Working-Set Approximation for a Caching System with Object Sharing

GEORGE KESIDIS, School of EECS
Penn State University
NADER ALFARES, School of EECS
Penn State University
XI LI, School of EECS
Penn State University
BHUVAN URGAONKAR, School of EECS
Penn State University
MAHMUT KANDEMIR, School of EECS
Penn State University
TAKIS KONSTANTOPOULOS, Dept. of Math. Sci.
University of Liverpool, UK

We consider a content-caching system that is shared by a number of proxies. The cache could be located in an edge-cloud datacenter and the proxies could each serve a large population of mobile end-users. The proxies operate their 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. We provide a "working-set" approximation to quickly estimate the cache-hit probabilities under such object sharing. We describe our implementation of MemCacheD with Object Sharing (MCD-OS). We also provide numerical results of experiments using our MCD-OS prototype to evaluate the additional `set` request overhead of object sharing. We identify several interesting directions for future work. In particular, we discuss the outline of an approach to sharing cache network I/O (in addition to sharing memory capacity) based on token bucket mechanisms. We also discuss why and how a proxy may issue "mock" requests to exploit a shared cache and "free ride" on other proxies with whom it shares content. In addition to the introduction of artificial delays in servicing LRU-list misses which are cache hits, we also discuss how pricing may be carried out and used to alleviate such free riding.

Additional Key Words and Phrases: caching, LRU, working-set approximation, object sharing

ACM Reference Format:
George Kesidis, Nader Alfares, Xi Li, Bhuvan Urgaonkar, Mahmut Kandemir, and Takis Konstantopoulos. 2020. Working-Set Approximation for a Caching System with Object Sharing. In *ACM SIGMETRICS '20, June 08–12, 2020, Boston, MA*. ACM, New York, NY, USA, 23 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

As the public cloud-computing marketplace rapidly expands and diversifies its services and associated pricing rules, the edge-cloud marketplace is still developing. Comparable edge-cloud services most likely will be much more expensive than those of the public ("remote") cloud. In particular, content-caching services preferably locate at the edge to reduce networking costs and delays. One way to reduce their costs is to have different caching proxies (as, *e.g.*, [25]) 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 $B \ll N$, *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. Note that for caching of encrypted data (*e.g.*, owing to copyright protections), a layered encryption strategy (as in block chains, legers) could be used to first encrypt to the network edge and then encrypt to the individual (authorized) users.

In this paper, we consider a caching system where each proxy i somehow pays for or is allocated 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 [23]. 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).

This paper is organized as follows. Related prior work is discussed in Section 2. In Section 3, we give some background on (not-shared) caches for variable-length objects. In Section 4, an approach to cache memory management is presented wherein a cached object's length is shared among multiple LRU-lists. In Section 5, we propose an approach to approximating hitting times for such a system of shared cache memory under the Independent Reference Model (IRM) model. Numerical results are given in Section 6. In Section 7, we described an implementation of MemCacheD with Object Sharing (MCD-OS) and give additional numerical results. In Section 8, we describe an approach to sharing cache I/O (specifically network bandwidth from the cache to its users) based on token-bucket mechanisms. In Section 9, we briefly discuss pricing issues and deterring proxies from issuing mock requests. Finally, we conclude with a summary and discussion of future work in Section 10.

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.*, [14, 22, 27]. 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), *cf.* Section 5.3. For example, [9] describes how cache memory can be partitioned according to a game wherein different proxy utilities increase with cache-hit probability. 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 [11] 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 [23], 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 [23], 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, *cf.* Section 9.

3 BACKGROUND ON A (NOT SHARED) CACHE FOR VARIABLE-LENGTH OBJECTS UNDER THE IRM

Assume that the aggregate demand process for object $n \in \{1, 2, \ldots, N\} =: [N]$ is Poisson with intensity λ_n. The Poisson demands are assumed independent. Let the total demand intensity be $\Lambda = \sum_{n=1}^{N} \lambda_n$. So, this is the classical IRM with query probabilities $p_n = \lambda_n / \Lambda$ [1, 5]. First suppose all objects are of unit length and that the cache has capacity B. The set $\mathcal{R}$ of B-permutations of $[N]$ (think of $\mathcal{R}$ as the set of injective functions $r = (r(1), \ldots, r(B))$ from $[B]$ into $[N]$) is the state-space of an LRU Markov chain. The stationary invariant distribution of this Markov chain was found by W.F. King [19]. Let π be this invariant distribution for $B = N$. In fact, π turns out to be an invariant distribution for more general interarrival distributions so long as the object-querying decisions are independent (*i.e.*, this is a kind of insensitivity result).

Now suppose the cache capacity is $B \ll \sum_{n=1}^{N} \ell_n$, where object n has length ℓ_n bytes (or in terms of some other common unit of length), and let $\mathcal{R}$ be the set of N-permutations of $[N]$. In state $r \in \mathcal{R}$, the number of objects in the cache is given by

$$\kappa(r) = \max \left\{ K \in [N] : \sum_{k=1}^{K} \ell_{r(k)} \leq B \right\}. \tag{1}$$

For $K \leq N$, define the K-vector $r^{(K)} = \{r(1), r(2), \ldots, r(K)\}$, *i.e.*, the N-vector r truncated to its first K elements, so a K-permutation of $[N]$. Thus, the stationary probability that the *cache occupancy* equals $r^{(\kappa(r))}$ is

$$\sum_{\rho \in \mathcal{R}:\, \rho^{(\kappa(r))} = r^{(\kappa(r))}} \pi(\rho)$$

By the PASTA (Poisson Arrivals See Time Averages) theorem [2, 28], the hitting probability of object n when the objects are of variable length is

$$h_n = \sum_{r:\, r(n) \leq \kappa(r)} \pi(r). \tag{2}$$

See the byte-hit performance metric of [4]. Because this computation is very complex, a "working-set approximation" (given below) was developed in [10].

4 PARTITIONING CACHE MEMORY

Suppose cache memory is "virtually" allocated so that proxy $i \in \{1, 2, ..., J\}$ receives $b_i \leq B$ and

$$\sum_{i=1}^{J} b_i \leq B. \tag{3}$$

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, *cf.* Section 9.

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 [23]), *i.e.*,

$$\mathcal{P}(n) \quad \leftarrow \quad \mathcal{P}(n) \cup \{i\}, \tag{4}$$

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.

