It's Time to Revisit LRU vs. FIFO

Ohad Eytan
IBM Research
ohadey@cs.technion.ac.il

Danny Harnik
IBM Research
dannyh@il.ibm.com

Effi Ofer
IBM Research
effio@il.ibm.com

Roy Friedman
Technion
roy@cs.technion.ac.il

Ronen Kat
IBM Research
ronenkat@il.ibm.com

Abstract

We revisit the question of the effectiveness of the popular LRU cache eviction policy versus the FIFO heuristic which attempts to give an LRU like behavior. Several past works have considered this question and commonly stipulated that while FIFO is much easier to implement, the improved hit ratio of LRU outweighs this. We claim that two main trends call for a reevaluation: new caches such as front-ends to cloud storage have very large scales and this makes managing cache metadata in RAM no longer feasible; and new workloads have emerged that possess different characteristics.

We model the overall cost of running LRU and FIFO in a very large scale cache and evaluate this cost using a number of publicly available traces. Our main evaluation workload is a new set of traces that we collected from a large public cloud object storage service and on this new trace FIFO exhibits better overall cost than LRU. We hope that these observations reignite the evaluation of cache eviction policies under new circumstances and that the new traces, that we intend to make public, serve as a testing ground for such work.

1 Introduction

Caching refers to the practice of putting a relatively fast and small storage as a front-end to a slower and larger storage. The cache has the potential to respond to read requests and provide faster access times - latency and throughput. Historically, caching refers to putting fast non-volatile memory in front of persistent storage such as disk. However, this notion can be generalized to all sorts of combinations, ranging from faster levels of memory serving as a cache to slower ones (e.g. L2 and L1 cache in front of a DRAM memory), to different classes of persistent storage (e.g. faster SSDs serving as a "flash-cache" to slower HDDs) and at times also the same type of storage but at different locations (e.g. in CDNs, local nodes serve as a cache to distant back-end nodes or storage). In this paper we focus on the latter type of caching that has become more relevant with the rise of the cloud. Namely, edge caching of remote cloud content. In particular, our study looks at such a setting for a cloud object storage service. An object storage service stores large, immutable objects in the cloud using a RESTful API. Since it services customers over the web, it typically suffers from poor latency and as such, holding a front-end edge cache can be very beneficial.

LRU vs. FIFO. The cache eviction policy is the central component for deciding the content of the cache which in turn dictates the cache efficiency. As such, evaluating cache eviction policies has received much attention over the years. There are numerous cache replacement policies in the literature, yet to most developers the LRU policy serves as a synonym for caching. LRU evicts the Least Recently Used item from the cache, and as such needs to monitor the most recent access to each item in the cache. A closely related eviction policy is the FIFO algorithm, which like its name (First In First Out) evicts the oldest item in the cache. This practice has the obvious drawback that an item that was reused just recently might still be evicted from the cache if it was inserted a long time ago. On the other hand, FIFO's main merit is its management simplicity: hold all the items in the cache in a queue and evict the next in line, with no need to update information about items already in the queue.

As such, comparing the performance of LRU vs. FIFO has garnered attention from the early days of caching. Naturally, it was observed that FIFO has the potential to perform significantly worse than LRU. FIFO is notorious for having the "Belady anomality" [1] in which adding more space to the cache can at times reduce the hit rate of the cache. Several works went on to claim that from a practical point of view, LRU is better than FIFO, whether based on experiments [2], or by defining theoretical models and analyzing the behavior of LRU vs. FIFO under these models [2–6]. In this paper we intend to revisit this statement and claim that with a change of circumstances, LRU can no longer be assumed to be better than FIFO.

It's Time to Revisit LRU vs. FIFO. We stipulate that two main trends have changed the picture in this debate and call for a reevaluation:

1. A New Scale to Caches - In the early days of caching the main deployment was memory based, and managing the cache metadata was carried out entirely in memory (by metadata we refer to the information required to carry out the eviction policies). Since the metadata is significantly smaller than the actual data being cached, this would typically take up only a small fraction of the cache memory, at the expense of some of the potential caching space.

