes meet the performance of the recommended configuration even when starting at a non-recommended one (1%). This highlights the benefits of dynamic adaptivity: no need to manually explore all traces in order to detect the best configuration.

5.6 Comparison with other adaptation mechanisms

Next, we compare our adaption mechanisms with Mini-Sim [31]. To that end, we configured W-TinyLFU with 101 Mini-Sim instances, each simulating a different possible partition of Window and Main cache (denoted MS-W-TinyLFU).

As suggested in [31] we enforce S_m – the simulate cache size – to be ≥ 100 and the sampling ratio to be ≥ 0.001 . That is $S_m = \max(100, 0.001 \cdot C)$ where C is the tested cache size and the sampling ratio is $\frac{S_m}{C}$.

The decision interval was configured to be once in 1,000,000 accesses. Such a long interval empirically worked the best.

Figure 10 illustrates the results for this evaluation. As can be observed, in the frequency biased database trace (Figure 10a), WI-W-TinyLFU is superior to Mini-Sim, especially when the cache size is large. Mini-Sim also lags in the frequency biased search engine trace (Figure 10b). These results support our observation that Mini-Sim’s sampling method is less efficient in frequency biased traces. In other traces, such as the SPC1 benchmark (Figure 10c) and the F1 trace (Figure 10d) the Mini-Sim approach works just as well as WI-W-TinyLFU. In conclusion, while Mini-Sim is a viable solution