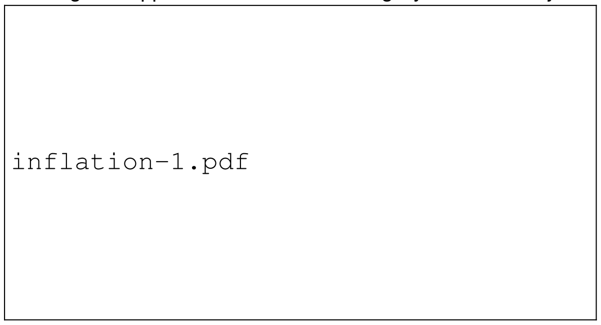
This is guaranteed to terminate after a finite number of iterations because in every iteration, one object is evicted from an LRU-list and there are obviously only ever a finite number of objects per LRU-list.

Figure 1 provides an illustration of such inflation for a shared cache serving 3 LRUs. Here, insertion of a new object causes evictions in all three LRUs.

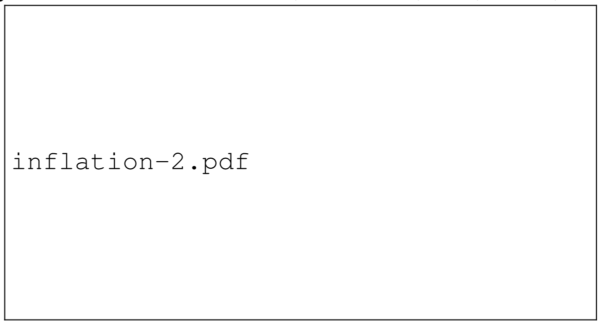
As another example, consider a scenario where object x is in LRU-list j but not i and object y is in both but at the tail of i. Also, both caches are full. So, a query for x by i (LRU miss but cache hit) causes i to evict y. Thus, from j's point-of-view, x deflates but y inflates, so evictions from j 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. Though we do not consider set requests in this paper, our implementation does accommodate them, *cf.* Section 7.

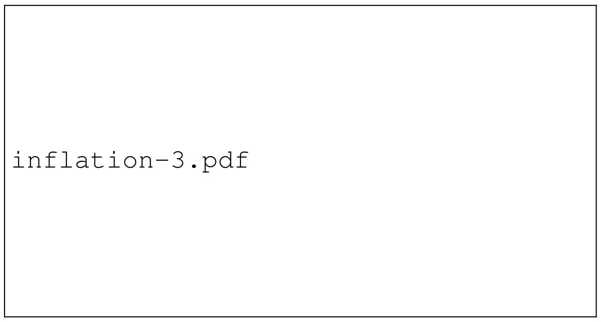
Note that if during the eviction iterations, $\mathcal{P}(n) \rightarrow \emptyset$ for some object n, then n may be removed from the physical cache (physically evicted) – cached objects n in the physical cache that are not in any LRU-lists are flagged as such and have lowest priority (are first evicted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



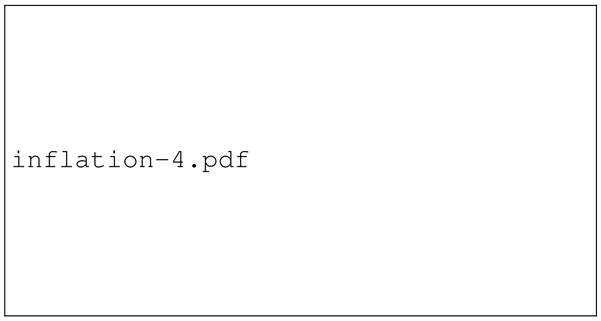
(a) Assume we have three equal sized LRUs. Object 2 is shared by LRU 1, 2, and 3. Object 3 is shared by LRU 2 and 3. A new item, object 1, is inserted to the head of LRU 1.



(b) LRU 1 evicts object 2. So, the virtual length of object 2 inflates in LRUs 2 and 3. So, LRU 2 exceeds its limit and needs to evict object 3.



(c) The increased virtual length of object 3 similarly requires LRU 3 to evict.



(d) LRU 3 evicts object 3. Now no LRU exceeds its limit and processing of the insertion of object 1 into LRU 1 in (a) is completed.

Fig. 1. Inflation of object size caused by deleting a shared object.

Section 9 discusses artificially delaying a response to an LRU-list miss that this a physical cache hit.

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 ($\mathcal{P}(n) \to \emptyset$). 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 4.1. *For a fixed set of active 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.

5 APPROXIMATING LRU-LIST HIT PROBABILITIES UNDER THE IRM

5.1 Working-set approximations for shared-object caches under IRM

In this section, we propose an approach to computing the approximate hitting probabilities of the foregoing caching system following the Denning-Schwartz "working-set approximation" [10] for a not-shared cache under the IRM. Note that [12, 13] address the asymptotic accuracy of this approximation. Also see [8].

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

$$\forall i \; b_i = \sum_{k=1}^{N} h_{i,k}\ell_k \tag{5}$$

where

$$\forall i, k \; h_{i,k} = 1 - \mathrm{e}^{-\lambda_{i,k} t_i} \tag{6}$$

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 $L_{i,k} \leq \ell_k$, *i.e.*,

$$\forall i, \;\; b_i \;\; = \;\; \sum_{k=1}^{N} h_{i,k} L_{i,k} \;\; = \;\; \sum_{k=1}^{N} (1 - \mathrm{e}^{-\lambda_{i,k} t_i}) L_{i,k}. \tag{7}$$

One may take

$$L_{i,k}^{(1)} \;\; = \;\; \ell_k \mathbb{E} \frac{1}{1 + \sum_{j \neq i} Z_{j,k}}, \tag{8}$$

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 (8) into (7) gives, for $i \in \{1, 2\}$,

$$0 = b_i - \sum_{k=1}^{N} (1 - \mathrm{e}^{-\lambda_{i,k} t_i})(1 - \frac{1}{2}(1 - \mathrm{e}^{-\lambda_{3-i,k} t_{3-i}}))\ell_k; \tag{9}$$

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

Empirically, we found that using (8) under-estimates the object hitting probabilities, *i.e.*, $L_{i,k}^{(1)}$ is too large, significantly when $J = 2$. To explain this, we argue that object sharing creates a kind of *positive association* between the LRU-list hit events, because hits in one cause the objects to effectively reduce in size in others, so that they remain in the LRU-lists longer (larger eviction times), thus increasing the hit probabilities in others.