However, with the rise of the cloud and the data deluge, we are now considering caching in persistent storage (even on spinning disks), that leverage geographical proximity to achieve speed advantage. Such a cache can hold capacities that are orders of magnitude higher than traditional caches, and respectively, the cache metadata associated with such a capacity can no longer fit in memory and need to spill over to the persistent media. In such a scenario, FIFO with its very simple management requirements, has a significant advantage over other caching strategies.

2. New Workloads - Caching evaluations carried out in the past have centered around workloads for memory, files and block storage. But *times, they are a'changing*, and new workloads, such as big data analytics and machine learning, have different characteristics that may significantly skew old results. In our study, we evaluate how to build a front-end cache for a large public cloud object store, a workload that carries a large scale and a new semantic behavior. Previous empirical caching studies may no longer hold for this new workload.

This Work. We reevalute the cache effectiveness of the LRU and FIFO eviction policies on a number of real world traces. The main trace that we use is a new trace collected from the IBM Cloud Object Storage service (COS), a trace that we intend to make public. We collected weekly traces of 99 tenants, accounting for over 850 Million I/O requests and amounting to 158 TBs of data being accessed.

Our evaluations show that on real world traces, FIFO achieves hit rates that are very close to those of LRU, yet require a much smaller overhead for managing the cache. We then model the overall effectiveness of a cache policy taking into account cache management operations and observe that in many cases FIFO outperforms LRU. This is particularly evident in the new cloud based traces where depending on the exact configuration, FIFO outperforms LRU on 80% or more of the weekly traces of the cloud object store. These results vary depending on the performance gap between the cache and the backend storage, where the larger the gap is, the more the cache hit ratio dominates the results. In extreme cases, LRU typically wins out by a slight margin but in many realistic settings, FIFO is typically the preferred policy.

We hope that these findings lead to a new discussion around cache eviction policies, and the new traces serve as a new testing ground for such works.

2 Large Caches and Cost Model

2.1 The Effect of Large Scale Cache Deployment on Cache Management

The premise of a cache is that it is faster than the backend storage. But speed comes at a cost, which is why the backend usually stores all the data and only a subset resides in the cache. The goal of a good caching strategy is therefore to make sure that the "hottest" data items reside in the cache, so that as many read requests as possible will be served from the cache (i.e., *cache hits*) and thus achieve better overall storage performance. The central component for choosing what data should reside in the cache is the cache eviction policy which decides what data to evict from the cache in order to make room for new data that has just been requested.

This paper puts a spotlight on the overhead of actually implementing a cache eviction policy. In our discussion we distinguish between cache *data* and *metadata*. While the data is the actual user data being held in the cache, the metadata refers to the information stored in order to find data in the cache and choose the right item to evict from the cache according to the specific cache policy. Our observations hinge on the fact that in very large cloud caches, as discussed in Section 1, the amount of metadata becomes too large to be held solely in memory. The implications of this differ depending on the cache eviction policy.

Cache Eviction Policies. The most well known cache eviction policies are the LRU, which monitors recency, and the Least Frequently Used (LFU), which monitors frequency, of data in order to make intelligent eviction decisions. Most other algorithms, such as ARC [7], GDSF [8], Hyperbolic [9], FRD [10], LIRS [11] and W-TinyLFU [12, 13], combine recency and frequency information to make their eviction decisions. Each of these eviction heuristics has its weaknesses and advantages and their success is very workload dependent. However, all of these algorithms require updates of the metadata upon a cache hit, which is the crux of our observation. Once metadata is not held entirely in RAM, it is very likely that metadata updates would entail relatively expensive updates to persistent disk.

FIFO is a simple heuristic that attempts to approximate LRU to the best of its ability. It is unique in that it is hardly affected when the cache metadata does not fit in memory. In this work we focus solely on comparing LRU to the FIFO heuristic and leave comparison to other methods to future work (see discussion in Section 5). We note that we have experimented with several different caching algorithms and found that on the diverse workloads that we run, none of the

algorithms consistently outperform LRU in terms of hit rate. Thus LRU serves as a good cache eviction policy as any.

The Impact of Large Caches on LRU vs. FIFO. The FIFO algorithm keeps a queue of objects in the order that they were inserted into the cache. Upon a cache miss, it evicts one or more objects from the head and inserts a new object into the tail of the queue. The list does not change upon a cache hit. These characteristics of the FIFO are very useful when not all of the metadata is in memory. Instead, only the head and tail are kept in memory (the head and tail each consist of more than a single item but rather a variable number of items that is limited by the available memory). The rest of the queue is kept in persistent memory, and once in a while the tail is flushed into the persistent storage or additional items from the head are read from it.

