On a Caching System with Object Sharing

Nader Alfares
School of EECS
Penn State University
University Park, PA
nna5040@psu.edu

George Kesidis
School of EECS
Penn State University
University Park, PA
gik2@psu.edu

Xi Li
School of EECS
Penn State University
University Park, PA
xzl45@psu.edu

Bhuvan Urgaonkar
School of EECS
Penn State University
University Park, PA
buu1@psu.edu

Mahmut Kandemir
School of EECS
Penn State University
University Park, PA
mtk2@psu.edu

Takis Konstantopoulos
Mathematics
University of Liverpool
Liverpool, UK
t.konstantopoulos@liverpool.ac.uk

ACM Reference Format:
Nader Alfares, George Kesidis, Xi Li, Bhuvan Urgaonkar, Mahmut Kandemir, and Takis Konstantopoulos. 2020. On a Caching System with Object Sharing. In *International Workshop on Middleware and Applications for the Internet of Things (M4IoT '20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3429881.3430107

Abstract — We consider a data/content-caching system located in an edge-cloud that is shared (to reduce costs) by a number of proxies each serving, e.g., a group of data-driven IoT devices. Each proxy operates its own LRU-list of a certain capacity in the shared cache. The length of objects simultaneously appearing in plural LRU-lists is equally divided among them, *i.e.*, object sharing among the LRUs. We give numerical results for our MemCacheD with Object Sharing (MCD-OS) prototype. Also, a working-set approximation for the shared cache is described and how to reduce ripple evictions is discussed.

1 Introduction

The still-developing edge-cloud marketplace is important to the timely support of data-driven applications of resource-limited Internet of Things (IoT) devices. In particular, content-caching services preferably locate at the edge to reduce networking costs and delays, and address the limited storage and energy available to IoT devices, e.g., [8, 11]. One way to reduce the cost of edge-caches is to have different caching proxies (as, *e.g.*, [14]) share objects stored in common cache memory.

We herein consider J proxies that each service a large pool of users/processes making requests for content from a database with N data-objects via a cache of size $B \ll N$ memory units, *e.g.*, caching as part of a Content Distribution Network (CDN). Each proxy typically operates under a Least Recently Used (LRU) caching policy wherein the most recently queried for data-objects are cached. The J proxies *share* both cache memory and possibly also the upload network-bandwidth to their users.[1]

In this paper, we consider a caching system where the noncooperative proxies each pay for an allocation of cache memory (and possibly network I/O as well), thus preventing starvation of any proxy. Objects may be shared among different LRU-lists (or just "LRUs", each corresponding to a proxy), as the shared cache of [13]. That is, the cost of storing a common object in the LRUs is shared among the proxies. Also, an LRU-list miss but physical cache hit is accompanied by a delay corresponding to a physical cache miss. This said, a proxy may make inferences regarding the LRU-lists of others by comparing the cache hits they experience to what they would be without object sharing. Mock queries may change some near-future LRU-list misses to hits (particularly for content not in the physical cache), but will come at the cost of both memory and network I/O resources (possibly *causing* some near-future cache misses that would have been hits). So, the free-riding behavior described in [13] is disincentivized.

This paper is organized as follows. Related prior work is discussed in Sec. 2. In Sec. 3, an approach to cache memory management based on [13] is presented wherein a cached object's length is shared among multiple LRU-lists. In Sec. 4, we described an implementation of MemCacheD with Object Sharing (MCD-OS) and give numerical results. In Sec. 5, we describe an approach to approximating hitting times for such a system of shared cache memory under the Independent Reference Model (IRM) model. In Sec. 6, we describe how a cache with object sharing can be "overbooked" and thus its memory is more efficiently used. Finally, we conclude with a summary and discussion of future work in Sec. 7.

[1] Note that for caching of encrypted data (*e.g.*, owing to copyright protections), a layered encryption strategy (as in block chains or legers) could be used to first encrypt to the network edge and then encrypt to the individual (authorized) users.