To see why, consider Prop. 5.1 below for Boolean random variables $Y_{j,k}$ indicating the *dependent* events that object k is stored in LRU-list j in steady-state.

LEMMA 5.1. *F_1, F_2 be two cumulative distribution functions (CDFs) on $\mathbb{R}$ such that $F_1(x) \le F_2(x)$ for all $x \in \mathbb{R}$. Let g be a non-increasing function. Then*

$$\int_{-\infty}^{\infty} g(x) dF_1(x) \le \int_{-\infty}^{\infty} g(x) dF_2(x).$$

PROOF. Let $F_i^{-1}(u) := \inf\{x \in \mathbb{R} : F_i(x) > u\}$, $i = 1, 2$. By change of variables in Lebesgue-Stieltjes integrals, we have

$$\int_{-\infty}^{\infty} g(x) \mathrm{d}F_i(x) = \int_{0}^{1} g(F_i^{-1}(u)) \mathrm{d}u.$$

Since $F_1 \le F_2$ we have $F_1^{-1} \ge F_2^{-1}$ and so $g(F_1^{-1}(u)) \le g(F_2^{-1}(u))$ for all $0 < u < 1$. □

PROPOSITION 5.1. *For an arbitrary object index k, consider $J \ge 2$ nonnegative random variables $Y_{1,k}, \dots, Y_{J,k}$ and J* independent *random variables $Z_{1,k}, \dots, Z_{J,k}$ such that, for all i, $Z_{i,k}$ and $Y_{i,k}$ have the same distribution. If for any LRU-list $i \in \{1, 2, \dots, J\}$ we have*

$$\mathbb{P}\left(\sum_{j \ne i} Y_{j,k} \le x\right) \le \mathbb{P}\left(\sum_{j \ne i} Z_{j,k} \le x\right) \tag{10}$$

then

$$\mathbb{E}\left(1 + \sum_{j \ne i} Y_{j,k}\right)^{-1} \quad \le \quad \mathbb{E}\left(1 + \sum_{j \ne i} Z_{j,k}\right)^{-1}. \tag{11}$$

PROOF. Simply take $g(x) = 1/(1 + x)$ in Lemma 5.1 . □

Note that according to (10), $\sum_{j \ne i} Y_j$ tends to be larger than $\sum_{j \ne i} Z_j$, similar to positive associations or positive correlations properties among random variables $Y_i \ge 0$ [17, 18, 26].

By Jensen's inequality,

$$L_{i,k}^{(1)} \quad \ge \quad \ell_k \frac{1}{1 + \sum_{j \ne i} h_{j,k}} =: L_{i,k}^{*} \tag{12}$$

$$\ge \quad \ell_k \frac{h_{i,k}}{h_{i,k} + \sum_{j \ne i} h_{j,k}} =: L_{i,k}^{(2)}. \tag{13}$$

Empirically, we found that using the $L_{i,k}^{*}$ may give approximate hitting probabilities only marginally larger than (8). But, empirically, we found that using (13) tends to *over*-estimate the cache hitting probabilities. Note that $\sum_{i=1}^{J} L_{i,k}^{(2)} = \ell_k$, *i.e.*, $L_{i,k}^{(2)}$ models how object k is shared over time by the different caches i, rather than the mean object length given that it is stored in the cache.

Substituting (8) into (7) gives, for $i \in \{1, 2, \dots, J\}$,

$$0 = b_i - \sum_{k=1}^{N} h_{i,k} \mathbb{E} \frac{1}{1 + \sum_{j \ne i} Z_{j,k}} \ell_k \;=:\; \frac{\partial u_i}{\partial t_i} \;=:\; \partial_i u_i \tag{14}$$

Under (6) and $\mathbb{E} Z_{j,k} = h_{j,k}$ for independent Boolean $Z_{j,k}$, equations (14) are a set of J equations in J unknowns $\{t_i\}_{i=1}^{J}$.

Note that for all the above definitions, $\forall i, k,\ L_{i,k} \le \ell_k$, so one expects corresponding hit cache probabilities to be larger than without object-sharing; recall Prop. 4.1.

5.2 Existence and uniqueness of solution to the working-set approximation (14)

A basic assumption is that,

$$\forall i \; b_i < \frac{1}{J}\sum_{k=1}^{N} \ell_k, \tag{15}$$

i.e., no LRU-list is large enough to hold all of the objects even if the objects were fully shared. Note that if $\forall i \; b_i \geq \frac{1}{J}\sum_{k=1}^{N} \ell_k$, then the total available cache memory ($\geq \sum_{i=1}^{J} b_i$) is sufficiently large to hold all possible data objects (hence is not a "cache").

PROPOSITION 5.2. *If (15) holds then there are real numbers $s_j \geq 0, S_j < \infty$, such that $s_j < S_j$ and there exists a unique solution $\{t_i\}_{i=1}^{J} \in \prod_{i=1}^{J}[s_i, S_i]$ to (14).*

PROOF. Consider the quantities u_i as utilities of a noncooperative J-player game with strategies

$$\{t_j\}_{j=1}^{J} \in \prod_{j=1}^{J}[s_j, S_j] =: \mathcal{S}$$

where $0 \leq s_j < S_j < \infty$. First note that each u_i of (14) is continuously differentiable on $\mathcal{S}$.

For a J-dimensional vector $\underline{t} = (t_1, \ldots, t_J) \in \mathcal{S}$ let $\underline{t}_{-i}$ be the $(J-1)$-vector obtained by eliminating the entry t_i. Since the strategy-space $\mathcal{S}$ is compact and the utility functions $u_i(\underline{t})$ are strictly concave in t_i (since $\partial_i^2 u_i < 0$) a Nash equilibrium exists [3]. Alternatively, we can use Brouwer's theorem [6] to establish existence of the Nash equilibrium.