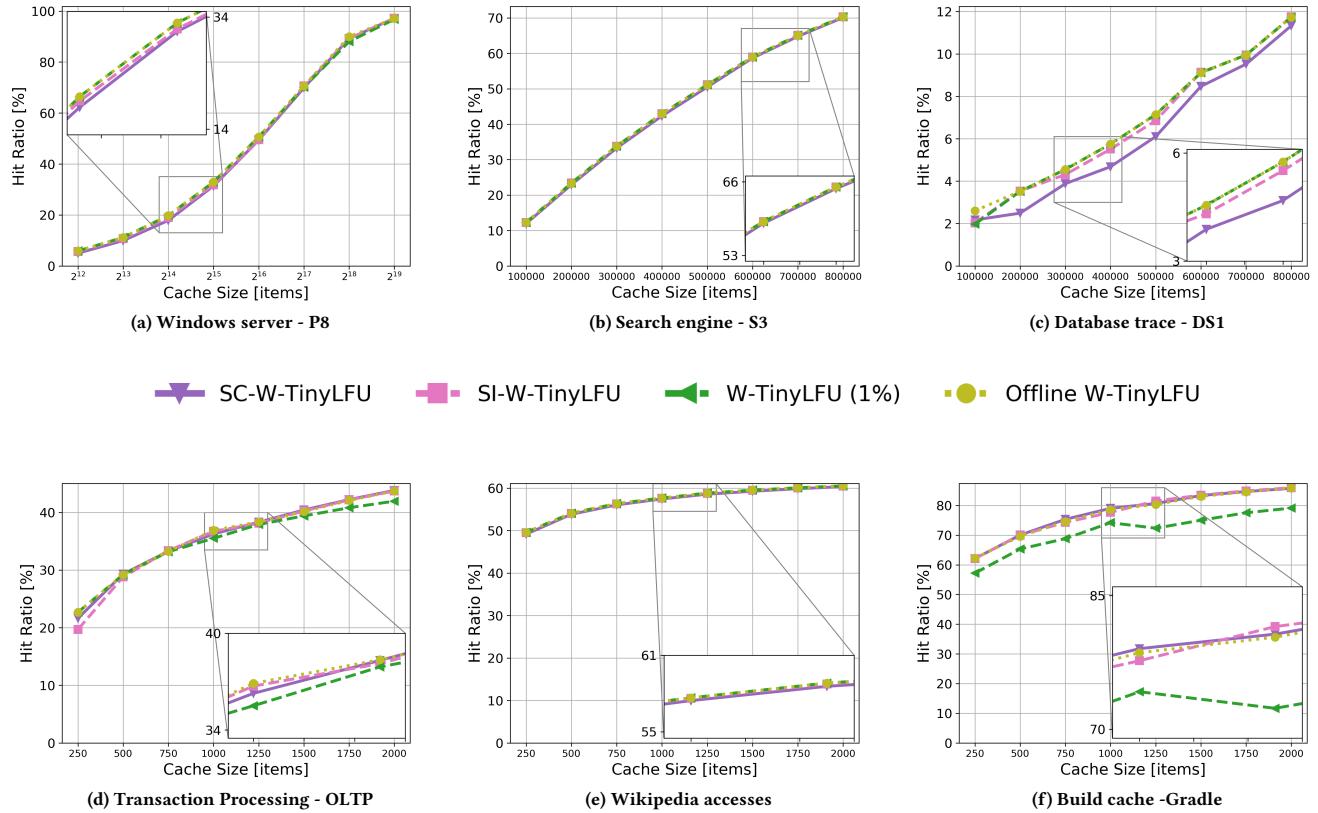


Figure 9: Evaluation of sketch's increment schemes compared to their offline optimal static configuration

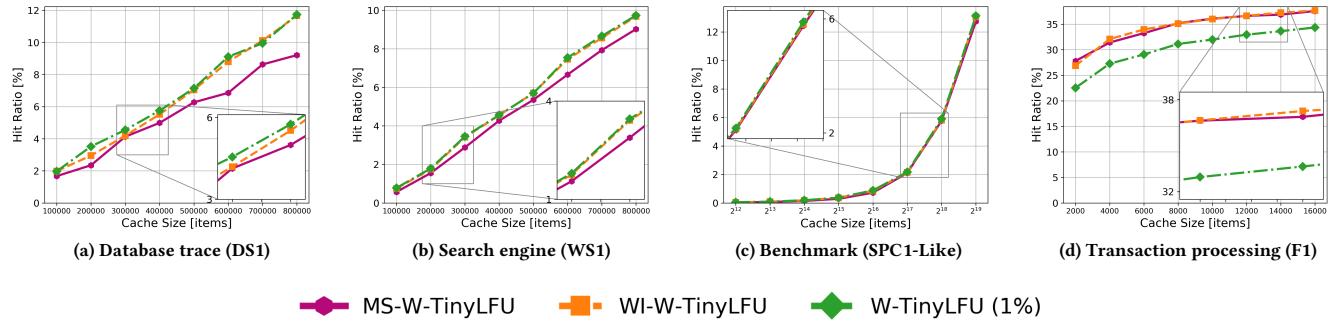


Figure 10: Comparative evaluation of Mini-Sim and our own adaptation techniques and the non-adaptive scheme

for adaptivity, it is lacking in frequency biased traces and small caches. In contrast, our own hill climber and indicator seem to provide a more robust adaptivity to the cache. Hence, we continue the evaluation only with WI-W-TinyLFU and WC-W-TinyLFU.

5.7 Comparative evaluation

In this section, we compare W-TinyLFU, WI-W-TinyLFU, WC-W-TinyLFU, and FRD to other leading works: ARC [24] and Hyperbolic caching [7].

Web search traces. Web search traces are typically very frequency biased, as the server sees an aggregation of searches from many