2 Related prior work

There is substantial prior work on cache sharing, including at the network edge in support of mobile end-users, *e.g.*, [7, 12, 15]. At one extreme, the queries of the proxies are aggregated and one LRU cache is maintained for all of them using the entire cache memory. At another extreme, the cache memory is statically partitioned among the proxies (without object sharing). For example, [2] describes how cache memory can be partitioned according to a game wherein different proxy utilities increase with cache-hit probability. In our object-sharing problem formulation (involving non-cooperative users of a for-profit caching service), cache memory is not statically partitioned, but there is "virtual" cache memory allocated per user each of which is used for a LRU-list of potentially shared data objects.

For a system with a single LRU (LRU-list) in the cache, a lower priority (paying less) proxy could have a different tail (least recently used object) pointer corresponding to lower amount of allocated memory, but different proxies would then compete for "hot" (higher ranked) objects stored in the cache. To reduce such competition, an interesting system of [4] also has a single LRU maintained in the cache but with highest priority (paying most) proxies having access to the entire cache while lower priority proxies having a *head* (most recently used object) pointer corresponding to a lower amount of allocated memory.

Now consider a scenario where the clients of different proxies may query for (*e.g.*, via a `get` request in Memcached) the same object. In [13], proxies are assigned a share of cached content based on their demand. Individual data objects are *shared* among different proxy caches that store them, each according to the LRU policy, *i.e.*, a share of their length is attributed to each proxy's cache (LRU-list). In [13], the cache blocks some requests selected at random to deter a proxy from "cheating" by issuing mock requests for specific content primarily of interest only to its users in order to keep it cached (hot), while leveraging cached content apportioned to other proxies, *i.e.*, more generally popular content, recall the discussion of Section 1.

3 Object sharing in cache memory

Suppose cache memory is "virtually" allocated so that proxy $i \in \{1, 2, ..., J\}$ *effectively* receives $b_i \leq B$ amount of memory. Each partition is managed simply by a LRU linked-list of pointers ("LRU-list" or just "LRU" in the following) to objects stored in (physical) cache memory collectively for all the proxies.

Let $\mathcal{P}(n) \subset [J]$ be the set of proxies for which object n currently appears in their LRU-list, where $\mathcal{P}(n) = \emptyset$ if and only if object n is not *physically* cached. Note that $\mathcal{P}(n)$ is not disclosed to the proxies, *i.e.*, the proxies cannot with certainty tell whether objects *not* in their LRU-list are in the cache.

Upon request by proxy i for object n of length ℓ_n, object n will be placed at the head of i's LRU-list and all other objects in LRU-list i are demoted in rank.

If the request for object n was a hit on LRU-list i, then nothing further is done.

If it was a miss on LRU-list i, then

- if the object is not stored in the physical cache then it is fetched from the database, stored in the cache and forwarded to proxy i;
- otherwise, the object is produced for proxy i after an equivalent delay.

Furthermore, add i to $\mathcal{P}(n)$ (as in [13]), *i.e.*,

$$\mathcal{P}(n) \quad \leftarrow \quad \mathcal{P}(n) \cup \{i\}, \tag{1}$$

then add the length $\ell_n/|\mathcal{P}(n)|$ to LRU-list i and reduce the "share" of all other caches containing n to $\ell_n/|\mathcal{P}(n)|$ (from $\ell_n/(|\mathcal{P}(n)| - 1)$).

So, if the query for (get request of) object n by proxy i is a miss, its LRU-list length will be inflated and possibly exceed its allocation b_i; thus, LRU-list eviction of its tail (least recently used) object may be required. When an object m is "LRU-list evicted" by any proxy, the apportionment of ℓ_m to other LRU-lists is *increased* (inflated), which may cause other objects to be LRU-list evicted by other proxies. A simple mechanism that the cache operator could use is to evict until no LRU-list exceeds its allocated memory is to iteratively:

1. identify the LRU-list i with largest overflow (length minus allocation)
2. if this largest overflow is not positive then stop
3. evict i's lowest-rank object
4. reassess the lengths of all caches
5. go to 1.

This is guaranteed to terminate after a finite number of iterations because in every iteration, one object is evicted from an LRU-list and there are obviously only ever a finite number of objects per LRU-list.

