Caching with Reserves

Sharat Ibrahimpur ✉
University of Waterloo, Canada

Manish Purohit ✉
Google Research, Mountain View, CA, USA

Zoya Svitkina ✉
Google Research, Mountain View, CA, USA

Erik Vee ✉
Google Research, Mountain View, CA, USA

Joshua R. Wang ✉
Google Research, Mountain View, CA, USA

Abstract

Caching is among the most well-studied topics in algorithm design, in part because it is such a fundamental component of many computer systems. Much of traditional caching research studies cache management for a single-user or single-processor environment. In this paper, we propose two related generalizations of the classical caching problem that capture issues that arise in a multi-user or multi-processor environment. In the *caching with reserves* problem, a caching algorithm is required to maintain at least k_i pages belonging to user i in the cache at any time, for some given reserve capacities k_i. In the *public-private caching* problem, the cache of total size k is partitioned into subcaches, a private cache of size k_i for each user i and a shared public cache usable by any user. In both of these models, as in the classical caching framework, the objective of the algorithm is to dynamically maintain the cache so as to minimize the total number of cache misses.

We show that *caching with reserves* and *public-private caching* models are equivalent up to constant factors, and thus focus on the former. Unlike classical caching, both of these models turn out to be NP-hard even in the offline setting, where the page sequence is known in advance. For the offline setting, we design a 2-approximation algorithm, whose analysis carefully keeps track of a potential function to bound the cost. In the online setting, we first design an $O(\ln k)$-competitive fractional algorithm using the primal-dual framework, and then show how to convert it online to a randomized integral algorithm with the same guarantee.

2012 ACM Subject Classification Theory of computation → Caching and paging algorithms

Keywords and phrases Approximation Algorithms, Online Algorithms, Caching

Digital Object Identifier 10.4230/LIPIcs.APPROX/RANDOM.2022.52

Category APPROX

Related Version *Full Version*: `https://arxiv.org/abs/2207.05975` [11]

Acknowledgements We would like to thank Sungjin Im for suggesting the caching with reserves problem and for useful discussions.

1 Introduction

Caching is one of the most well-studied problems in online computation and also one of the most crucial components of many computer systems. In the classical caching (also referred to as paging) problem, page requests arrive online and an algorithm must maintain a small set of pages to hold in a cache so as to minimize the number of requests that are not served from the cache. Caching algorithms have been widely studied through the lens of competitive analysis and tight results are known [1, 10, 14]. Tight algorithms are also known for many generalizations such as weighted paging [3, 4], generalized caching [2, 5] and paging with rejection penalties [9]. Due to its practical importance, a large number of heuristic algorithms have been proposed such as Least Recently Used (LRU), Least Frequently Used (LFU), CAR [7], ARC [15], and many others. Although they do not provide the best worst-case performance, they attempt to maximize the hit rate of the cache on practical instances. However, such traditional caching policies (both theoretical and practical) attempt to optimize the global efficiency of the system and are not necessarily suitable for cache management in a multi-user or multi-processor environment. In many of today's cloud computing services, caches are shared among all the users utilizing the service and optimizing only for global efficiency can lead to highly undesirable allocation for some users. For example, a user who only accesses pages at long intervals may reap no benefit from the cache at all. In this paper, we propose two generalizations of the classical caching problem that are suited for caching in a shared multi-processor environment.

In a multi-user setting, a naive way to guarantee that all users benefit from the cache is to partition the cache among them and effectively maintain separate caches for each user. However, such a system can be extremely inefficient and lead to low overall throughput as the cache can remain underutilized. Instead, a number of recent systems [12, 13, 16, 17] aim to maximize the global efficiency of the cache while attempting to provide (approximate) *isolation guarantees* to each user, i.e., the cache hit rate for each user is at least as much as what it would be if the user was allocated its own isolated cache (of proportionally smaller size). We model the multi-user scenario as the *caching with reserves* problem wherein a caching algorithm is required to maintain at least k_i pages belonging to user i in the cache at any time for some input reserve capacities k_i. As in the classical caching framework, the objective of the algorithm is to dynamically maintain the cache so as to minimize the total number of cache misses. The reserve capacities for users provide an implicit isolation guarantee since k_i cache slots are reserved for pages of user i. We remark that when the reserve capacities are all zero, then the problem reduces to classical unweighted caching.

A similar issue arises in the multi-processor setting where we have different "levels" of caches. Lower-level caches tend to be smaller and dedicated to a particular processor, while higher-level caches can be used by multiple processors and are larger in size. Consider a system with m separate processors, each of which has its own independent cache. In addition, there is a separate public cache shared by all the processors. We model such a setting as the *public-private caching* problem where a cache of total size k is partitioned into $(m + 1)$ subcaches, one private cache for each user and a shared public cache. In contrast with classical caching, in this case cache slots themselves have identities and a page requested by user i cannot be placed in a cache slot that belongs to the private cache of some other user j.

1.1 Our Contributions

We propose and study the *caching with reserves* and *public-private caching* problems. We show that the two problems are equivalent up to constant factors (Section 3).

▶ **Proposition 1.** *If $\mathcal{A}$ is a c-competitive online algorithm for* caching with reserves*, then there exists an online algorithm $\mathcal{A}'$ that is $2c$-competitive for* public-private caching*. Similarly, if $\mathcal{B}$ is a c-competitive online algorithm for* public-private caching*, then there exists an online algorithm $\mathcal{B}'$ that is $2c$-competitive for* caching with reserves*.*

Our next set of results considers the *offline* scenario where the entire request sequence is known in advance. Recall that in the classical setting, there is a simple exact solution (Belady's algorithm [8], which evicts the page that is requested farthest in the future). In our more complex setting, we show both variations are NP-hard.

▶ **Theorem 2.** *Both the* offline caching with reserves *problem and the* offline public-private caching *problem are strongly NP-hard.*

We defer the full proof of the NP-hardness to the full version of the paper [11] and note here the key difficulty in our reduction from 3-SAT. A naive strategy to reduce 3-SAT to our problem is to try to transform boolean variable assignments (e.g. $x_1 = T, x_2 = F$) into the contents of cache at a particular point in time (e.g., agent 1 has its "true" page in cache and agent 2 has its "false" page in cache). This runs into a stumbling block: to check that a clause is satisfied, one needs to request the relevant pages. Since we only expect one of them to actually be in cache, this provides the opportunity for a cheating solution to swap the contents of cache. Our construction sidesteps this issue by embracing page swapping and instead demanding that a variable assignment be encoded as a particular sequence of page swaps.