An implementation of LRU treats cache misses in the same way as FIFO. But during a cache hit, it needs to modify the order of the queue and move the hit item into the tail. Assuming that most of the metadata is on persistent storage, and since hits can be on items from anywhere in the queue, then this update requires additional expensive random I/Os to the persistent storage. The difference between the two algorithms is illustrated in Figure 1.[1]

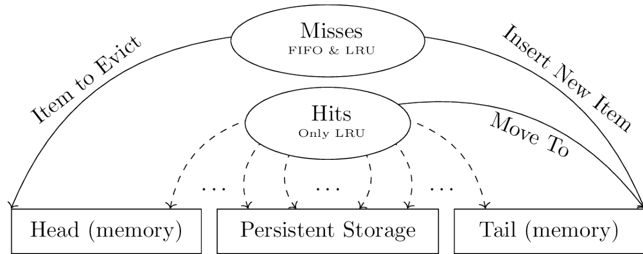


Figure 1: Metadata accesses in FIFO and LRU implementations. The metadata is divided into three parts: the tail where new objects are inserted, the main body, and the head from which objects are evicted. In FIFO, objects inserted into the tail slowly propagated into the head, while in LRU, objects may also move from the body or the head or even the tail into the top of tail of the queue.

Note that there are other considerations such as supporting concurrent updates to the cache, which have been thoroughly studied (e.g. the use of the CLOCK [14] eviction policy is valued for its simple concurrency support). FIFO, like CLOCK has advantages over LRU in this regard as well.

2.2 A Cache Cost Model

Traditional evaluation of cache eviction policies is all about calculating the hit rate of a given trace. But given the observations about the overheads of metadata management, the user experience is no longer dictated solely by hit rate. Evaluating actual user experience (mostly latency) is much trickier than simply understanding the hit rate. The exact latency of cache hits or misses may differ significantly between implementations and systems and even within the same system over time (e.g., as a factor of concurrency and contention on resources with other processes). In order to get a general sense of the user experience that one can expect, we resort to calculating a *cost* model of a caching policy. The cost that we chose is a rough estimation of overall latency in an ideal setting where IOs are performed sequentially and the latency of the cache and the remote backend are fixed. The calculation of the cost is a function of the number of cache misses and cache hits, as well as the latency of local access and remote access. Denote the latency of data on the remote backend by ℓ_{Remote}, and the latency of reading data from the cache by ℓ_{Cache}. The cost of metadata on local persistent storage is denoted by $\ell_{CacheMD}$. Denote the hit-rate of FIFO and LRU by HR_{FIFO} and HR_{LRU}, respectively. Our cost function is formulate as follows:

$$Cost_{LRU} = HR_{LRU} \cdot \overbrace{(\ell_{Cache} + \ell_{CacheMD})}^{data+metadata} + (1 - HR_{LRU}) \cdot \overbrace{\ell_{Remote}}^{data}$$

$$Cost_{FIFO} = HR_{FIFO} \cdot \overbrace{\ell_{Cache}}^{data} + (1 - HR_{FIFO}) \cdot \overbrace{\ell_{Remote}}^{data}$$

For both LRU and FIFO a cache miss entails reading data from the remote backend storage and adding an item to the tail of the queue. The latter is usually in RAM so the cost is negligible. Each hit entails access to the locally cached data. For FIFO there is no additional work, but for LRU, as explained above, an additional update is required to the cache metadata which is, with high probability, on persistent storage. In our evaluations (Section 4) we assume that the cache data and metadata reside on the same media and for simplicity use $\ell_{CacheMD} = \ell_{Cache}$. We also ignore the overhead associated with lookup in the cache as this should be the same for both FIFO and LRU.

2.3 Related Work

Several works have addressed the problem of large caches that cannot hold their metadata in RAM. They present alternatives to approximate LRU and are generally far more complex than FIFO. The TBF design [15] approximates LRU using Two in-memory Bloom Filters combined with iterating over an in memory cache index to find eviction candidates. Me-Clock [16] uses only one bloom filter, but one that supports deletions. It uses a FIFO queue to traverse eviction candidates (as in Clock [14]). Our tests indicate that FIFO is competitive with these methods, yet is far simpler to implement (see further discussion in Section 5).