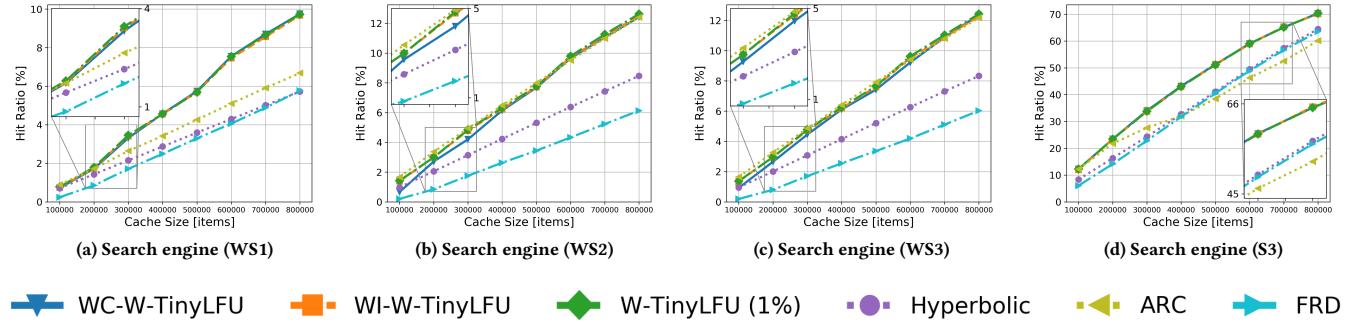


Figure 11: Comparative evaluation over web search traces

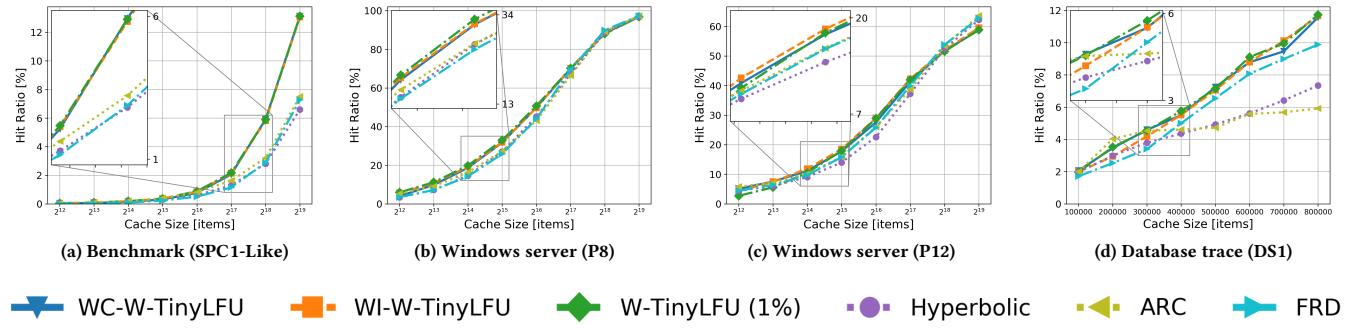


Figure 12: Comparative evaluation over storage traces

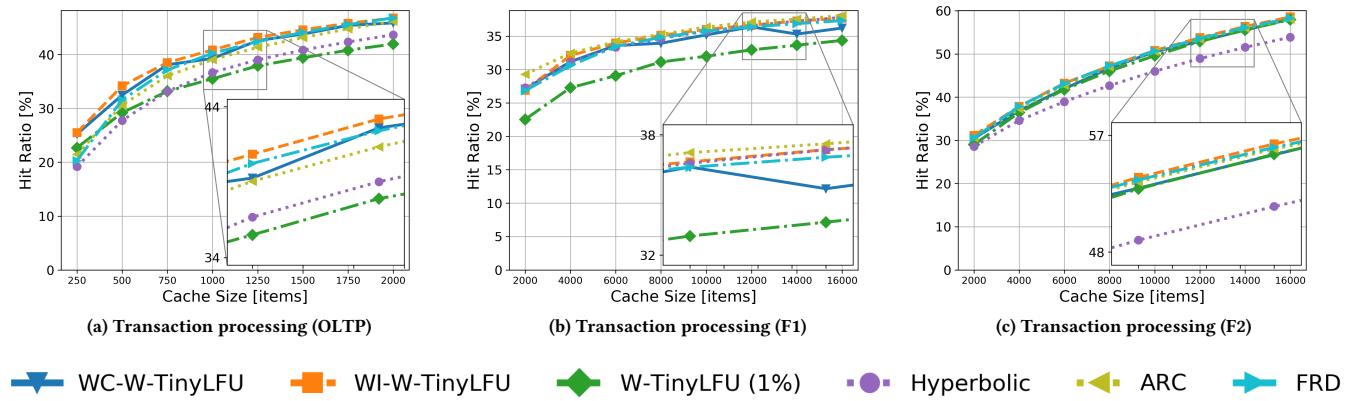


Figure 13: Comparative evaluation over transaction processing traces

users. Figure 11 shows a comparative evaluation over Web search traces. Observe that the W-TinyLFU based schemes are always at

the top. In contrast, FRD and hyperbolic caching are the worst for these workloads. ARC is competitive in the WS2 trace (Figure 11b)