Generally, a Nash equilibrium may occur on the boundary of the strategy-space. However, note here that for an arbitrary $\underline{t}_{-i}$,

$$\begin{aligned}
&\lim_{t_i \to 0} \partial_i u_i(t_i, \underline{t}_{-i}) = b_i > 0 \text{ and} \\
&\lim_{t_i \to \infty} \partial_i u_i(t_i, \underline{t}_{-i}) = b_i - \sum_{k=1}^{N} \frac{1}{1 + \sum_{j \neq i}(1 - \mathrm{e}^{-\lambda_{j,k} t_j})}\ell_k \\
&\qquad \leq b_i - \frac{1}{J}\sum_{k=1}^{N} \ell_k < 0, \qquad \text{by (15).}
\end{aligned}$$

Because of this and the strict concavity of u_i in t_i, if all S_j are sufficiently large and $s_j \geq 0$ sufficiently small, then all $\partial_i u_i(\underline{t})$ are *unimodal* in t_i for all $\underline{t}_{-i}$ such that $\underline{t} \in \mathcal{S}$. As a result, the Nash equilibria are all interior to $\mathcal{S}$ so that the first-order necessary conditions for u_i-optimality must all hold, *i.e.*, (14) are satisfied.

By such unimodality and because strict concavity implies $\partial_i^2 u_i \neq 0$, uniqueness of the solution follows. □

By the same argument:

COROLLARY 5.1. *Proposition 5.2 is also true under (12) or (13) as well, the latter with $s_i > 0$ for all i.*

Note that the diagonal-dominance conditions implying negative definiteness of the Jacobian of the gradient map, which would imply uniqueness of the solution $\{t_i\}_{i=1}^{J}$ to (14) [21, 24], do not hold here.

5.3 Static cache partitioning with shared objects

Consider a caching system as described above with LRU-lists but *without* object sharing, *i.e.*, the full length of the an object is charged to each LRU-list in which it resides. In this case, from any proxy's point-of-view, the system is just as static cache partitioning mentioned in Section 1. But from the cache's point of view, there may be room for additional objects in the memory even when $\sum_{i=1}^{J} b_i = B$ because every object is only store once in the cache. This additional

i	b_0	b_1	$h_{i,1}$	$h_{i,10}$	$h_{i,100}$	$h_{i,1000}$
0	8	8	0.501	0.113	0.0226	0.00399
0	8	64	0.503	0.113	0.0216	0.00396
0	64	8	0.998	0.687	0.189	0.0343
0	64	64	0.998	0.697	0.195	0.0400
1	8	8	0.203	0.0673	0.0226	0.00709
1	8	64	0.853	0.453	0.172	0.0574
1	64	8	0.208	0.0699	0.0231	0.00736
1	64	64	0.860	0.465	0.179	0.0609

Table 1. Empirical hitting probabilities for a simulated cache under the IRM of size $B = 1000$ for unit-length objects ($\forall n,\ \ell_n = 1$) that is shared by two LRU-lists $i = 1, 2$ respectively with Zipf popularity parameters $\alpha_0 = .75$ and $\alpha_1 = .5$. Simulation time was sufficiently long so that these hitting probabilities are obtained with high confidence.

memory could be used to store objects recently evicted from all LRU-lists (as described above), or the cache operator could attempt to *overbook* the cache, *i.e.*, operate such that $\sum_{i=1}^{J} b_i > B$. That is, the cache operator would benefit from object-sharing and its customers (the proxies) would not.

For example, considering the working-set approximation (5) for independent LRU caches without object sharing under the IRM, if the cache operator can estimate

$$\sum_{k=1}^{N} \ell_k (1 - \prod_{i=1}^{J} (1 - h_{i,k}))$$

for a current set of J proxies, then she can admit a new proxy requiring cache memory b_{J+1} if

$$b_{J+1} \ \le\ B - \sum_{k=1}^{N} \ell_k (1 - \prod_{i=1}^{J} (1 - h_{i,k})).$$

6 NUMERICAL RESULTS ON CACHE MEMORY SHARING

We ran a number of experiments to test the foregoing approximations of hitting probabilities of the shared-object cache. To approximate, we solved (14) using the Newton-Raphson algorithm; this was simplified by the concavity properties and uniqueness of solution discussed in the proof of Prop. 5.2.

The result of a typical experiment for a cache shared by $J = 2$ LRU-lists is given in Table 1 (where hitting probabilities were evaluated by simulation with high confidence) and Table 2 (where (6) and (14) was used). The experiment involved LRU-lists $i = 0, 1$ of size $b_i \in \{8, 64\}$ and $N = 1000$ objects all of unit length. As a reference, we provide Table 4 giving hitting probabilities of isolated caches without data-object sharing.

The "popularity" of a data-object n refers to the mean rate λ_n. at which it is requested. A commonly used model for popularity is given by the so-called Zipf law:

$$\lambda_n \propto \rho(n)^{-\alpha}, \qquad (16)$$

where $\alpha > 0$ is the Zipf parameter and $\rho(n)$ is the popularity rank of object n, *i.e.*, $\rho(n') = 1$ if $n' = \text{argmax}_n \lambda_n$ is unique and $\rho(n'') = N$ if $n'' = \text{argmin}_n \lambda_n$ is unique. For example, values $0.64 \le \alpha \le 0.83$ were given for different datasets in Table 1 of [7].

In a typical two LRU-list experiment, we took the Zipf parameter $\alpha_0 = 0.75$ for LRU-list $i = 0$ and $\alpha_1 = 0.5$ for LRU-list $i = 1$.

i	b_0	b_1	$h_{i,1}$	$h_{i,10}$	$h_{i,100}$	$h_{i,1000}$
0	8	8	.571	.140	.0264	.00474
0	8	64	.805	.253	.0504	.00916
0	64	8	.996	.630	.162	.0309
0	64	64	.999	.791	.243	.0483
1	8	8	.223	.0767	.0249	.00795
1	8	64	.773	.374	.138	.0458
1	64	8	.393	.146	.0487	.0157
1	64	64	.900	.517	.205	.0701

Table 2. Hitting probabilities numerically approximated using Newton-Raphson to solve (7) for LRUs i with mean object lengths (13) for $\{t_i\}_{i=1}^J$, and then using (6), for the shared cache of Table 1.

i	b_0	b_1	$h_{i,1}$	$h_{i,10}$	$h_{i,100}$	$h_{i,1000}$
0	8	8	0.357	0.0756	0.0139	0.00248
0	8	64	0.392	0.08479	0.0156	0.00280
0	64	8	0.983	0.516	0.121	0.0227
0	64	64	0.990	0.557	0.135	0.0254
1	8	8	0.125	0.041	0.0133	0.00421
1	8	64	0.676	0.300	0.107	0.0350
1	64	8	0.136	0.0453	0.0146	0.0046
1	64	64	0.712	0.325	0.117	0.03857

Table 3. Hitting probabilities numerically approximated instead using mean object lengths (8) for the shared cache of Table 1, *i.e.*, solving (14). Using mean object lengths (12) gives similar results.

cache i	b_0	b_1	$h_{i,1}$	$h_{i,10}$	$h_{i,100}$	$h_{i,1000}$
0	8	n/a	.354	.0735	.0133	.00222
0	64	n/a	.981	.504	.123	.0200
1	n/a	8	.123	.0403	.0137	.00376
1	n/a	64	.665	.295	.105	.0343

Table 4. Empirical hitting probabilities of isolated (not object-sharing) caches under the IRM of size b_i, $i = 1, 2$, for unit-length objects ($\forall n$, $\ell_n = 1$) respectively with Zipf popularity parameters $\alpha_0 = .75$ and $\alpha_1 = .5$.