Despite this hardness result, we still provide constant-approximation algorithms in the offline setting. Due to the equivalence of the two models, we focus on *caching with reserves* problem for the rest of the paper. We give a 2-approximation algorithm in Section 4 for the offline setting. It is a non-trivial adaptation of Belady's algorithm to the multi-agent setting. The analysis utilizes a potential function that was recently proposed to give an alternative proof of optimality for Belady's algorithm [6]. It tracks how far in the future the cached pages are for the algorithm vs. the optimum.

▶ **Theorem 3.** *There is a* 2*-approximation algorithm for offline caching with reserves.*

In the *online* scenario, where the algorithm knows nothing about page requests until they occur, we give a fractional algorithm (which may keep pages fractionally in cache) using the primal-dual framework (Section 5).

▶ **Theorem 4.** *There is a* $2\ln(k+1)$*-competitive fractional algorithm for online caching with reserves.*

We also show that the fractional solution can be rounded online in a way that preserves the competitive ratio up to a constant, obtaining an online randomized (integral) algorithm (Section 6).

▶ **Theorem 5.** *There is an* $O(\ln k)$*-competitive integral algorithm for online caching with reserves.*

2 Preliminaries and Notation

In the classical caching problem, we are given $\mathcal{U}$, a universe of n pages, together with a cache of size k. At each time step, at most k pages are in cache. We are presented a sequence of page requests $\sigma = \langle p_1, p_2, \ldots \rangle$, where each $p_t \in \mathcal{U}$. At time t, page p_t arrives. If p_t is not in cache, then a *cache miss* occurs and the algorithm incurs unit cost. It must then fetch page p_t into the cache, possibly by evicting some other page from the cache. That is, if there would be $k+1$ pages in cache, the algorithm must remove some page other than p_t from cache. An *online* algorithm makes the eviction choice without knowing the future request sequence, whereas an *offline* algorithm is assumed to know the entire request sequence in advance.

Motivated by applications in multi-processor caching and shared cache systems, we define two new related problems. Let $\mathcal{I} = \{1, \ldots, m\}$ be a set of m agents and suppose that the universe $\mathcal{U}$ is a disjoint union of pages belonging to each agent, i.e., $\mathcal{U} = \sqcup_{i \in \mathcal{I}} \mathcal{U}(i)$. In the *public-private caching* model, the cache of total size k is subdivided as follows: each agent $i \in \mathcal{I}$ is allocated k_i cache slots and the remaining $k_0 \triangleq k - \sum_{i\in\mathcal{I}} k_i$ slots are *public*.* In this model, only pages belonging to agent i can be placed in any of the k_i cache slots allocated to agent i, while any page can be held in the public slots. As in the traditional caching problem, the goal of the algorithm is to minimize the total number of evictions. In the *caching with reserves* model, the cache is not divided, but instead for each agent $i \in \mathcal{I}$, the algorithm is required to maintain at least k_i pages from $\mathcal{U}(i)$ in the cache at any time. To avoid any complications, we assume that we already have pages in cache that meet this constraint at the start of the algorithm. (These may be dummy pages that are never requested during the actual sequence.) Throughout, we let $n_i = |\mathcal{U}(i)|$ denote the number of distinct pages owned by agent i, and for any page $p \in \mathcal{U}(i)$, we let $ag(p) = i$ be the agent that owns page p.

We analyze the online algorithm in terms of its *competitive ratio*. This is the maximum ratio, over all possible problem instances, of the cost incurred by the algorithm over the cost of the optimal offline solution of this instance.

3 Equivalence of Public-Private Caching and Caching with Reserves

We now prove Proposition 1 (restated below for convenience), showing that the two models defined in the introduction are equivalent up to constant factors.

▶ **Proposition 1.** *If $\mathcal{A}$ is a c-competitive online algorithm for* caching with reserves*, then there exists an online algorithm $\mathcal{A}'$ that is $2c$-competitive for* public-private caching*. Similarly, if $\mathcal{B}$ is a c-competitive online algorithm for* public-private caching*, then there exists an online algorithm $\mathcal{B}'$ that is $2c$-competitive for* caching with reserves*.*

Proof. We first explain how to convert back-and-forth between caching strategies for the two problems. Note that both of the following conversions can be done online and we will maintain that the cache states in the two problems are identical after every page request.

The easy direction is turning a public-private caching strategy into a caching with reserves strategy. Suppose a page request p comes in. If p is in cache, then we do not evict any page in either strategy. If it is not, then the public-private caching strategy evicts some page q to make room for it. Our caching with reserves strategy makes precisely the same eviction, which we show maintains the reserve constraints:

- If evicted page q was in a private cache, then p is placed in that same private cache. Hence, p and q have the same agent i. And agent i has the same number of pages before and after the arrival of p, maintaining our reserve constraint.
- If q was in a public cache, then let i be the agent that owns q. Agent i must have at least k_i pages in cache other than q, namely the k_i pages in its private cache. So evicting q does not put agent i below its reserve for our caching with reserves algorithm.

We now turn to the harder case of turning a caching with reserves strategy into a public-private caching strategy. To keep the analysis clean, we permit the public-private caching strategy to perform extra evictions at any step (but it is still charged for each one). Suppose a page request p comes in. If p is in cache, then we do not evict in either strategy. If it is not, then the caching with reserves strategy evicts some page q to make room for it, which belongs to some agent i. We can handle this with at most two evictions, as the following shows:

* We assume throughout the paper that $\sum_{i\in\mathcal{I}} k_i < k$. If $\sum_{i\in\mathcal{I}} k_i = k$, the problem can be solved as m separate instances of classical caching.

- If q is currently in the public cache, then we evict it and replace it with p, making the two caches match again.
- If q is currently in a private cache and the agent of p is also i, then we again can evict it and replace it with q, making the two caches match again.
- If q is currently in a private cache and the agent of p is not i, then agent i must have at least $k_i + 1$ pages in cache at the start of this step since we're about to evict q. Hence, there must be some page q' in public cache. "Move" q' into private cache by evicting both q and q', then placing q' into the slot previously occupied by q. We can then place p into the slot previously occupied by q' in public cache.

We now have conversions between the two problems that approximately preserve the number of evictions, and are ready to prove the main claim. We will use τ_e to denote the first transformation, from public-private caching strategies into caching with reserves strategies. We will use τ_h to denote the second transformation, from caching with reserves strategies to public-private caching strategies.