[1] One can consider an LRU implementation that does locality based caching of the persisted metadata and this has a chance to alleviate some of the overhead. See further discussion in Section 5.

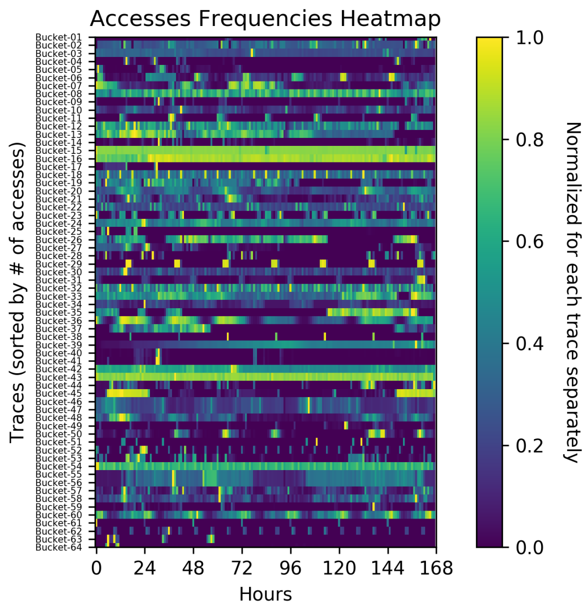


Figure 2: Accesses frequencies of different workloads traced from IBM COS over a one week period

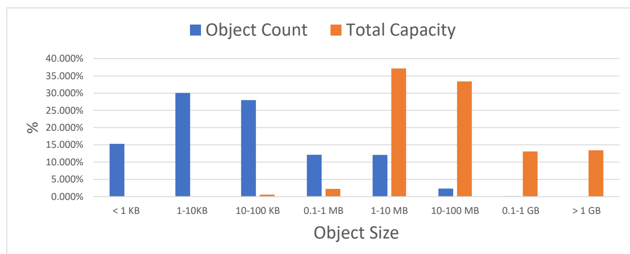


Figure 3: The objects size distribution in the IBM COS trace as well as the capacity of different size categories

3 Traces

To evaluate our work, we used a variety of available traces as well as our newly collected set. We use the Microsoft Cambride traces (MSR) [17, 18], the Fujitsu VM storage traces (SYSTOR) [19, 20] and traces generated from the tpcc benchmark (TPCC) [21]. Finally, we collected traces from the IBM cloud based object store service (IBM COS) [22]. Aggregated information about the different traces can be found in Table 1.

The IBM COS trace IBM Cloud Object Store is a public cloud based object storage service for storing immutable objects, primarily via a RESTful API. We collected 99 traces from this service, each comprised of all the data access requests issued over a single week by a single tenant of the service. The traces include PUT, GET, and DELETE objects requests and include object sizes along with obfuscated object

Table 1: Summary of the trace collections.

Group Name	Traces #	Accesses *Millions*	Objects *Millions*	Objects Size *Gigabytes*
MSR	3	68	24	905
SYSTOR	3	235	154	4,538
TPCC	8	94	76	636
IBM COS	99	858	149	161,869

names. Each trace includes anywhere from 22 thousand to 187 million object requests. We were able to identify some of the workloads as SQL queries, Deep Learning workloads, Natural Language Processing (NLP), Apache Spark data analytic, and document and media servers. But many of the workloads' types remain unknown. The access patterns seen in our traces are very diverse, as depicted in Figure 2 for a selected number of traces. One can see that some workloads access the data continuously, while some access the data periodically, and yet others seem to be relatively ad hoc or random.

The object sizes also show great variance. The overall distribution of object sizes in shown in Figure 3. We see that a vast majority of the objects (85%) in the traces are smaller than a megabyte, Yet these objects only account for 3% of the of the stored capacity. Note that since object storage traces reference objects of variable size one should adopt a strategy for handling such variable sized data within a cache. In our simulation we break large objects into fixed size 4MB blocks and treat each one separately at the cache layer. Requests smaller than 4MB take up their actual length in the cache. This is aligned with the fact that GET requests to the IBM COS often include range reads in which only a part of an object is read.