For example, consider a scenario where object x is in LRU-list j but not i and object y is in both but at the tail of i. Also, both caches are full. So, a query for x by i (LRU miss but cache hit) causes i to evict y. Thus, from j's point-of-view, x deflates but y inflates, so evictions from j may or may not be required.

Also, a `set` request for an object simply *updates* an object in the cache which may cause it to inflate and, in turn, cause evictions, *cf.* Section 4.

Note that if during the eviction iterations, $\mathcal{P}(n) \to \emptyset$ for some object n, then n may be removed from the physical cache (physically evicted) – cached objects n in the physical cache that are not in any LRU-lists are flagged

as such and have lowest priority (are first evicted if there is not sufficient room for any object that is/becomes a member of any LRU-list). Even under LRU-list eviction consensus, the physical cache may store an object *if it has room* to try to avoid having to fetch it again from the database in the future.

In summary, a single proxy i can cause a new object n to enter the cache ($\mathcal{P}(n)$ changes from $\emptyset$ to $\{i\}$) whose entire length ℓ_n is applied to its cache memory allocation b_i, but a consensus is required for an object n to leave the cache ($\mathcal{P}(n) \rightarrow \emptyset$). So, the physical cache itself is not LRU. Also, as objects are requested, their apportionments to proxy LRU-lists may deflate and inflate over time.

Proposition 3.1. *For a fixed set of proxies i, this object-sharing caching system will have a higher stationary object hit-rate per proxy compared to a* not-shared *system of LRU caches, where each proxy i's LRU cache has the same amount of allocated memory b_i in both cases.*

An elementary proof of this proposition is based on a simple coupling argument to show that for each proxy, the objects in the not-shared system's cache are always a subset of what's in the LRU-list of the shared system. This follows simply because the size of any object n apportioned to the shared system

$$\ell_n/|\mathcal{P}(n)| \leq \ell_n,$$

i.e., not greater than its full size which is apportioned in the system without object sharing.

4 MemCacheD with Object Sharing (MCD-OS)

4.1 Background on Memcached

Memcached (MCD) is a popular distributed in-memory cache [10] that offers a `set/get` key-value API (it offers some additional functions such as `update` which is a special case of `set` so we ignore them). Placement/routing of requests to servers within a cluster is done via a consistent hashing function that clients apply to keys. If a `get` request is a hit, the server holding the requested key-value pair responds with the value. If the `get` is a miss, then the client must fetch the item from a (remote) database and issue a `set` command to the cache. The `set` command will add the object if it is not already in the cache, otherwise it will update its value. To guarantee O(1) access time, MCD maintains a hash table on the server side linking all objects in cache, where an object is indexed by the hash value of its key.

The basic unit of storage in MCD is an *item* which stores a key-value pair and some meta-data such as a time-to-live (TTL) value. To overcome internal memory fragmentation, MCD divides memory into multiple *slabs* each of which contains items within a range of sizes. Slabs are 1MB large by default. A group of slabs containing items within the same size range is called a *slabclass*. Instead of using the vanilla LRU, MCD uses type of segmented LRU (S-LRU) that is known to approximate LRU well while posing lower computational needs (and processing delay) when servicing hits (which is the common case in a well-provisioned cache). In MCD's S-LRU, items are separated into three sub-lists called HOT, WARM, and COLD. Newly created items always begin in HOT which is an LRU-based list. An item at the tail of HOT is moved onto WARM only if it has a relatively long TTL and has been accessed at least twice (two or more accesses is taken as indicative of relatively high popularity). WARM holds popular and long-lived items and is operated as a first-in first-out (FIFO) list. Finally, COLD holds relatively unpopular items and is operated as an LRU list.

4.2 Our implementation

We implement an MCD with object sharing, MCD-OS, by making modifications to Memcached v. 1.5.16. We make no changes to the client side of MCD. Our prototype is available here [9]. In particular, we retain MCD's consistent hashing functionality for client-driven content placement/routing in clustered settings as is. We make several changes to the server side of MCD. Requests coming from each proxy are handled by a pool of MCD-OS threads dedicated to that proxy. We retain the slabclass functionality for its fragmentation-related benefits and hash table for quick object access. An item's slabclass continues to be determined by its actual (and not inflated/deflated) size. However, we remove per-slabclass LRU lists and instead implement a single LRU per proxy. Given our specific interest in the LRU replacement policy, we set up MCD-OS in our evaluation such that: (i) flat LRU as opposed to S-LRU is used and (ii) there is only one slabclass. Implementing MCD-OS for S-LRU with multiple slabclasses is part of our ongoing work.