Suppose we have some algorithm $\mathcal{A}$ for caching with reserves, and let $\mathcal{A}' \triangleq \tau_h(\mathcal{A})$. Furthermore, let the optimal solutions to caching with reserves and public-private caching be $\mathcal{O}_{cr}$ and $\mathcal{O}_{ppc}$, respectively.

$$\begin{aligned}
\text{evictions}(\mathcal{A}') &\leq 2 \cdot \text{evictions}(\mathcal{A}) && \text{Transformation Guarantee} \\
&\leq 2c \cdot \text{evictions}(\mathcal{O}_{cr}) && \mathcal{A} \text{ is a } c\text{-approximation} \\
&\leq 2c \cdot \text{evictions}(\tau_e(\mathcal{O}_{ppc})) && \mathcal{O}_{cr} \text{ Optimality} \\
&\leq 2c \cdot \text{evictions}(\mathcal{O}_{ppc}) && \text{Transformation Guarantee}
\end{aligned}$$

Similarly, suppose we have some algorithm $\mathcal{B}$ for public-private caching and let $\mathcal{B}' \triangleq \tau_e(\mathcal{B})$. Again, let the optimal solutions to caching with reserves and public-private caching be $\mathcal{O}_{cr}$ and $\mathcal{O}_{ppc}$, respectively.

$$\begin{aligned}
\text{evictions}(\mathcal{B}') &\leq \text{evictions}(\mathcal{B}) && \text{Transformation Guarantee} \\
&\leq c \cdot \text{evictions}(\mathcal{O}_{ppc}) && \mathcal{B} \text{ is a } c\text{-approximation} \\
&\leq c \cdot \text{evictions}(\tau_h(\mathcal{O}_{cr})) && \mathcal{O}_{ppc} \text{ Optimality} \\
&\leq 2c \cdot \text{evictions}(\mathcal{O}_{cr}) && \text{Transformation Guarantee}
\end{aligned}$$

This completes the proof. ◀

4 Offline Caching with Reserves

In this section, we present a 2-approximation algorithm for the offline caching with reserves problem. The algorithm itself can be thought of as a generalization to Belady's classic Farthest-in-Future algorithm [8]. Indeed, the algorithm we present reduces to it in the trivial case that $k_i = 0$ for all i. However, in general, in our setting, there are cases where the farthest-in-future page cannot be evicted due to the reserve constraints.

Our algorithm maintains a partition of the pages in cache into sets N_i. For $i > 0$, the set N_i consists only of pages for agent i; further, we maintain $|N_i| = k_i$ at the beginning of each time step. The set N_0 contains the remaining cached pages. When a page p associated with agent i arrives and is not already in cache, we insert it into N_i. This causes $|N_i| = k_i + 1$, so we move the farthest-in-future page from N_i to N_0. This, in turn, causes N_0 to be too large. So we evict the farthest-in-future page from N_0. Notice that we are always allowed to evict such a page, since we maintain k_i pages of agent i in each N_i. In the case that p arrives but is already in N_0, we first move it to N_i, then proceed similarly. In this way, an arriving page always "passes through" N_i. The full details are in Algorithm 1.

Algorithm 1 Offline algorithm for caching with reserves.

```
Let N ← set of pages in the cache initially
Partition N = ⊔_{i=0}^m N_i where each N_i (for i ≠ 0) contains some arbitrary k_i pages belonging
  to agent i and N_0 contains all the remaining pages
Set rank(q), for each page q, to the time of q's first request
for each requested page p do
    Let i = ag(p)
    if p ∈ N_i then                          /* Cache hit in a set reserved for i.  */
        Serve page p from cache
    else if p ∈ N_0 then                     /* Cache hit in a set not reserved for i.  */
        Serve page p from cache
        /* Move p from N_0 to N_i.  */
        N_i ← N_i ∪ {p} and N_0 ← N_0 \ {p}
        /* Move highest-ranked page from N_i to N_0.  */
        Let q_i ∈ N_i be the page in N_i with maximum rank (if k_i = 0, this will be p)
        N_i ← N_i \ {q_i} and N_0 ← N_0 ∪ {q_i}
    else                                                              /* Cache miss.  */
        /* Add p to N_i, then move highest-ranked page from N_i to N_0.  */
        N_i ← N_i ∪ {p}
        Let q_i ∈ N_i be the page in N_i with maximum rank (if k_i = 0, this will be p)
        N_i ← N_i \ {q_i} and N_0 ← N_0 ∪ {q_i}
        /* Evict highest-ranked page from N_0.  */
        Let q be the page in N_0 with maximum rank (q ≠ p even if q_i = p)
        N_0 ← N_0 \ {q}
        Evict page q, fetch page p into cache and serve it
    Set rank(p) to the time of p's next request (if none, set it later than the last request)
```

Our analysis proving the 2-approximation generalizes a potential argument for Belady's algorithm (proposed recently [6]), but is technically more complicated due to the multi-tiered approach we take. The proof compares our sets N_i with sets N_i^* for the optimal algorithm. (To be more precise, the optimal algorithm maintains a certain set of pages in cache at each time step. We define a partition of these pages into the N_i^* such that each N_i^* consists only of pages from agent i, and $|N_i^*| = k_i$ at the beginning of each time step.) We call any page's next request time its *rank*. We define, for any rank s, the value $n_i(s)$ to be the number of pages in the set N_i with rank at least s at a given time. Similarly, $n_i^*(s)$ is the number of pages in the set N_i^* with rank at least s.[†]

We define our potential function as

$$\Phi = \sum_{i=0}^{m} \phi_i \ , \quad \text{where } \phi_i = \max_s \ [n_i(s) - n_i^*(s)].$$

Notice that $\phi_i \geq 0$ for every i, because when s is larger than the rank of any page in cache, we have $n_i(s) = n_i^*(s) = 0$. Hence $\Phi \geq 0$.

We show that Algorithm 1 satisfies the requirements of Theorem 3 (restated below).

▶ **Theorem 3.** *There is a 2-approximation algorithm for offline caching with reserves.*

[†] The sets N_i and N_i^* and the quantities $n_i(s)$ and $n_i^*(s)$ vary over time, but we suppress the dependence on t in the notation for brevity.