4 Evaluation Results

Hit Rate Comparison. FIFO attempts to approximate LRU behavior and the simulation results show that it is doing so relatively well. Figure 4 presents several representative Miss Ratio Curves (MRCs) indicating that overall the two methods closely mirror each other. As expected, LRU is often slightly better than FIFO in terms of pure hit rate, yet many times FIFO is similar or even better than LRU. Figure 5 shows that in about half of the traces LRU achieves a higher hit rate while the other half is either identical or FIFO is better.

Cost Comparison. As described in Section 2.2, hit rate is a very central component, but not the only one in assessing the eviction policy. Once we look at the latency cost function rather than hit rate, the picture changes dramatically. Figure 6 presents the cost winners for various latency configurations and in these FIFO comes out as a clear favorite. It should be noted that the higher the difference between the front-end cache latency and the remote storage latency, the more LRU

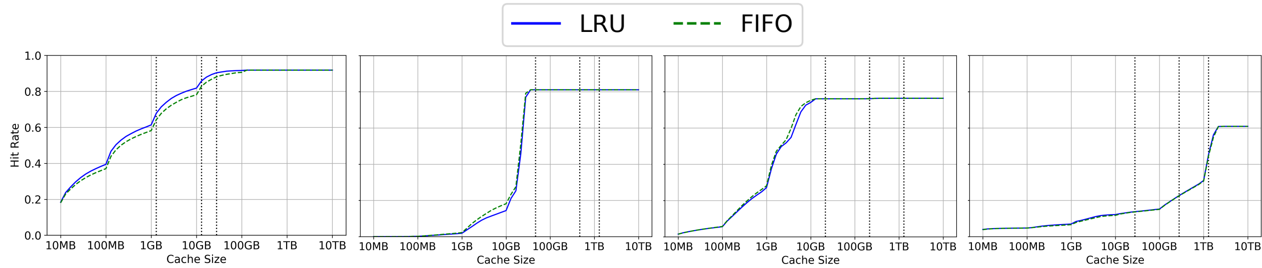


Figure 4: A representative handful of MRCs of the IBM COS traces. The vertical dashed lines, from left to right, indicate cache sizes of 1%, 10% and 30% of the total size of objects in the trace.

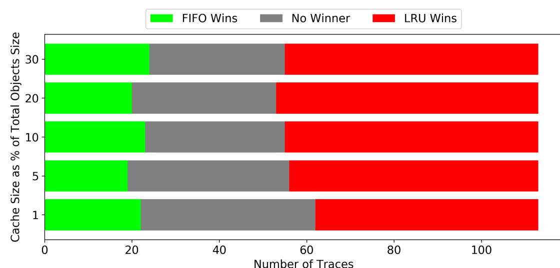


Figure 5: The number of traces in which LRU achieves more cache hits, equal cache hits, or FIFO has more cache hits, evaluated over various cache sizes.

is favored in terms of cost. This is because an extremely high cost for remote access makes hit rate the dominating factor in the equation. This is why LRU is favored in traditional caches where the entire cache resides in RAM. In our tests we used latencies of $\ell_{Cache} = 1ms$ to the front-end and $\ell_{Remote} = 10ms, 50ms, 100ms$ for the remote storage. That being said, these results are more general since the actual latency does not matter in the cost evaluation, rather it is the ratio between the front-end and back-end latencies. Hence the graphs only mention ℓ_{Cache} and ℓ_{Remote} as numbers with no units, with the understanding that these represent ratios.

Table 2: Trace breakdown for a cache that is 30% of the total data size, $\ell_{Cache} = 1$ and $\ell_{Remote} = 50$.

Group	FIFO wins	No Winner	LRU wins
MSR	1	0	2
SYSTOR	1	0	2
TPCC	4	1	3
IBM COS	78	12	9