First note that the cache-hit probabilities of Table 1 are generally higher than those (LRU-hit probabilities) of Table 4, *i.e.*, object sharing obviously increases cache-hit probabilities, recall Prop. 4.1. The cache-hit probabilities by working-set approximation of Table 2 (corresponding to mean object lengths (13)) are larger than those of Table 1. The cache-hit probabilities of Table 3 (corresponding to mean object lengths (8)) are smaller (by ~30%) than those of Table 1. These working-set approximations required orders of magnitude less computation than system simulation with high confidence of Table 1. The approximations of Table 2 are 10-20% higher in most cases, except when the cache-memory sizes are quite different – *e.g.*, "0,8,64" (LRU-list $i = 0$ with $b_0 = 8$ and $b_1 = 64$) and "1,64,8" – the approximation is about 80% higher.

Typical results for a cache shared by three or more LRU-lists are shown in Tables 5 and 6 (for $J = 3$ LRU-lists). Here we see that the approximation (8) is reasonably accurate.

Finally, we note from Table 7 the lower cache-hit probabilities for a typical instance of the set of parameters of the caching system of Table 5, again consistent with the statement of Prop. 4.1. The differences range from marginal to over 10% in the case of the LRU 2 with the smaller memory allocation ($b_2 = 8$).

i	b_0	b_1	b_2	$h_{i,1}$	$h_{i,10}$	$h_{i,100}$	$h_{i,1000}$
0	8	8	8	0.368	0.0758	0.0142	0.00226
0	8	8	64	0.407	0.0877	0.0158	0.00273
0	8	64	8	0.389	0.0823	0.0149	0.00271
0	8	64	64	0.422	0.0924	0.0167	0.0028
0	64	8	8	0.983	0.5138	0.1170	0.02303
0	64	8	64	0.989	0.5568	0.1325	0.02660
0	64	64	8	0.986	0.5387	0.1262	0.02366
0	64	64	64	0.992	0.5763	0.1445	0.02724
1	8	8	8	0.126	0.0412	0.0130	0.00423
1	8	8	64	0.136	0.0448	0.0138	0.00438
1	8	64	8	0.676	0.2991	0.1069	0.03422
1	8	64	64	0.699	0.3205	0.1131	0.03574
1	64	8	8	0.136	0.0438	0.0136	0.00425
1	64	8	64	0.143	0.0476	0.0146	0.00458
1	64	64	8	0.699	0.3159	0.1129	0.03639
1	64	64	64	0.726	0.3318	0.1205	0.03916
2	8	8	8	0.708	0.1142	0.0121	0.00116
2	8	8	64	1.000	0.7560	0.1292	0.01411
2	8	64	8	0.745	0.1281	0.0130	0.00146
2	8	64	64	1.000	0.7882	0.1419	0.01628
2	64	8	8	0.771	0.1383	0.0146	0.00168
2	64	8	64	1.000	0.7968	0.1419	0.01435
2	64	64	8	0.793	0.1502	0.0147	0.00153
2	64	64	64	1.000	0.8196	0.1597	0.01416

Table 5. Empirical hitting probabilities for a simulated cache under the IRM of size $B = 1000$ for unit-length objects ($\forall n,\ \ell_n = 1$) that is shared by three LRU-lists $i = 0, 1, 2$ respectively with Zipf popularity parameters $\alpha_0 = .75$, $\alpha_1 = .5$, and $\alpha_2 = 1$. Simulation time was sufficiently long so that these hitting probabilities are obtained with high confidence.

7 MEMCACHED WITH OBJECT SHARING (MCD-OS)

7.1 Background on Memcached

Memcached (MCD) is a popular distributed in-memory cache [20] 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 have been accessed

i	b_0	b_1	b_2	$h_{i,1}$	$h_{i,10}$	$h_{i,100}$	$h_{i,1000}$
0	8	8	8	0.365	0.0776	0.0143	0.00255
0	8	8	64	0.401	0.0872	0.0161	0.00288
0	8	64	8	0.386	0.0832	0.0153	0.00274
0	8	64	64	0.421	0.0926	0.0171	0.00307
0	64	8	8	0.984	0.5213	0.1228	0.02302
0	64	8	64	0.990	0.5622	0.1366	0.02579
0	64	64	8	0.988	0.5455	0.1308	0.02463
0	64	64	64	0.993	0.5846	0.1446	0.02740
1	8	8	8	0.126	0.0416	0.0133	0.00424
1	8	8	64	0.134	0.0446	0.0143	0.00455
1	8	64	8	0.678	0.3011	0.1071	0.03519
1	8	64	64	0.704	0.3197	0.1147	0.03779
1	64	8	8	0.133	0.0442	0.0142	0.00451
1	64	8	64	0.142	0.0472	0.0152	0.00482
1	64	64	8	0.701	0.3171	0.1136	0.03742
1	64	64	64	0.725	0.3353	0.1212	0.04002
2	8	8	8	0.694	0.1116	0.0118	0.00118
2	8	8	64	1.000	0.7556	0.1314	0.01399
2	8	64	8	0.734	0.1242	0.0132	0.00133
2	8	64	64	1.000	0.7861	0.1429	0.01530
2	64	8	8	0.756	0.1314	0.0140	0.00141
2	64	8	64	1.000	0.7995	0.1484	0.01594
2	64	64	8	0.787	0.1434	0.0154	0.00155
2	64	64	64	1.000	0.8249	0.1599	0.01727

Table 6. Hitting probabilities numerically approximated instead using mean object lengths (8), solving (14) and substituting into (6), for the shared cache of Table 5.

i	b_0	b_1	b_2	$h_{i,1}$	$h_{i,10}$	$h_{i,100}$	$h_{i,1000}$
0	64	64	8	0.9800	0.5084	0.11760	0.02259
1	64	64	8	0.6683	0.2944	0.10437	0.03503
2	64	64	8	0.7005	0.1123	0.01176	0.00113

Table 7. Hitting probabilities of LRUs i when caches are not shared for parameters of system of Table 5.