The proof requires repeated reasoning about how the potential Φ changes with each step. For example, adding a page to N_i will increase ϕ_i by at most 1 (and possibly leave it unchanged). However, adding a page p to N_i whose rank is higher than anything in N_i^* guarantees that ϕ_i will increase by exactly 1 (since $n_i(s)$ increases by 1 for every $s \leq rank(p)$).

Initially let $N_i^* = N_i$ for all i from 0 to m (the sets N_i are initialized by Algorithm 1). Let ALG be the cost incurred by Algorithm 1 and OPT be the cost incurred by an optimal algorithm. Let $\Delta(ALG)$, $\Delta(\Phi)$, $\Delta(OPT)$ be incremental changes in ALG, Φ, OPT, respectively, with older value subtracted from the newer value.

▶ **Lemma 6.** *The runs of Algorithm 1 and of the optimal algorithm on a given sequence of page requests can be partitioned into steps such that for each step,* $\Delta(ALG) + \Delta(\Phi) \leq 2 \cdot \Delta(OPT)$.

Knowing Lemma 6, the approximation factor of 2 now follows from summing over all the incremental steps indexed by t, where $\cdot(t)$ is the value of each function after step t. We have $ALG(0) = \Phi(0) = OPT(0) = 0$ initially. By Lemma 6, for each t,

$$ALG(t) - ALG(t-1) + \Phi(t) - \Phi(t-1) \ \leq \ 2 \cdot (OPT(t) - OPT(t-1)).$$

Summing over all t (up to the last step T) and telescoping,

$$\begin{aligned} ALG(T) - ALG(0) + \Phi(T) - \Phi(0) \ &\leq \ 2 \cdot (OPT(T) - OPT(0)) \\ ALG(T) \ &\leq \ 2 \cdot OPT(T), \end{aligned}$$

where the last inequality uses $\Phi(T) \geq 0$.

Proof of Lemma 6. To prove Lemma 6, we break the runs of Algorithm 1 and the optimal algorithm (together with updates to sets N_i^*) into steps, and for each step show that $\Delta(ALG) + \Delta(\Phi) \leq 2 \cdot \Delta(OPT)$. All the steps below constitute the processing of one request for a page p belonging to agent i. Let $\delta_i(s) = n_i(s) - n_i^*(s)$, so that $\phi_i = \max_s \delta_i(s)$.

Step 1 (Add p to both N_i and N_i^*)

Update $N_i \leftarrow N_i \cup \{p\}$ and $N_i^* \leftarrow N_i^* \cup \{p\}$.

Neither ALG nor OPT changes in this step, since we don't evict anything. In addition, the potential Φ doesn't increase. To see this, we'll use the fact that the rank of p is the smallest among any page in cache (for our algorithm as well as for the optimal algorithm), since it is the page that has just arrived. We consider four cases based on whether N_i and N_i^* contained p before this step.

- If both N_i and N_i^* contained p already, then nothing changes.
- If neither contained it, then both $n_i(s)$ and $n_i^*(s)$ increase by 1 for all $s \leq rank(p)$, so their difference is unchanged.
- If p was newly added only to N_i^*, then Φ can only decrease.
- The remaining case is that p was newly added only to N_i. Note that since p is the page that was just requested (and its rank hasn't been updated to the next occurrence yet), it has the minimum rank of all pages. We prove that Φ doesn't increase by showing that before this step, $\phi_i \geq 1$, and after this step, any $\delta_i(\cdot)$ that might have changed are at most 1. Specifically, before this step, $|N_i| = |N_i^*| = k_i$. Since N_i did not contain p, and all other pages have higher rank, before this step we had $n_i(rank(p)+1) = k_i$. Since N_i^* contained p, we had $n_i^*(rank(p)+1) = k_i - 1$. Thus, before this step, $\phi_i \geq \delta_i(rank(p)+1) = 1$. After this step, $n_i(s) = k_i + 1$, $n_i^*(s) = k_i$, and $\delta_i(s) = 1$ for $s \leq rank(p)$ (and $\delta_i(s)$ is unchanged for $s > rank(p)$). Thus, Φ doesn't increase.

Step 2 (Remove p from both N_0 and N_0^*)

Update $N_0 \leftarrow N_0 \setminus \{p\}$ and $N_0^* \leftarrow N_0^* \setminus \{p\}$.

Again, ALG and OPT don't change since we make no evictions. Further, removing p – the lowest-ranked page in cache for both our algorithm and the optimal algorithm – does not increase Φ; the reasoning is similar to above.

- If neither N_0 nor N_0^* changes, then Φ remains the same.
- If p is newly removed from both, then $n_0(s)$ and $n_0^*(s)$ decrease by 1 for all $s \leq rank(p)$, and $\delta_0(s)$ for all s are unchanged.
- If p is newly removed only from N_0, Φ can only decrease.
- The remaining case is that p was newly removed only from N_0^*. Before this step, $|N_0| = |N_0^*| = k_0$. Since p is the page with minimum rank, before the step, $n_0(s) = n_0^*(s) = k_0$ for $s \leq rank(p)$. Also, since before the step $p \notin N_0$ and $p \in N_0^*$, we had $n_0(rank(p)+1) = k_0$ and $n_0^*(rank(p)+1) = k_0 - 1$, implying $\Phi \geq \delta_0(rank(p)+1) = 1$. After the removal of p, $n_0(s) = k_0$, $n_0^*(s) = k_0 - 1$ and $\delta_0(s) = 1$ for $s \leq rank(p)$. Thus, Φ doesn't increase.

Step 3 (Ensure $|N_i| = |N_i^*| = k_i$)

In Step 1, we added p to N_i (resp., N_i^*). If it wasn't already there, we increased the size by 1. If that happened, then in this step, we move a page from N_i to N_0 to ensure $|N_i| = k_i$ (resp., move from N_i^* to N_0^* to ensure $|N_i^*| = k_i$). Let q_i be the page in N_i with maximum rank. If $|N_i| = k_i + 1$, then q_i is moved to N_0, consistent with Algorithm 1. We choose which page to move from N_i^* to N_0^* based on the cases below. It could be the page p itself if it is the only one available, the page $q \in N_i^*$ with minimum rank other than p (so it actually has the second-minimum rank in N_i^*), or the page $q_i^* \in N_i^*$ with maximum rank. ALG and OPT don't change in this step, and in each case we show that Φ doesn't increase.