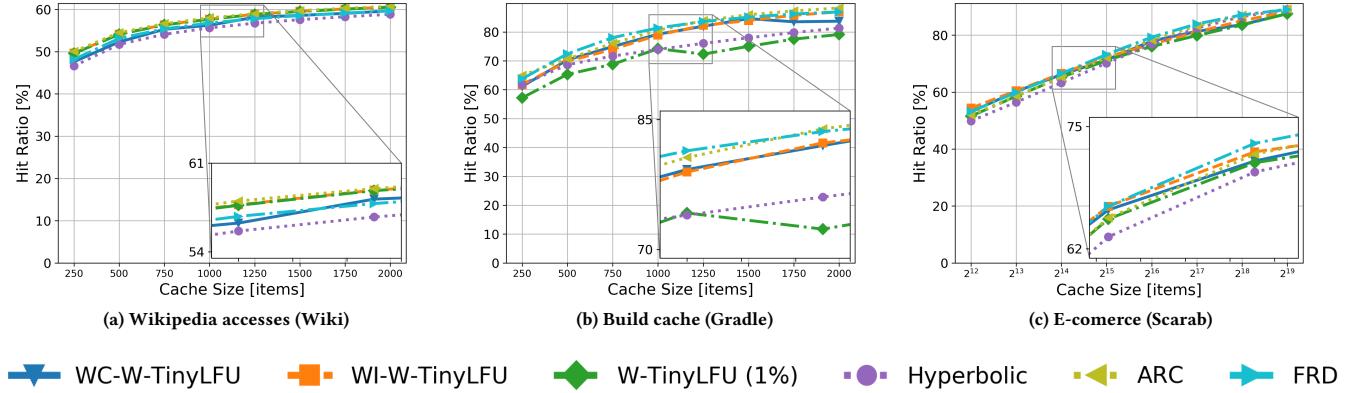


Figure 14: Comparative evaluation over other processing traces

and WS3 trace (Figure 11c) but lags behind in the WS1 trace (Figure 11a) and the S3 trace (Figure 11d). In these traces W-TinyLFU is the best and our adaptive policies exhibit very close performance.

Storage traces. Figure 12 shows results for a variety of storage traces, including Windows servers, a database and a benchmark. In the SPC1-Like benchmark trace (Figure 12a), WI-W-TinyLFU, WC-W-TinyLFU and W-TinyLFU are the leading policies. FRD, Hyperbolic and ARC are lagging.

In the Windows server traces P8 (Figure 12b) and P12 (Figure 12c), W-TinyLFU based schemes are better for smaller caches. For larger caches, in P8 all the policies yield the same hit ratio and for P12 there is a slight advantage of up to 2% for ARC, FRD, and hyperbolic caching. Overall, there is no clear winner in these traces.

In the database trace DS1 (Figure 12d), W-TinyLFU and WC-W-TinyLFU yield the highest hit ratio and are almost indistinguishable while WI-W-TinyLFU is slightly worse with FRD closely below. However, all W-TinyLFU based policies are considerably better than Hyperbolic caching and ARC.

Transaction processing. Figure 13 shows the evaluation over transaction processing traces. Such traces often exhibit a mixture recency and frequency. ARC is competitive in all these traces and the non-adaptive W-TinyLFU lags behind in the OLTP trace (Figure 13a) and in the F1 trace (Figure 13b). In contrast, our adaptive algorithms (WI-W-TinyLFU and WC-W-TinyLFU) and FRD are very similar to ARC. In all these traces our algorithms are either slightly better than ARC (OLTP and F2) or are slightly worse (F1) but the differences are below 1%. In contrast, Hyperbolic caching is only competitive in the F1 trace and W-TinyLFU only in F2.

Other traces. Figure 14 shows the evaluation for other types of traces including Wikipedia, Gradle, and Scarab. In the Wikipedia trace (Figure 14a), all schemes perform similarly with Hyperbolic caching being marginally worse.