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.

7.2 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. 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

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

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 (*cf.* Section 9).

In Table 8, 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.

- `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.

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

Table 9. Means and standard deviations of `set` request execution times under MCD-OS and MCD.

- `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.

Our MCD-OS implementation will be open-sourced.

7.3 Overhead of object sharing for MCD-OS

GK: mention that overhead is only for LRU misses (note that S-LRU economizes on cache hits). Could batch process multiple LRU misses in the hopes that some involve the same object missed from plural different LRUs (so inserting smaller object and hence less ripple). But, delaying processing of LRU misses could mean plural misses of the *same* LRU for the same object...

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. The histogram of the number of evictions per `set` request for MCD-OS is given in Figure 2. 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 3, the CDFs 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. The corresponding means and standard deviations are given in Table 9.

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.

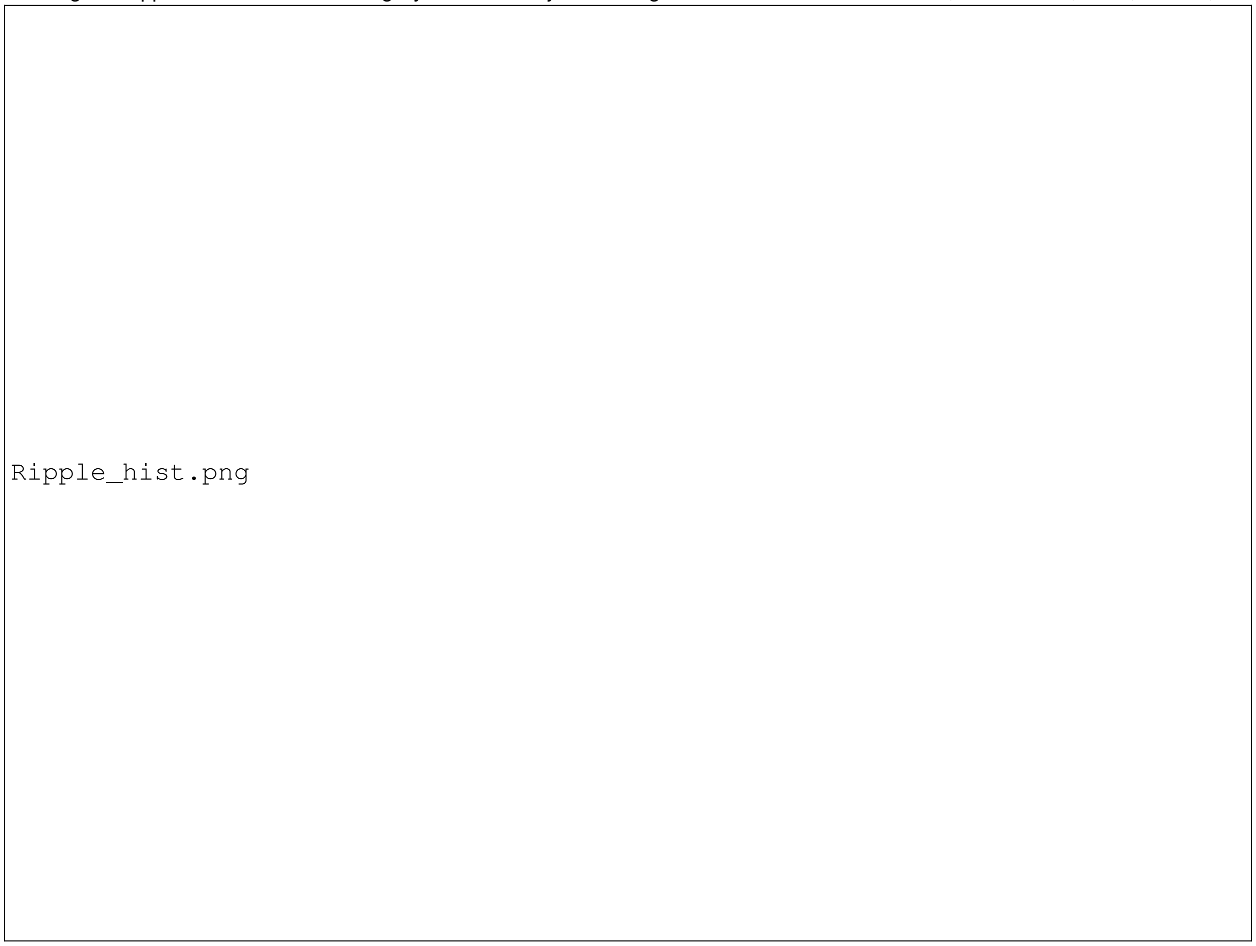


Fig. 2. A histogram of the number of evictions per `set` request under MCD-OS. There were no `set` commands observed with more than 10 associated evictions. Note that the number for MCD without object sharing is always 1.

8 DISCUSSION OF CACHE NETWORK I/O SHARING

Let $R_{i,n}$ be the incident mean-rate that proxy i requests object n having length ℓ_n. Let $\lambda_{i,n}$ be the g_i*-admitted* mean request rate, where over any interval of length t, the total length of requests admitted is at most $g_i(t)$. Thus,

$$\forall i, \ \sum_n \ell_n \lambda_{i,n} \quad \leq \quad \lim_{t\to\infty} \frac{g_i(t)}{t}. \tag{17}$$

Let the total mean-rate of requests for object n to the cache be

$$\lambda_n = \sum_{i=1}^{J} \lambda_{i,n}. \tag{18}$$

A proxy does not wish to unnecessarily pay more than it needs for larger demand envelope g nor to have its users' requests blocked (or delayed in order to conform to its demand envelope g). So, assume that g_i is selected (purchased) by proxy i so that it minimally impedes the estimated incident rate of its users' requests. For example, if the aggregate demand from proxy i is Poisson (so that the mean number of requests in a time interval approximately equals the variance), then

$$g_i(t) = g_i(0) + tR_i + 3\sqrt{tR_i}, \tag{19}$$

set_cdf.png

Fig. 3. CDFs of the set request execution times comparing MCD with MCD-OS.

may be selected, where $R_i = \sum_n R_{i,n}$ is the mean aggregate request rate. For large R_i recall that the (Poisson distributed) number of requests in an interval of time t approximately $\mathsf{Normal}(tR_i, tR_i)$. Hence, in selecting g_i in (19), we limit the probability of a blocked request to about 1%. Here, $g_i(0) > 0$ corresponds to a number of allowed simultaneous requests. In practice, such a demand envelope g_i may be approximated by piecewise linear one implemented by one or more token-bucket mechanisms, *e.g.*, [15, 16]. That is, a token corresponds to a common unit of object/content length (the dimension of ℓ_n).