- If $k_i = 0$, then $N_i = N_i^* = \{p\}$. Move p from N_i to N_0 and from N_i^* to N_0^*.
  Φ is unaffected in this case because for any s, $n_i(s)$ changes by the same amount as $n_i^*(s)$, and $n_0(s)$ changes by the same amount as $n_0^*(s)$.
  All the cases below assume that $k_i > 0$.
- If $|N_i| = k_i + 1$ but $|N_i^*| = k_i$, move q_i from N_i to N_0.
  We show that when q_i is removed from N_i, ϕ_i decreases by 1. Since N_i had more pages than N_i^*, before this step $\phi_i \geq 1$. Also before this step, $\delta_i(s) \leq 0$ for $s > rank(q_i)$ (since $n_i(s) = 0$ for those s), so the maximum was not achieved for those values of s. And for $s \leq rank(q_i)$, $\delta_i(s)$ decreases by 1 after this step, leading to the decrease of ϕ_i. Now, when q_i is added to N_0, ϕ_0 increases by at most 1. But this is compensated by the decrease in ϕ_i, showing that overall Φ doesn't increase.
- If $|N_i| = k_i$ but $|N_i^*| = k_i + 1$, move the second-lowest-ranked page $q \in N_i^*$ to N_0^*. Note that by our assumption that $k_i > 0$, N_i^* has at least two pages.
  Adding a page to N_0^* can only decrease the potential. Now we consider the effect on ϕ_i of removing q from N_i^*. We show that for any s for which $\delta_i(s)$ could have changed, it was negative before this step. For any $s > rank(q)$, $\delta_i(s)$ doesn't change. Note that page p has minimum rank in both N_i and N_i^*. So, before this step, for $s \leq rank(p)$, $n_i^*(s) = |N_i^*| = k_i + 1$ and $n_i(s) = |N_i| = k_i$, so $\delta_i(s) < 0$. For $s \in (rank(p), rank(q)]$, $n_i^*(s) = k_i$ and $n_i(s) \leq k_i - 1$, so again $\delta_i(s) < 0$. Thus when $\delta_i(s)$ for $s \leq rank(q)$ increases by 1, it remains at most 0, and does not increase Φ (which is always at least 0).
- Recall that $q_i \in N_i$ and $q_i^* \in N_i^*$ are the pages with maximum ranks in the respective sets. If $|N_i| = |N_i^*| = k_i + 1$ and $rank(q_i) \leq rank(q_i^*)$, move q_i from N_i to N_0 and q_i^* from N_i^* to N_0^*.

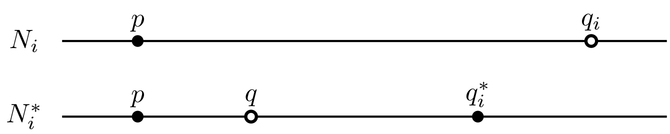

Figure 1 Illustration for the last case of Step 3 in the proof of Lemma 6.

We first consider the removal of q_i from N_i and of q_i^* from N_i^*. For $s \le rank(q_i)$, both $n_i(s)$ and $n_i^*(s)$ decrease by 1, so $\delta_i(s)$ doesn't change. For $s > rank(q_i^*)$, $n_i(s)$, $n_i^*(s)$, and $\delta_i(s)$ are unchanged. For $s \in (rank(q_i), rank(q_i^*)]$, before this step we had $n_i(s) = 0$ and $n_i^*(s) \ge 1$, with $\delta_i(s) \le -1$. So increasing $\delta_i(s)$ by 1 for these s does not change Φ. Now we consider the addition of q_i to N_0 and of q_i^* to N_0^*. For any s, $n_0^*(s)$ increases at least as much as $n_0(s)$ does, so Φ does not increase.

- If $|N_i| = |N_i^*| = k_i + 1$ and $rank(q_i^*) < rank(q_i)$, move q_i to N_0 and the second-lowest-ranked page in N_i^* (call it q) to N_0^*. Note again that N_i^* has at least two pages.
  In this case ϕ_0 may increase by 1, but we show that this is offset by a decrease in ϕ_i. We analyze what happens for values of s in the intervals separated by three values: $rank(p) < rank(q) < rank(q_i)$ (see Figure 1). Before this step, $\delta_i(rank(q_i)) = n_i(rank(q_i)) - n_i^*(rank(q_i)) = 1 - 0 = 1$, so $\phi_i \ge 1$. Page p is the page with minimum rank in both N_i and N_i^*. For $s \le rank(p)$, before the step $\delta_i(s) = 0$, and it stays 0 after the step. For $s \in (rank(p), rank(q)]$, before the step $n_i^*(s) = |N_i^*| - 1 = k_i$ and $n_i(s) \le |N_i| - 1 = k_i$, so $\delta_i(s) \le 0$, and it stays that way. For $s > rank(q_i)$, also $\delta_i(s) = 0$ and stays 0. Thus, the maximum $\delta_i(s)$ was achieved for some $s \in (rank(q), rank(q_i))$. But in this interval, $n_i(s)$ decreases by 1, while $n_i^*(s)$ stays the same. Thus, the maximum $\delta_i(s)$ decreases by 1, causing ϕ_i to also decrease.

Step 4 (OPT moves)

If p was in cache, then the optimal algorithm doesn't do anything. Note that in this case, based on previous rearrangements, $|N_0^*| = k_0$. Neither OPT nor Φ changes. If p was not in cache, the optimal algorithm fetches p and evicts some page, say $q \in N_j^*$. Then $\Delta(\text{OPT}) = 1$. Also note that in this case the previous steps added p to $\bigcup_\ell N_\ell^*$, resulting in $|N_0^*| = k_0 + 1$. If $j = 0$, delete q from N_0^*. This restores $|N_0^*| = k_0$ and increases Φ by at most 1. If $j \ne 0$, then there must be some $q' \in N_0^*$ belonging to agent j (otherwise it would mean that agent j had only k_j pages in cache, and the optimal algorithm violated reserve sizes by evicting agent j's page). Move q' from N_0^* to N_j^* and delete q from N_j^*. This increases Φ by at most 2, satisfying the desired inequality.

Step 5 (ALG moves)

If p was in cache, then do nothing. Otherwise, fetch p and evict the page q with maximum rank in N_0, also deleting it from N_0. In this case, $\Delta(\text{ALG}) = 1$. We show that this is compensated by $\Delta(\Phi) = -1$. Before this step, we had $|N_0| = k_0 + 1$ but $|N_0^*| = k_0$, so $\phi_0 \ge 1$. For $s > rank(q)$, we had $\delta_0(s) \le 0$, and this doesn't change. So the maximum must have been achieved for $s \le rank(q)$, and $\delta_0(s)$ for those s decreases by 1.