Note that on an LRU miss, MCD-OS will require the proxy to fetch the object from a remote database and issue a `set` command to store it in cache followed by adding the item to the front of this proxy's LRU-list. Therefore, there is no need for MCD-OS to add an artificial delay in response to an LRU miss that is a physical cache hit.

In Table 1, we summarize different types of behavior offered by MCD-OS in response to `set/get` requests from a proxy. We present the key functionalities implemented in MCD-OS to achieve this behavior as a list of functions below. We selectively list new logic added by us and omit related functionality that MCD already implements. In the Appendix, we provide detailed pseudocode for these functions.

proxy i issues `get(k)` **; hits in LRU i**
• promote item with key k to the head of LRU i
proxy i issues `get(k)` **; misses in LRU i but hits in cache**
• insert the item with key k into the head of LRU i • update the status of all other LRUs sharing this item (deflation)
proxy i issues `get(k)` **; misses in both LRU i and cache**
• return cache miss to client // client is expected to fetch the item from database and issue `set(k, v)`
proxy i issues `set(k, v)` **; key k doesn't exist in cache**
• package the key-value pair `(k,v)` into an item, store in cache • set virtual length of the item to its actual length • insert the item to head of LRU i
proxy i issues `set(k, v)` **; key k already exists in cache**
• update the item with key k to reflect the new value `v` • promote the item to head of LRU i • update the status of all other LRUs sharing this item (may involve a combination of inflation and deflation)

Table 1. Summary of MCD-OS behavior in response to `set/get` requests from a proxy.

`inflate`**:** This new function is invoked when a shared item needs to be inflated. This happens upon the eviction of that item from one of the proxy LRUs or if the virtual length of the item increases after a `set` operation.

`deflate`**:** This new function is invoked when a shared item needs to be deflated. This happens upon the insertion into a proxy LRU of an item that is shared with one or more other proxies, or if the virtual length of the item decreases after a `set` operation.

`insert`**:** This is analogous to the native MCD function `item_link` that inserts an item into the appropriate LRU-list. It is used for item insertion and replacement. We modify it to also invoke the functions `inflate` or `deflate` corresponding to an increase or a decrease in the virtual length of the inserted item.

`evict`**:** This is analogous to the native MCD function `item_unlink` that evicts an item from its LRU-list. We modify it to also invoke the function `inflate` after item eviction to update virtual lengths of copies of the evicted item that still resides in some other proxies' LRU-lists.

`process_command`**:** This is a native MCD function that parses client requests and implements `get` and `set` logic. We enhance it to additionally implement object sharing.

4.3 Overhead of object sharing for MCD-OS

Object sharing introduces additional overhead for `set` commands associated with a ripple of evictions among the LRUs owing to item size deflation/inflation. In the following, we compare the overhead of `set` commands (after a cache miss) for MCD-OS and MCD, the latter with the same collective `get` commands but a single LRU cache of the same collective size ($\sum_i b_i$).

For our experiments, we used $J = 9$ proxies with $N = 10^6$ items, where each item was 100kB. The total cache memory was 3 GB. In a typical experiment, we considered very different proxies $i \in [J]$ with Zipf parameter $0.5 + 0.5(i-1)$ and memory allocations: b =100 MB for proxies 1,2,3; b =200 MB for proxies 4,5,6; and b =700 MB for proxies 7,8,9. The number of `get` commands issued in each experiment was 3×10^6 after the cold misses have abated.