For Gradle (Figure 14b) the default W-TinyLFU is the worst. Such traces are very recency biased as all requests are for the newest build. Thus, the fact that a certain build was requested many times does not indicate that it would be requested again in the future.

WI-W-TinyLFU and WC-W-TinyLFU close the performance gap and are usually at most 1% from ARC, Hyperbolic, and FRD.

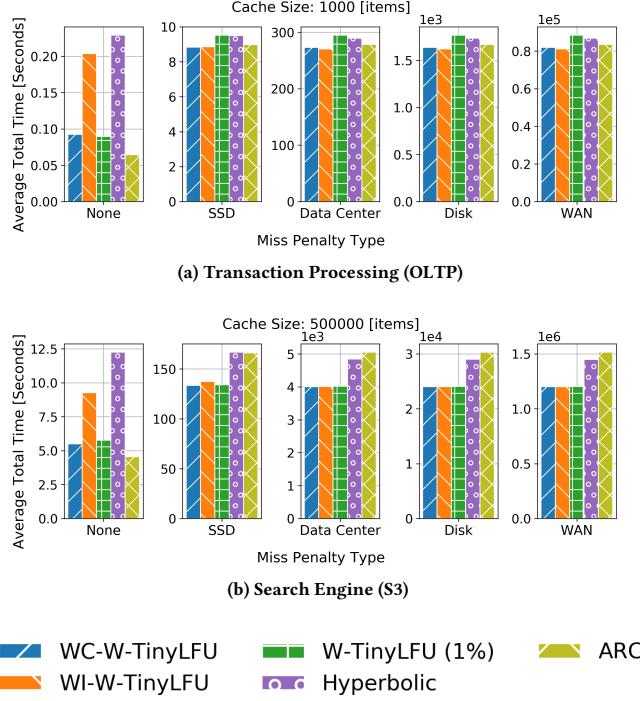
In Scarab (Figure 14c), Hyperbolic caching is the worst and all other policies achieve very similar hit ratios with at most 1% advantage to FRD.

5.8 Performance evaluation

Last, we evaluate the computational overheads of our adaptive policies. Specifically, we measure the completion time of real traces while simulating the delays incurred by serving cache misses from various main storage options. We considered miss latencies corresponding to an SSD access, a datacenter access, a disk access and a WAN (CA to Netherlands) access as reported in the benchmark project of [27]. We also examined a 0 miss penalty option, denoted ‘none’, which captures the nominal performance overhead of the cache management scheme. Intuitively, as long as the computational overheads are negligible compared to the miss penalty, the completion time would be dominated by the hit-ratio. However, if a policy is overly complex, it might incur a long completion time despite its high hit-ratio. The measurements were taken on a single core of an Intel i5-6500 CPU @ 3.20GHz.

Figure 15 shows result for this evaluation over the OLTP trace (Figure 15a) and the S3 trace (Figure 15b). From looking at the ‘none’ miss penalty bars, we observe that Hyperbolic caching is the most computationally intensive policy while ARC is the least computationally intensive one. WC-W-TinyLFU is almost identical to W-TinyLFU and WI-W-TinyLFU is a bit more computationally intensive. Further, all adaptive schemes demonstrated over 100 millions of cache accesses per second (dropped for lack of space).

Next, we examine the OLTP trace (Figure 15a) where WC-W-TinyLFU and WI-W-TinyLFU yield the highest hit-ratio. As can be seen, indeed these policies also offer the smallest completion time for all none zero delays. When we examine the S3 trace (Figure 15b), we note that W-TinyLFU has the highest hit-ratio. Indeed, it also has the lowest completion time. Note that our adaptive policies have very little impact on the completion time and thus preserve the good properties of W-TinyLFU in this case. Overall, our evaluation

**Figure 15: Effect of the miss penalty on execution time**

indicates that the calculation overheads of our adaptive policies do not hinder the performance even for a DRAM cache whose main storage is on SSD.