Step 6 (Update the rank of p)

At this point, if $k_i = 0$, then $p \in N_0 \cap N_0^*$; otherwise, $p \in N_i \cap N_i^*$. In either case, changing $rank(p)$ preserves $\delta_0(s)$ and $\delta_i(s)$ for all s, so Φ is unchanged. ◀

This completes the proof of Lemma 6 and the proof of Theorem 3.

5 Online Caching with Reserves

In this section, we design an $O(\log k)$-competitive fractional online algorithm for caching with reserves. In particular, we prove Theorem 4, which is restated here for convenience. In Section 6, we show that any fractional algorithm for online caching with reserves can be *rounded* to obtain a randomized integral algorithm by losing only a constant factor in the competitive ratio. We remark that our rounding algorithm does not necessarily run in polynomial time.

▶ **Theorem 4.** *There is a $2\ln(k+1)$-competitive fractional algorithm for online caching with reserves.*

The fractional algorithm is based on the primal-dual framework and closely follows the analysis of [4]. As page requests arrive, the algorithm maintains a feasible solution to the primal LP, which corresponds to its eviction decisions, and an approximately feasible solution to the dual LP. The costs of these two solutions are within a factor 2 of each other. Using LP duality, this results in a bound on the cost of the algorithm compared to the optimum.

5.1 Notation

Consider some fixed page $p \in \mathcal{U}$, and let $t_{p,1} < t_{p,2} < \ldots$ be the time steps when page p is requested in the online sequence. For any $a \geq 0$, define $I(p,a) = \{t_{p,a}+1, \ldots, t_{p,a+1}-1\}$ to be the time interval between the ath and $(a+1)$st requests for page p (assume that $t_{p,0} = 0$ for all pages). Let $a(p,t)$ be the number of requests to page p that have been seen until time t (inclusive). Hence, by definition, for any time t, and any page $p \in \mathcal{U} \setminus \{p_t\}$, we have $t \in I(p, a(p,t))$. At any time t, an agent $i \in \mathcal{I}$ is said to be *tight* if exactly k_i pages of agent i are held in cache. Let $\mathcal{T}$ denote the set of tight agents.[‡]

5.2 Formulation

We use the variable $x(p,a) \in \{0,1\}$ to denote whether page p is evicted between its ath and $(a+1)$th request, i.e., in the interval $I(p,a)$ (where 1 denotes an eviction). We have the following linear programming relaxation and its dual formulation.[§]

Primal LP

$$\min \sum_{p\in\mathcal{U}} \sum_{a\geq 1} x(p,a)$$

subject to:

$$\sum_{p\in\mathcal{U}, p\neq p_t} x(p, a(p,t)) \geq n-k \qquad \forall t \qquad (1)$$

$$\sum_{p\in\mathcal{U}(i), p\neq p_t} x(p, a(p,t)) \leq n_i - k_i \qquad \forall t, \forall i \qquad (2)$$

$$x(p,a) \leq 1 \qquad \forall p, \forall a \qquad (3)$$

$$x \geq 0 \qquad (4)$$

Dual LP

$$\max \sum_t (n-k)\alpha(t) - \sum_{t,i} (n_i - k_i)\beta(t,i) - \sum_{p,a} \gamma(p,a)$$

subject to:

$$\sum_{t\in I(p,a)} \Big(\alpha(t) - \beta(t, ag(p))\Big) - \gamma(p,a) \leq 1 \qquad \forall p, \forall a \qquad (5)$$

$$\alpha, \beta, \gamma \geq 0 \qquad (6)$$

‡ The set of tight agents varies with the time t, but we suppress the dependence on t for convenience.

§ We assume without loss of generality that the algorithm is aware of the total number of pages belonging to each agent, $n_i = |\mathcal{U}(i)|$.

The primal objective simply measures the total number of evictions. The first constraint enforces that at any time t at least $n-k$ pages apart from p_t are outside the cache, which implies that at most k pages (including p_t) are inside the cache. The second constraint enforces that at any time, at most $(n_i - k_i)$ pages of agent i are outside cache (which implies that at least k_i pages are inside the cache). Note that this is true even if $p_t \in \mathcal{U}(i)$, since then we know that p_t must be in cache, so of the remaining $n_i - 1$ pages, at least $k_i - 1$ must be in the cache, so the total amount outside cache must be at most $(n_i - 1) - (k_i - 1) = n_i - k_i$.

5.3 Algorithm

For convenience, we assume without loss of generality that the cache is initialized to an arbitrary feasible configuration, i.e., each agent i has some arbitrary k_i pages in the cache, and the rest of the cache has k_0 other arbitrary pages. The LP variables are also initialized to reflect this initial configuration. At each time step, as a new page request arrives online, a new set of constraints for the primal LP are revealed, along with the corresponding new variables in the dual. All newly introduced variables are initialized to zero. Note that after the arrival of a new page request at time t, only the primal constraint (1) may now be unsatisfied; however, (2) and (3) remain feasible. So to maintain a feasible primal solution, we modify the primal (and dual) variables until Constraint (1) is satisfied. The online algorithm is required to maintain that all the primal variables $x(p, a)$ only monotonically increase over time. We remark that the dual solution that we maintain will always be approximately feasible. The violation in (5) is at most $O(\log k)$ at all times (Claim 8).

Algorithm 2 Fractional Online Algorithm for Caching with Reserves.

```
Let η ← 1/k
foreach request for page p at time t do
    Initialize x(p, a(p,t)) ← 0, α(t) ← 0, γ(p, a(p,t)) ← 0 and ∀i ∈ I, β(t,i) ← 0
    while primal constraint (1) is unsatisfied do
        Increase dual variable α(t) by dα
        foreach tight agent i ∈ T do
            Increase dual variable β(t,i) by dα
        foreach page q ∈ U do
            if ag(q) ∈ T then
                Do nothing
            else if x(q, r(q,t)) = 1 then
                Increase γ(q, r(q,t)) by dα
            else
                Increase x(q, r(q,t)) by dx = (x(q, r(q,t)) + η)dα
```

5.4 Analysis

First, we note that the primal solution that we construct is feasible by design.

▷ Claim 7. At all times t, we maintain the inequality: Primal Objective $\leq 2 \cdot$ Dual Objective.