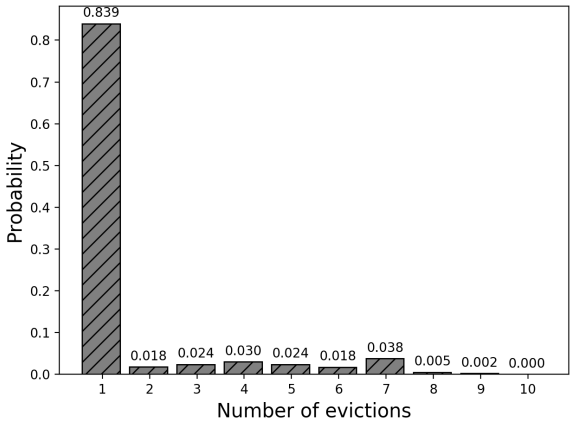

Figure 1. A histogram of the number of evictions per `set` request under MCD-OS. There were no `set` commands observed with more than 10 associated evictions. Note that the number for MCD without object sharing is always 1.

The histogram of the number of evictions per `set` request for MCD-OS is given in Figure 1. As shown, in a small number of cases, the size of this "eviction ripple" can be as large as 9. However, overall only 16% of the `set` requests experienced more than one eviction (i.e., an overhead beyond what an eviction in MCD would experience).

In Figure 2, the Cumulative Distribution Functions (CDF) of the `set` execution times are plotted under both MCD and MCD-OS. Note that, though there is a single eviction per `set` under MCD, there is some variability when updating the LRU. See Table 2.

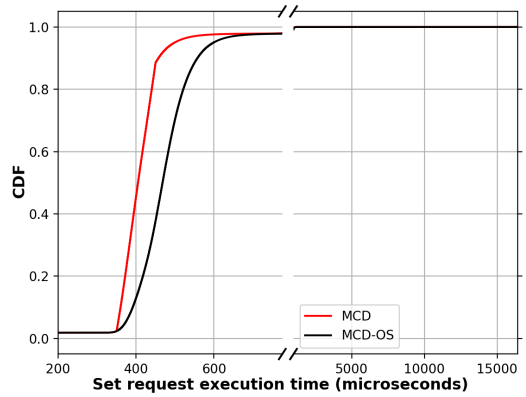

Figure 2. CDFs of the `set` request execution times comparing MCD with MCD-OS.

cache	mean	std dev
MCD	412 μs	111 μs
MCD-OS	474 μs	127 μ s

Table 2. Means and standard deviations of `set` request execution times under MCD-OS and MCD.

Other experiments showed that when all the proxies are very similar, the additional `set` overhead was reduced, even negligible. Also, a `get` under MCD-OS would obviously require additional overhead to look-up which LRU (based on proxy identifier) is requested, but we found it to be negligible.

5 Discussion: Working-set approximation

We now describe an approach to computing the approximate hitting probabilities of the shared caching system following the Denning-Schwartz "working-set approximation" [1, 3] for a not-shared cache under the Independent Reference Model (IRM). [5, 6] nicely address the asymptotic accuracy of this approximation.

Let $\lambda_{i,k}$ be the mean request rate for object k, of length ℓ_k, by proxy i. A simple generalization of the working-set approximation for variable-length objects is: if $\min_i b_i \gg \max_k \ell_k$ then

$$\forall i \; b_i = \sum_{k=1}^{N} h_{i,k}\ell_k \tag{2}$$

where

$$\forall i,k \; h_{i,k} = 1 - \mathrm{e}^{-\lambda_{i,k}t_i} \tag{3}$$

and t_i are interpreted as (assumed common) mean eviction times of objects k in LRU-list i, *i.e.*, the time between when an object enters the cache and when it's evicted from the cache.