6 DISCUSSION

As been demonstrated in the past and echoed here, no single previously statically configured existing cache management policy yields the best hit ratio on every workload and for every cache size³. Hence, finding the best policy for a given system requires careful studying of its workloads and the performance obtained by various policies over these workloads. Yet, expecting developers to perform such a detailed research for each system may not be practical, and in particular, workloads may not be known *a priori*. In this work, we have investigated methods for dynamically adapting cache management policies to the recency vs. frequency bias of an imposed workload. Our efforts have focused on instantiating the well-known hill climbing method to caching and on a novel indicator based scheme. We have applied these methods to the recent W-TinyLFU and FRD policies [13, 25].

By examining the benefits from adjusting various parameters of these policies, we concluded that modifying the window size of W-TinyLFU yields more promising results than adapting its sketch parameters. Especially, the corresponding adaptive schemes, WC-W-TinyLFU and WI-W-TinyLFU (for window climber and window indicator respectively), were always competitive with the best state-of-the-art policies, and never degraded performance in cases

³ In fact, even existing adaptive policies such as ARC and Hyperbolic caching underperform noticeably on some workloads.

W-TinyLFU was already the winning policy. In the case of FRD, the adaptive schemes matched the performance of the recommended static configuration in [25] even when started from a non-optimal configuration. This highlights the fact that adaptivity can yield best results without knowing *a priori* the workloads and without the laborious task of trying different configurations with different workloads. In terms of CPU overhead, our evaluation showed that completion times are dominated by the hit-ratios and that the computational overheads of our schemes are negligible, whether the miss penalty comes from an SSD, a datacenter, a disk or a WAN access.

When comparing the hill climbing and the indicator approaches, the latter obtains slightly overall better results and adapts faster (in one step). Yet, hill climbing is simpler to implement and requires less space. Hence, in resource constrained environments, hill climbing may be preferable.

The entire code base used in this paper including the testing setup is available in open source at [12].

Acknowledgments. We thank the anonymous reviewers and in particular our shepherd Fred Douglis for their helpful comments and insight. This work was partially funded by ISF grant #1505/16.

REFERENCES

- [1] AKHTAR, S., BECK, A., AND RIMAC, I. Caching online video: Analysis and proposed algorithm. *ACM Trans. Multimedia Comput. Commun. Appl.* 13, 4 (Aug. 2017), 48:1–48:21.
- [2] ARLITT, M., CHERKASOVA, L., DILLEY, J., FRIEDRICH, R., AND JIN, T. Evaluating content management techniques for web proxy caches. In *In Proc. of the 2nd Workshop on Internet Server Performance* (1999).
- [3] ARLITT, M., FRIEDRICH, R., AND JIN, T. Performance evaluation of web proxy cache replacement policies. *Perform. Eval.* 39, 1–4 (Feb. 2000), 149–164.
- [4] BANSAL, S., AND MODHA, D. S. Car: Clock with adaptive replacement. In *Proc. of the 3rd USENIX Conf. on File and Storage Technologies (FAST)* (2004), pp. 187–200.
- [5] BELADY, L. A. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal* 5, 2 (1966), 78–101.
- [6] BERGER, D. S., SITARAMAN, R. K., AND HARCHOL-BALTER, M. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX Symposium on Networked Systems Design and Implementation NSDI* (2017), pp. 483–498.
- [7] BLANKSTEIN, A., SEN, S., AND FREEDMAN, M. J. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference (USENIX ATC)* (2017), pp. 499–511.
- [8] CHENG, Y., DOUGLIS, F., SHILANE, P., WALLACE, G., DESNOYERS, P., AND LI, K. Erasing belady’s limitations: In search of flash cache offline optimality. In *Proc. of USENIX Annual Technical Conference (ATC)* (2016), pp. 379–392.
- [9] CHERKASOVA, L. Improving www proxies performance with greedy-dual-size-frequency caching policy. Tech. rep., In HP Tech. Report, 1998.
- [10] COHEN, S., AND MATIAS, Y. Spectral bloom filters. In *Proc. of the ACM SIGMOD Int. Conf. on Management of data* (2003), pp. 241–252.
- [11] CORMODE, G., AND MUTHUKRISHNAN, S. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms* 55, 1 (Apr. 2005), 58–75.
- [12] EINZIGER, G., EYTAN, O., FRIEDMAN, R., AND MANES, B. Codebase and traces used in this work. <https://www.cs.technion.ac.il/~ohadey/adaptive-caching/AdaptiveSoftwareCacheManagementMiddleware2018.html>, 2018.
- [13] EINZIGER, G., FRIEDMAN, R., AND MANES, B. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (TOS)* (2017).
- [14] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture - A Quantitative Approach* (5. ed.). Morgan Kaufmann, 2012.
- [15] HUANG, S., WEI, Q., FENG, D., CHEN, J., AND CHEN, C. Improving flash-based disk cache with lazy adaptive replacement. *ACM Trans. on Storage (ToS)* 12, 2 (Feb. 2016).
- [16] JIANG, S., AND ZHANG, X. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proc. of the International Conference on Measurements and Modeling of Computer Systems SIGMETRICS* (June 2002), pp. 31–42.
- [17] KARAKOSTAS, G., AND SERPANOS, D. N. Exploitation of different types of locality for web caches. In *Proc. of the 7th Int. Symposium on Computers and Communications (ISCC)* (2002).