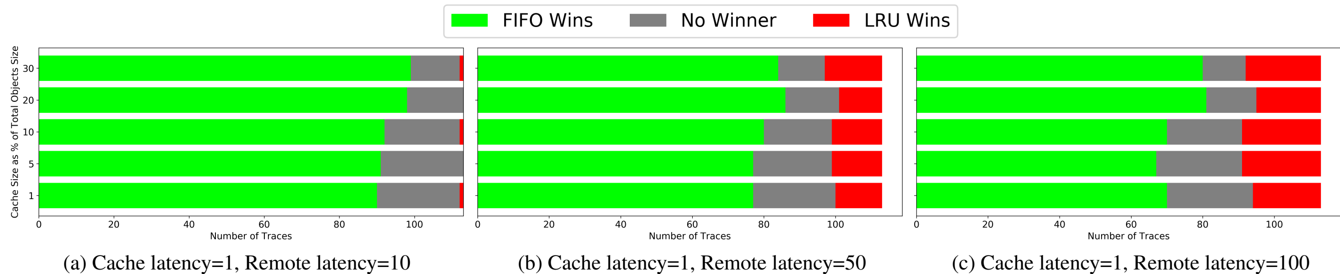
Interestingly, when looking at the breakdown of the traces, we realize that FIFO is not a clear winner in the more tradition workloads, but is clearly superior for the new IBM COS trace. This is exemplified in Table 2 which shows a breakdown according to trace family (for a single setting). Although the sample size is small, for the MSR, SYSTOR and TPCC traces there is no clear winner, but for the IBM COS trace there is a strong bias towards FIFO. We also observed that there is no correlation between the trace size and a FIFO preference (there is great variance in the trace sizes in the IBM COS set). The same ratio of traces preferring FIFO holds for the smallest or largest traces in the set.

However, looking just at the number of winners for either LRU or FIFO does not paint the entire picture, since the actual difference in cost is ignored. Figure 7 gives us a deeper (albeit complex) look at the actual behavior in the traces. For each trace it depicts the difference in latency cost between FIFO and LRU as a function of the actual hit rate. We observe that the difference in hit rate rarely exceeds 5% and for the vast majority of the traces is within 1%. The background color indicates the cost for each trace, and here we see that as the hit rate grows, so does the preference for using FIFO (since LRU behave exactly like FIFO on misses, yet is taxed on hits). Finally, another measure that does take into account the cost difference is the total cost of all traces in the IBM COS set. This measure (shown in Figure 8) clearly favors FIFO over LRU.

5 Discussion

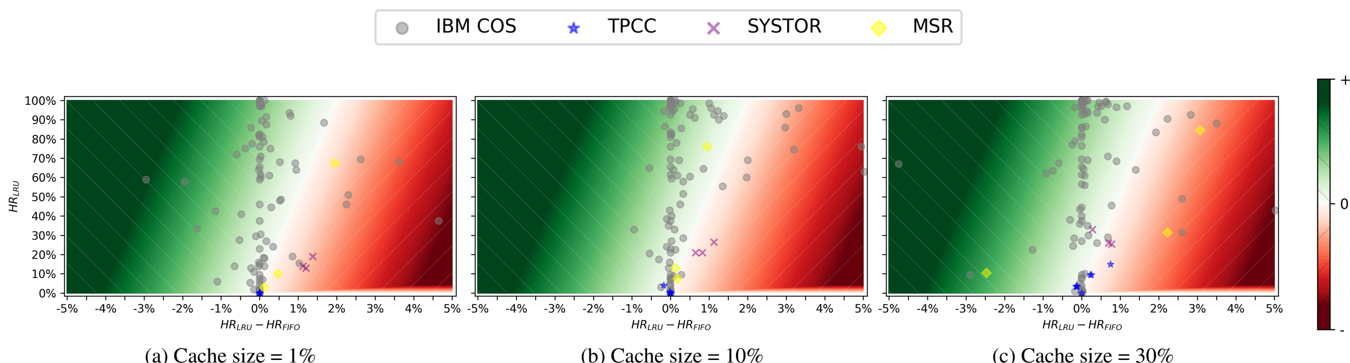
In this paper we advocate to revisit the LRU vs. FIFO question in light of new caching opportunities. We argue that on new traces, and under new media and cache settings, FIFO is actually a better choice than LRU. This is in contrast to previous works that actually have "LRU is better than FIFO" in their title [4, 5]. We hope that our work ignites new studies of cache eviction policies that tackle the cache metadata handling in extremely large edge caches.