For our shared caching system, only a fraction of an object k's length ℓ_k will be attributed to a particular LRU-list i, depending on how k is shared over (eviction) time t_i. For all i,k, let this attribution be $L_{i,k} \le \ell_k$, *i.e.*,

$$\forall i, \; b_i \;\; = \;\; \sum_{k=1}^{N} h_{i,k}L_{i,k} = \sum_{k=1}^{N}(1-\mathrm{e}^{-\lambda_{i,k}t_i})L_{i,k}. \tag{4}$$

One may take

$$L_{i,k}^{(1)} \;\; = \;\; \ell_k \mathbb{E}\frac{1}{1+\sum_{j\neq i} Z_{j,k}}, \tag{5}$$

where $Z_{j,k}$ are *independent* Bernoulli random variables such that $h_{j,k} = \mathbb{P}(Z_{j,k}=1) = 1-\mathbb{P}(Z_{j,k}=0)$. That is, under the assumption of independent LRU-lists, $L_{i,k}^{(1)}$ is the stationary mean attribution of the length of object k to LRU-list i given that k is stored in LRU-list i. For example, for a system with just $J=2$ caches, *i.e.*, $j \in \{1,2\}$,

$$\begin{aligned}\mathbb{E}\frac{1}{1+\sum_{j\neq i} Z_{j,k}} &= 1\cdot(1-h_{3-j,k}) + \frac{1}{2}h_{3-j,k} \\ &= 1 - \frac{1}{2}h_{3-j,k}.\end{aligned}$$

So, substituting (5) into (4) gives, for $i \in \{1,2\}$,

$$0 = b_i - \sum_{k=1}^{N}(1-\mathrm{e}^{-\lambda_{i,k}t_i})(1-\frac{1}{2}(1-\mathrm{e}^{-\lambda_{3-i,k}t_{3-i}}))\ell_k; \tag{6}$$

a system with two nonlinear equations in two unknowns t_1, t_2.

Empirically, we found that using (5) is a good estimate of when $J > 2$ (see [9]), but significantly under-estimates the object hitting probabilities, *i.e.*, $L_{i,k}^{(1)}$ is too large, when $J=2$. In [9], an alternative working-set approximation is given which is accurate for $J=2$. Also, it is shown that there exists a unique solution to (6) for arbitrary $J \ge 2$.

6 Discussion: Overbooking

Consider a caching system as described above with LRU-lists but *without* object sharing, *i.e.*, the full length of an object is charged to each LRU-list in which it resides. In this case, from the proxies' point-of-view, the system is just as static cache partitioning as mentioned in Section 2. Suppose LRU/proxy i is paying to experience the cache-hit probabilities it would get if cache memory amount b_i^* was dedicated to it without object sharing, *i.e.*, b_i^* is prescribed in the Service Level Agreement (SLA). Consider a *virtual* cache memory allocation b_i given by (4) and (5) (accurate for $J > 3$ LRUs [9]). Specifically, let $h_{i,n}$ be the cache hitting probability of object n under

object sharing (so depends on b_i - recall (2)) and $h^*_{i,n}$ be that without object sharing (so depends on b^*_i); and define minimal b_i such that

$$\forall i, n,\ h_{i,n} \geq h^*_{i,n} \quad \Rightarrow \quad \forall i,\ b_i \leq b^*_i. \tag{7}$$

Object sharing with J LRUs operates so that

$$\sum_{i=1}^{J} b_i \quad \leq \quad B, \tag{8}$$

which allows for the possibility of *overbooking*, *i.e.*,

$$\sum_{i=1}^{J} b^*_i \quad > \quad B. \tag{9}$$

For purposes of admission control, before the degree of object-sharing of a new LRU $J+1$ can be assessed, the cache operator can conservatively admit a new proxy $J+1$ if

$$b^*_{J+1} \quad \leq \quad B - \sum_{i=1}^{J} b_i, \tag{10}$$

where the cloud operator estimates the "virtual" cache allocation b_i for existing proxies $i \in \{1, 2, ..., J\}$. Once admitted, the object popularities $\lambda_{i,n}$ can be estimated and fed into our working-set approximation to compute cache-hit probabilities under object-sharing toward determining the proper virtual allocation b_i[2]. Alternatively, the object cache-hit probabilities can be directly estimated by simply trial reducing virtual allocation b_{J+1} starting from b^*_{J+1}. Or, LRU $J+1$ can be less conservatively admitted based on a virtual allocation correponding to some estimated object popularities based on those of existing LRUs $1, ..., J$.