- [18] KETAN SHAH, A., AND MATANI, M. D. An o(1) algorithm for implementing the lfu cache eviction scheme. Tech. rep., 2010. "<http://dhruvbird.com/lfu.pdf>".
- [19] LEE, D., CHOI, J., KIM, J., NOH, S. H., MIN, S. L., CHO, Y., AND KIM, C. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Computers* 50, 12 (2001), 1352–1361.
- [20] LI, C. DLIRS: Improving low inter-reference recency set cache replacement policy with dynamics. In *Proc. of the 11th ACM International Systems and Storage Conference (SYSTOR)* (June 2018).
- [21] LI, C., SHILANE, P., DOUGLIS, F., AND WALLACE, G. Pannier: Design and analysis of a container-based flash cache for compound objects. *ACM Trans. on Storage (ToN)* 13, 3 (Sept. 2017).
- [22] LIBERATORE, M., AND SHENOY, P. Umass trace repository. <http://traces.cs.umass.edu/index.php/Main/About>, 2016.
- [23] MANES, B. Caffeine: A high performance caching library for java 8. <https://github.com/ben-manes/caffeine> (2016).
- [24] MEGIDDO, N., AND MODHA, D. S. Arc: A self-tuning, low overhead replacement cache. In *Proc. of the 2nd USENIX Conf. on File and Storage Technologies (FAST)* (2003), pp. 115–130.
- [25] PARK, S., AND PARK, C. FRD: A filtering based buffer cache algorithm that considers both frequency and reuse distance. In *Proc. of the 33rd IEEE International Conference on Massive Storage Systems and Technology (MSST)* (2017).
- [26] RUSSELL, S. J., AND NORVIG, P. *Artificial Intelligence: A Modern Approach* (2nd ed.). Prentice Hall, 2010.
- [27] SCOTT, C. Latency numbers every programmer should know. https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html.
- [28] SUBRAMANIAN, R., SMARAGDAKIS, Y., AND LOH, G. H. Adaptive caches: Effective shaping of cache behavior to workloads. In *Proc. of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (2006), MICRO, pp. 385–396.
- [29] URDANETA, G., PIERRE, G., AND VAN STEEN, M. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks* 53, 11 (July 2009), 1830–1845.
- [30] VIETRI, G., RODRIGUEZ, L. V., MARTINEZ, W. A., LYONS, S., LIU, J., RANGASWAMI, R., ZHAO, M., AND NARASIMHAN, G. Driving cache replacement with ml-based lecar. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)* (2018).
- [31] WALDSPURGER, C., SAEMUNDSSON, T., AHMAD, I., AND PARK, N. Cache modeling and optimization using miniature simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 487–498.
- [32] YOUNG, N. On-line caching as cache size varies. In *Proc. of the second annual ACM-SIAM symposium on Discrete algorithms (SODA)* (1991), pp. 241–250.