For long-term stability, the cache network I/O bandwidth C should satisfy

$$C \geq \lim_{t\to\infty} \sum_i \frac{g_i(t)}{t} \quad \Rightarrow \quad C \geq \sum_n \ell_n \lambda_n. \tag{20}$$

Admission control of new proxies and dynamic pricing could be based on in part on (20).

It is possible that, periodically, demand may exceed cache network bandwidth, so the cache will have a *finite* request queue. The total length of in-profile [15, 16] *objects* corresponding to queued requests will be upper bounded by

$$\max_{t>0} \sum_i g_i(t) - Ct.$$

Instead of blocking *out-of-profile* requests, one could set-up queues for both in-profile and out-of-profile requests, where the former would obviously have priority over the latter.

Finally, recall the edge-computing context in support of mobile users. If the same object is requested at the same time over a wireless channel, the necessary tokens may be shared among requesting proxies. Again, a problem here is intellectual property protections: content could be encrypted to the cache and then commonly encrypted to a *group* of authorized mobile subscribers. The cryptographic keys required for the latter may be periodically refreshed to deal with subscriber churn.

9 DISCUSSION OF PRICING AND MOCK REQUESTS

Generally, prices charged to proxy i could depend in part on resource reservations (g_i, b_i). Service may be organized in tiers, where tier k is associated with demand envelope $g = k\gamma$ and cache memory allocation $b = k\beta$, and $k = 1$ corresponds to an atomic or base service tier. Though the price π_k of tier k is obviously increasing with (integer) k, it may be such that π_k/k is decreasing as a kind of volume discount. On the other hand, when resources are scarce and competition for them is high, π_k/k may be increasing. Also, costs may be affected by usage – there is precedent for *penalizing* low utilization, presumably to encourage customers to better characterize their workloads and not be resource wasteful.

A query by proxy i that is an LRU-miss but a physical cache hit can be subjected to an added delay so that it appears as a cache miss from proxy i's point-of-view[1]. So, from proxy i's point of view, there is no incentive to "free ride" on other proxies by issuing mock queries[2] so that i's LRU-list is populated primarily by objects that are popular only among its own users. Mock requests will deplete apportioned network bandwidth resources too. Also, if allocated resources are multiples of an "atomic" service tier, a proxy may not be able to purchase just a little more network bandwidth to mitigate this impact of mock requests.

Prices under object sharing should obviously be upper bounded by those for the same resources but without object sharing, *i.e.*, resources dedicated to the proxy. A discount from such an upper bound may be given based on the improved cache hit probability under object sharing (reflecting how the queries by the proxy under consideration align with those of other proxies). Given empirical estimates of object popularities (relative request rates) for a given proxy, cache/LRU-hit probabilities can be estimated using working-set approximations. Alternatively, pricing can be based on the larger amount of memory allocation without object sharing that would be required to achieve the mean cache-hit probability (across all data objects) that is enjoyed under object sharing.

10 SUMMARY AND FUTURE WORK

In this paper, we considered object sharing by LRU caches. Such sharing will reduce the cost of operation at a given level of performance (cache-hit probabilities) or improve performance for given budgets. Mock requests are disincentivized by adding artificial delays to queries (get requests) that are LRU misses but cache hits, *i.e.*, delays that would correspond to cache misses. We proposed an extension of the classical working-set approximation of cache-hit probabilities to this shared-object setting, and evaluated its performance both numerically and based on experiments with a Memcached prototype (MemDacheD-OS or MCD-OS). In particular, we numerically evaluated the `set` overhead of object sharing. In ongoing work, we are implementing MCD-OS for S-LRU with multiple slabclasses. We expect that cache-hit probabilities will not change significantly under S-LRU. Future work will also consider in depth the pricing and network-bandwidth sharing issues described above.

[1]Again, this is not necessary in client-driven MCD-OS as an LRU miss will require the client to obtain the object from the remote database and issue a `set` command.

[2]Recall mention of mock requests in Section 2 and how blocking some requests selected at random was suggested as a deterrant in [23].

ACKNOWLEDGMENTS

This research supported in part by a Cisco Systems URP gift, NSF CNS grant 1526133 and NSF CSR grant 1717571.

REFERENCES

[1] O.I. Aven, E.G. Coffman, and Y.A. Kogan. 1987. *Stochastic analysis of computer storage*. D. Reidel Publishing Co.

[2] F. Baccelli and P. Bremaud. 1991. *Elements of Queueing Theory*. Springer-Verlag, Application of Mathematics: Stochastic Modelling and Applied Probability, No. 26, New York, NY.

[3] T. Başar and G.J. Olsder. 1999. *Dynamic Noncooperative Game Theory*. Classics in Applied Mathematics, SIAM, Philadelphia.

[4] A. Balamash and M. Krunz. 2004. An Overview of Web Caching Replacement Algorithms. *IEEE Communications Surveys & Tutorials* 6, 2 (2004).

[5] J. Van Den Berg and A. Gandolfi. 1992. LRU is better than FIFO under the independent reference model. *J. Appl. Prob.* 29 (1992).

[6] K.C. Border. 1985. *Fixed Point Theorems with Applications to Economics and Game Theory*. Cambridge University Press, London.

[7] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. 1999. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proc. IEEE INFOCOM*.

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

[9] M. Dehghan, W. Chu, P. Nain, and D. Towsley. Feb. 2017. Sharing LRU Cache Resources among ContentProviders: A Utility-Based Approach . https://arxiv.org/abs/1702.01823.

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

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

[12] R. Fagin. 1977. Asymptotic approximation of the move-to-front search cost distribution and least-recently-used caching fault probabilities. , 222-250 pages.