Proof. At time $t = 0$, both the primal and dual solutions are initialized to have an objective of zero. Since the algorithm increases the primal and dual variables in a continuous fashion, consider any infinitesimal time step and let ΔP and ΔD denote the change in the primal and dual objectives in this step respectively. It suffices to show that $\Delta P \leq 2 \cdot \Delta D$ holds at all times.

Let $\mathcal{T}$ denote the set of agents that are tight during this step. Also partition the set $\mathcal{U} \setminus \{p\}$ into three parts: T is the set of pages belonging to tight agents, $E = \{q \in \mathcal{U} \setminus T \mid x(q, r(q,t)) = 1\}$ is the set of pages of non-tight agents that have been fully evicted, and S is the remaining set of pages. So we have $|T| + |S| + |E| = n - 1$, and $|T| = \sum_{i \in \mathcal{T}} n_i$. We also define $k' := k - \sum_{i \in \mathcal{T}} k_i$.

The change in the dual objective is given by:

$$\begin{aligned}\Delta D &= (n-k)d\alpha - \sum_{i \in \mathcal{T}} (n_i - k_i)d\alpha - |E|d\alpha = \Big(n - k - |T| + \sum_{i \in \mathcal{T}} k_i - |E|\Big)d\alpha \\ &= \Big(|S| - \big(k - \sum_{i \in \mathcal{T}} k_i\big) + 1\Big)d\alpha = (|S| - k' + 1)d\alpha\end{aligned}$$

On the other hand, the change in primal objective is given by:

$$\begin{aligned}\Delta P &= \sum_{q \in S} \big(x(q, r(q,t)) + \eta\big)d\alpha \\ &= \Big(\sum_{q \in \mathcal{U}\setminus\{p\}} x(q, r(q,t)) - \sum_{q \in T} x(q, r(q,t)) - \sum_{q \in E} x(q, r(q,t)) + |S|\eta\Big)d\alpha\end{aligned}$$

Since the variables are updated only as long as constraint (1) is not satisfied, we can bound the first term in the above expression by $n - k$. All pages in T belong to tight agents, so we have $\sum_{q \in T} x(q, r(q,t)) = \sum_{i \in \mathcal{T}} (n_i - k_i)$. Lastly, all pages in E have $x(q, r(q,t)) = 1$. So

$$\begin{aligned}\Delta P &\le \Big(n - k - \sum_{i \in \mathcal{T}} (n_i - k_i) - |E| + |S|\eta\Big)d\alpha = \Big(|S| - \big(k - \sum_{i \in \mathcal{F}} k_i\big) + 1 + |S|\eta\Big)d\alpha \\ &\le \big(|S| - k' + 1 + |S|/k'\big)d\alpha \qquad \text{(since } \eta = 1/k \le 1/k'\text{)} \\ &\le 2(|S| - k' + 1)d\alpha = 2 \cdot \Delta D\end{aligned}$$

It remains to justify the final inequality, which is equivalent to showing that $|S| \ge k'$. By definition, we have $|S| = n - 1 - |E| - |T|$. Since (1) is violated and (2) is tight for $i \in \mathcal{T}$, the following strict inequality holds:

$$\sum_{q \in S} x(q, r(q,t)) + |E| + \sum_{i \in \mathcal{T}} (n_i - k_i) = \sum_{q \in S \cup T \cup E} x(q, r(q,t)) < n - k.$$

Combining the above, we get $|S| > k' - 1$, which implies that $|S| \ge k'$. ◁

▷ Claim 8. Dual solution maintained by the algorithm is $\ln(k+1)$-approximately feasible.

Proof. Consider any page p and interval $I(p,a) = \{t_{p,a} + 1, \ldots, t_{p,a+1} - 1\}$. We show that the following inequality holds at all times:

$$\sum_{t \in I(p,a)} (\alpha(t) - \beta(t, ag(p))) - \gamma(p,a) \le \ln(k+1),$$

which implies dual feasibility of the solution (α, β, γ) scaled down by a factor $\ln(k+1)$.

We analyze the changes that occur in the LHS of the above inequality. We interpret the set $I(p,a)$ in an online fashion: time $t \in \{t_{p,a} + 1, \ldots, t_{p,a+1} - 1\}$ is included in $I(p,a)$ at the start of time step t. Note that $x(p,a) = 0$ and the LHS is 0 at the start of time $t_{p,a} + 1$. Over time, as page requests $p_t (\ne p)$ arrive during times $t \in \{t_{p,a} + 1, \ldots, t_{p,a+1} - 1\}$, the LHS increases whenever the $\alpha(t)$ variable increases, but there is no corresponding increase in the $\beta(t, ag(p))$ or $\gamma(p,a)$ variables. We couple such increases to increases in the primal variable $x(p,a)$. Note that $x(p,a)$ gets capped at 1, and after that $\gamma(p,a)$ is coupled with $\alpha(t)$.

At any infinitesimal step, if some $\alpha(t)$ increases by $d\alpha$, then we have one of three cases. *Case 1:* Agent $ag(p)$ is tight and $\beta(t, ag(p))$ increases by $d\alpha$; *Case 2:* $x(p,a) = 1$ and $\gamma(p,a)$ increases by $d\alpha$; *Case 3:* $x(p,a)$ increases by $dx = (x(p,a)+\eta)d\alpha$. In the first two cases, the LHS does not change at all, while in the second case, the LHS changes by $d\alpha$. So overall

$$d(\text{LHS}) = \left(\frac{1}{x(p,a)+\eta}\right) dx(p,a)$$

A straightforward integration gives:

$$\begin{aligned} LHS &= \int_0^X \left(\frac{1}{x(p,a)+\eta}\right) dx(p,a) && \text{(where } X \text{ is the final value of } x(p,a)) \\ &\leq \int_0^1 \left(\frac{1}{x(p,a)+\eta}\right) dx(p,a) \\ &= [\ln(x(p,a)+\eta)]_0^1 = \ln\left(\frac{1+\eta}{\eta}\right) = \ln(k+1) \end{aligned}$$ ◁