7 Summary and Future Work

In this paper, we considered object sharing by LRU caches as would be deployed in an edge-cloud in support of data-driven IoT devices. Such sharing will reduce the cost of operation at a given level of performance (cache-hit probabilities) or improve performance for given budgets. We developed a memcached prototype (MemDacheD-OS or MCD-OS) and numerically evaluated the `set` overhead of object sharing. We also proposed an extension of the classical working-set approximation of cache-hit probabilities to this shared-object setting (its performance is further detailed in [9]).

We have also implemented MCD-OS for commonly used Segmented-LRU (S-LRU) with multiple slabclasses, where S-LRU is designed to reduce memory overhead for popular (hot) objects. Cache-hit probabilities do not change significantly ($\sim 2-3\%$ difference) for S-LRU with object sharing.

In ongoing work, we are evaluating ways to reduce the overhead of ripple evictions, *e.g.*, by modestly delaying evictions and then process them in batches, but likely at the expense of lower cache-hit rates. We are also evaluating MCD-OS with variable-length objects which are allocated in different slabs.

[2]Note that virtual allocations may also need to be recomputed when LRUs "depart" the cache.

Acknowledgements:

This research was supported in part by NSF CNS grants 1526133 and 1717571 and by a Cisco Systems URP gift.

References

[1] H. Che, Y. Tung, and Z. Wang. Hierarchical Web Caching Systems: Modeling, Design and Experimental Results. *IEEE JSAC*, 20(7), Sept. 2002.

[2] M. Dehghan, W. Chu, P. Nain, and D. Towsley. Sharing LRU Cache Resources among Content Providers: A Utility-Based Approach . *IEEE/ACM Transactions on Networking (TON)*, 27(2), Apr. 2019.

[3] P.J. Denning and S.C. Schwartz. Properties of the working-set model. *Commun. ACM*, 15(3):191–198, March 1972.

[4] A. Eryilmaz and al. A New Flexible Multi-flow LRU Cache ManagementParadigm for Minimizing Misses. In *Proc. ACM SIGMETRICS*, 2019.

[5] R. Fagin. Asymptotic miss ratios over independent references. *Journal Computer and System Sciences*, 14(2):222–250, 1977.

[6] C. Fricker, P. Robert, and J. Roberts. A Versatile and Accurate Approximation for LRU Cache Performance. In *Proc. International Teletraffic Congress*, 2012.

[7] N. Golrezaei, K. Shanmugam, A.G. Dimakis, A.F. Molisch, and G. Caire. Femtocaching: Wireless video content delivery through distributed caching helpers. In *Proc. IEEE INFOCOM*, 2012.

[8] C.-K. Huang, S.-H. Shen, C.-Y. Huang, T.-L. Chin, and C.-A. Shen. S-Cache: Toward an Low Latency Service Caching for Edge Clouds. In *Proc. ACM MobiHoc Workshop on Pervasive Systems in the IoT Era*, July 2019.

[9] G. Kesidis, N. Alfares, X. Li, B. Urgaonkar, M. Kandemir, and T. Konstantopoulos. Working-Set Approximation for a Caching System with Object Sharing. https://github.com/PSU-Cloud/MCD-OS/, Aug. 2019; https://arxiv.org/abs/1905.07641, May 2019.

[10] Memcached. https://memcached.org/.

[11] D. Niyato, D.I. Kim, P. Wang, and L. Song. A novel caching mechanism for Internet of Things (IoT) sensing service with energy harvesting. In *Proc. IEEE International Conference on Communications (ICC)*, May 2016.

[12] K. Poularakis, G. Iosifidis, A. Argyriou, I. Koutsopoulos, and L. Tassiulas. Distributed Caching Algorithms in the Realm of Layered Video Streaming. *IEEE Trans. Mob. Comput.*, 18(4):757–770, 2019.

[13] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica. FairRide: Near-Optimal, Fair Cache Sharing. In *Proc. USENIX NDSI*, Santa Clara, CA, USA, March 2016.

[14] Squid: Optimising Web Delivery. http://www.squid-cache.org.

[15] Y. Wang, X. Zhou, M. Sun, L. Zhang, and X. Wu. A new QoE-driven video cache management scheme with wireless cloud computing in cellular networks. *Mobile Networks and Applications*, 2016.