In light of our work, one should evaluate methods for approximating LRU with lower memory consumption (e.g. [15, 16]), but keep in mind the very simple FIFO as a base line rather than a full fledged LRU. Similarly, new approaches for memory to hit rate trade-offs should be considered. We note that the CLOCK [14] approximation for LRU seems like a good candidate heuristic to build on. Another approach could be to use a smart paging of LRU metadata into RAM and hope that locality properties in the trace are enough to pro-



(a) Cache latency=1, Remote latency=10 (b) Cache latency=1, Remote latency=50 (c) Cache latency=1, Remote latency=100

Figure 6: The number of traces each policy wins by cost comparison for different cache sizes. "No Winner" is where the cost difference is smaller than 0.01



(a) Cache size = 1% (b) Cache size = 10% (c) Cache size = 30%

Figure 7: Each dot in the plot represents a trace with the y-axis being the LRU hit rate while the x-axis is the hit rate difference between LRU and FIFO. The background color indicates the cost difference between the methods. The more green, the more it favors FIFO and the more red, the more it favors LRU (computed with with $\ell_{Cache} = 1$ and $\ell_{Remote} = 50$).

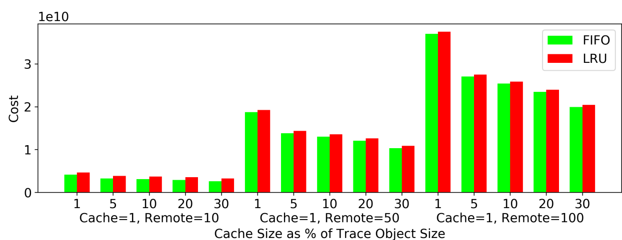


Figure 8: The total cost for all the IBM COS traces with different cache sizes and latency values.

vide sufficient performance. Another interesting direction is to devise memory efficient approximations of more complex eviction policies than LRU. Specifically, policies that take into account frequency as well as recency. Techniques like TinyLFU [12, 13] are a step in this direction.

Finally, the new traces that we collected from the world of cloud object storage should serve as a base for testing caching strategies in this realm. To the best of our knowledge this will be the first publicly available set of traces for such workloads and we hope that it leads to new studies about caching or other areas.

References

[1] L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Commun. ACM*, 12(6):349–353, June 1969.

[2] Asit Dan and Don Towsley. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 143–152, 1990.

[3] Allan Borodin, Prabhakar Raghavan, Sandy Irani, and Baruch Schieber. Competitive paging with locality of reference. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 249–259, 1991.

[4] J Van Den Berg and A Gandolfi. LRU is better than FIFO under the independent reference model. *Journal of applied probability*, 29(1):239–243, 1992.

[5] Marek Chrobak and John Noga. LRU is better than FIFO. *Algorithmica*, 23(2):180–185, 1999.

[6] Joan Boyar, Sushmita Gupta, and Kim S Larsen. Access graphs results for LRU versus FIFO under relative worst order analysis. In *Scandinavian Workshop on Algorithm Theory*, pages 328–339. Springer, 2012.

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

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

[9] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 499–511, 2017.

[10] Sejin Park and Chanik Park. 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.

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

[12] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31, 2017.

[13] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive software cache management. In *Proceedings of the 19th International Middleware Conference*, pages 94–106, 2018.

[14] Fernando J Corbato. A paging experiment with the multics system. Technical report, Massachusetts Inst of Tech Cambridge Project Mac, 1968.

[15] Cristian Ungureanu, Biplob Debnath, Stephen Rago, and Akshat Aranya. TBF: A memory-efficient replacement policy for flash-based caches. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1117–1128. IEEE, 2013.

[16] Zhiguang Chen, Nong Xiao, Yutong Lu, and Fang Liu. Me-CLOCK: A memory-efficient framework to implement replacement policies for large caches. *IEEE Transactions on Computers*, 65(8):2665–2671, 2015.

[17] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):1–23, 2008.

[18] SNIA IOTTA repository, MSR cambridge traces. http://iotta.snia.org/traces/388, 2008.

[19] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *Proceedings of the 10th ACM International Systems and Storage Conference*, pages 1–11, 2017.

[20] SNIA IOTTA repository, Systor '17 traces. http://iotta.snia.org/traces/4928, 2017.

[21] SNIA IOTTA repository, TPCC traces. http://iotta.snia.org/traces/131, 2007.

[22] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen Kat. IBM COS traces used in this work. http://cs.technion.ac.il/~ohadey/lru-vs-fifo/IBMCOSTraces.html, 2020.