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

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

[15] J. Heinanen, T. Finland, and R. Guerin. 1999. A single rate three color marker. *RFC 2697 available at www.ietf.org* (1999).

[16] J. Heinanen, T. Finland, and R. Guerin. 1999. A two rate three color marker. *RFC 2698 available at www.ietf.org* (1999).

[17] K. Joag-Dev and F. Proschan. 1983. Negative association of random variables with applications. *The Annals of Statistics* (1983), 286–295.

[18] A. Khursheed and K.M.L. Saxena. 1981. Positive dependence in multivariate distributions. *Communications in Statistics - Theory and Methods* 10, 12 (1981), 1183–1196.

[19] W.F. King. Aug. 1971. Analysis of paging algorithms. In *Proc. IFIP Congress*. Lyublyana, Yugoslavia.

[20] memcached [n.d.]. Memcached. https://memcached.org/.

[21] H. Moulin. 1984. Dominance Solvability and Cournot Stability. *Mathematical Social Sciences* 7 (1984), 83–102.

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

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

[24] J.B. Rosen. 1965. Existence and uniqueness of equilibrium points for concave N-person games. *Econometrica* 33, 3 (1965), 520–534.

[25] squid [n.d.]. Squid: Optimising Web Delivery. http://www.squid-cache.org.

[26] D. Wajc. Apr. 2017. Negative Association: Definition, Properties and Applications. http://www.cs.cmu.edu/~dwajc/notes/Negative%20Association.pdf.

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

[28] R.W. Wolff. 1989. *Stochastic Modeling and the Theory of Queues*. Prentice-Hall, Englewood Cliffs, NJ.

APPENDIX: MCD-OS IMPLEMENTATION DETAILS

We provide detailed pseudo-code for the key functions that we introduced into MCD for implementing the functionality described in Section 7.

Definitions of variables:

- `alloc_size`: total size of a proxy-LRU
- `avail_size`: available size of a proxy-LRU
- `attrib_size`: attributed size of a proxy-LRU
- `heads[]`: pointers to heads of proxy-LRUs
- `tails[]`: pointers to tails of proxy-LRUs
- `virtual_len`: virtual size of an item
- `actual_len`: actual size of an item
- $\mathcal{P}$: a linked list containing all proxies that have a certain item

```
Function inflate(item *it, libevent_thread* P):
    $N = |\mathcal{P}|$ // $\mathcal{P}$ contains all proxies sharing item it
    diff = $\frac{it \rightarrow actual\_len}{N}$ - it → virtual_len
    it → virtual_len = $\frac{it \rightarrow actual\_len}{N}$
    for $i$ in $\mathcal{P}$ do
        i → attrib_size += diff
        i → avail_size -= diff
    end
    for $i$ in $\mathcal{P}$ do
        // check whether proxy i has enough space after inflation, may evict more than one item
           in proxy-LRU i
        while $i \rightarrow attrib\_size > i \rightarrow alloc\_size$ do
            evict(tails[i], i) // evict the tail item of proxy-LRU i if it's full
        end
    end
    return
```

```
Function deflate(item *it, libevent_thread* k, libevent_thread* P):
    $N = |\mathcal{P}|$ // $\mathcal{P}$ contains all proxies sharing item it
    diff = it → virtual_len - $\frac{it \rightarrow actual\_len}{N}$
    it → virtual_len = $\frac{it \rightarrow actual\_len}{N}$
    for $i$ in $\mathcal{P}$ do
        i → attrib_size -= diff
        i → avail_size += diff
    end
    // check whether proxy k has enough space for the new item, may evict more than one object
       in proxy-LRU k
    while $k \rightarrow attrib\_size > k \rightarrow alloc\_size$ do
        evict(tails[k], k) // evict the tail item of proxy-LRU k if it's full
    end
    return
```

```
Function evict(item *it, libevent_thread *k):
    unlink item it from proxy-LRU k
    k → attrib_size -= it → virtual_len
    k → avail_size += it → virtual_len
    $\mathcal{P}$ = {proxies sharing item it}
    if $\mathcal{P} == \emptyset$ then
        remove item it from hashtable
        free item it in physical cache
    else
        inflate (it, $\mathcal{P}$)
    end
    return
```

```
Function insert(item *it, libevent_thread *k, bool new):
    link item it to head of proxy-LRU k
    add item in hashtable
    /* update statuses of proxies sharing item it */
    $\mathcal{P}$ = {proxies sharing item it}
    N = $|\mathcal{P}|$ // at least 1 since item it is inserted to k
    old_virtual_len = it→virtual_len
    new_virtual_len = $\frac{it\rightarrow actual\_len}{N}$
    diff = old_virtual_len - new_virtual_len
    if new then
        // item it is new to proxy-LRU k
        k→attrib_size += old_virtual_len
        k→avail_size -= old_virtual_len
    end
    switch diff do
        case diff == 0 do
            // virtual length doesn't change, check whether insertion causes eviction in LRU k,
               no effect on other LRUs
            while k→attrib_size > k→alloc_size do
                evict(tails[k], k) // evict the tail item if LRU k is full
            end
        end
        case diff > 0 do
            deflate(it, k, $\mathcal{P}$) // decreased virtual length causes deflation
        end
        case diff < 0 do
            inflate(it, $\mathcal{P}$) // increased virtual length causes inflation
        end
    end
    return
```

```
Function process_command(char *key, char *value, uint8_t command, libevent_thread *k):
    switch command do
        case GET do
            it = find_item(key) // search item in hash table
            if it then
                // physical cache hit
                if k has it then
                    // proxy-LRU hit
                    bump item it to head of proxy-LRU k
                else
                    // proxy-LRU miss
                    link item it to proxy-LRU k
                    k → attrib_size += it → virtual_len
                    k → avail_size -= it → virtual_len
                    $\mathcal{P}$ = {proxies sharing item it}
                    deflate(it, k, $\mathcal{P}$)
                end
                return it
            else
                // physical cache miss
                return NULL
            end
        end
        case SET do
            new = alloc_item(key,value)
            old = find_item(key)
            if old then
                // key already exists, replace item old with item new
                replace item old by item new in all proxy-LRUs
                new→virtual_len = old→virtual_len
                evict item old from physical cache and hash table
                if k has item new then
                    unlink item new in LRU k
                    insert(new, k, false) // promote
                else
                    insert(new, k, true)
                end
            else
                insert(new, k, true) // key doesn't exist, insert item new
            end
        end
    end
    return
```