Proof of Theorem 4. The proof follows directly from the two claims above. Let (x,α,β,γ) denote the primal and dual variables constructed by Algorithm 2, and $(x^*,\alpha^*,\beta^*,\gamma^*)$ be the corresponding variables in the optimal solutions. Using LP duality for the last step, we have:

$$\begin{aligned} \sum_{p\in\mathcal{U}} \sum_{\substack{a\geq 1:\\ t_{p,a}\leq T}} x(p,a) &\leq 2\Big(\sum_t (n-k)\alpha(t) - \sum_{t,i}(n_i-k_i)\beta(t,i) - \sum_{p,a}\gamma(p,a)\Big) && \text{(by Claim 7)} \\ &\leq 2\ln(k+1)\Big(\sum_t (n-k)\alpha^*(t) - \sum_{t,i}(n_i-k_i)\beta^*(t,i) - \sum_{p,a}\gamma^*(p,a)\Big) && \text{(by Claim 8)} \\ &\leq 2\ln(k+1)\Big(\sum_{p\in\mathcal{U}} \sum_{\substack{a\geq 1:\\ t_{p,a}\leq T}} x^*(p,a)\Big) \end{aligned}$$ ◀

6 Rounding

We now describe an $O(1)$-approximate rounding scheme for the fractional algorithm of Section 5, thus proving Theorem 5.

▶ **Theorem 5.** *There is an $O(\ln k)$-competitive integral algorithm for online caching with reserves.*

Proof. For any time $t = 1, 2, \ldots$, the randomized integral algorithm will maintain a distribution μ^t of cache states such that for any page p, the probability that p is not in the cache (of the randomized algorithm) at time t is exactly $x^t(p, r(p,t))$, where x^t denotes the value of LP variable x at time t. By the design of our primal-dual algorithm, the x-variables never decrease, so the cost incurred by the fractional algorithm to serve page p_t is given by:

$$\text{cost}(t) := \sum_{p\in\mathcal{U}, p\neq p_t} \Big(x^{t+1}(p, r(p,t)) - x^t(p, r(p,t))\Big).$$

We will shortly describe how the integral algorithm moves from the distribution μ^t to μ^{t+1} while ensuring that the expected number of fetches and evictions is at most $O(\text{cost}(t))$. We remark that our rounding algorithm does not necessarily run in polynomial time. This is because the support size of μ^t can be super-polynomial in $|\mathcal{U}|$ and k. This is not an issue for competitive analysis of online algorithms, so we simply assume that we are maintaining a probability distribution over $\binom{|\mathcal{U}|}{k}$ cache states.

Fix some time t. For each page $p \in \mathcal{U} \setminus \{p_t\}$, define $y(p) := 1 - x^t(p, r(p,t))$ and $y'(p) := 1 - x^{t+1}(p, r(p,t))$ to be the portion of page p that is in the cache at the start of times t and $t+1$, respectively. Also define $y(p_t) = 1 - x^t(p_t, r(p_t,t))$ and $y'(p_t) := 1$; note that the fractional algorithm pays cost $1 - y(p_t)$ to fully fetch p_t into the cache by the end of time step t. With the above notation, for any page $p \in \mathcal{U}$, we have $\Pr_{C\sim\mu^t}[p \in C] = y(p)$ and $\Pr_{C\sim\mu^{t+1}}[p \in C] = y'(p)$.

To simplify the description of our rounding scheme, we further assume that the changes that occur in the primal solution between states x^t and x^{t+1} do so through a sequence of smaller changes where the x-value changes for exactly two pages (and hence the y-value also changes for exactly two pages). Let $p, q \in \mathcal{U}$ and $\epsilon \in [0,1]$ be such that $y'(p) = y(p) + \epsilon$, $y'(q) = y(q) - \epsilon$, and $y'(p') = y(p')$ for all $p' \in \mathcal{U} \setminus \{p, q\}$.¶ Let μ, μ' denote distributions over integral cache states that agree with y and y', respectively. The cost incurred by the fractional algorithm to move from y to y' is exactly ϵ (because it only pays for evictions). We now describe how the integral algorithm moves from μ to μ' by incurring a cost of at most 6ϵ. To modify a δ probability measure of the cache-state from C to C', the integral algorithm pays a cost of $\delta \cdot |C \setminus C'|$. We divide the modification steps into three phases:

1. **Fixing the marginals:** In this phase, we modify the distribution μ so that for any page $p' \in \mathcal{U}$, $\Pr_{C\sim\mu}[p' \in C]$ changes from $y(p')$ to $y'(p')$. We accomplish this by: (i) adding p to an ϵ probability measure of cache states from μ that do not contain p; and (ii) removing q from an ϵ measure of cache states from μ that contain q. The cost incurred in this step is exactly ϵ.

   By the end of this phase, for any (possibly infeasible) cache state C in μ, we have $|C| \in \{k-1, k, k+1\}$. Let $0 \le \epsilon_1 \le \epsilon$ denote the probability measure of cache states with exactly $k-1$ pages. By the description of the modification step, it is clear that *exactly* ϵ_1 measure of cache states have cardinality $k+1$. Further, let $0 \le \epsilon_2 \le \epsilon$ denote the measure of cache states that violate the reserve constraint for some agent. Since only removing the page q could lead to a constraint violation, we must have $|C| \in \{k-1, k\}$ for any such violating cache state.
2. **Fixing the size:** In this phase, we match an ϵ_1 measure of cache-states of size $k-1$ with an ϵ_1 measure of cache-states of size $k+1$. Let C and C' denote page-sets of size $k-1$ and $k+1$, respectively, that are matched with some positive measure α. Pick an arbitrary page $p' \in C' \setminus C$. We remove p' from an α measure of state C', and add it to an α measure of state C. The total cost incurred in this phase is exactly $\epsilon_1 \le \epsilon$.

   By the end of this phase, all cache-states have cardinality exactly k. However, the removal of the page p' above may cause violations of the reserve constraint. Let $\epsilon_3 \in [0,1]$ denote the measure of cache states that satisfied all reserve constraints at the end of the first phase, but now violate some reserve constraint. By the above discussion, such cache states arise from the removal of page $p' \in C' \setminus C$ from C' (that had size $k+1$), so $\epsilon_3 \le \epsilon_1$. Overall, exactly $\epsilon_2 + \epsilon_3$ measure of cache states violate some reserve constraint. In fact, every violated cache state violates a single reserve constraint.

¶ Here, p plays the role of page p_t that is fetched into the cache, and q plays the role of pages in $\mathcal{U} \setminus \{p_t\}$ that are evicted to make space for p_t.

3. **Fixing the violated reserve constraint:** We now fix all violated reserve constraints by matching an $\epsilon_2 + \epsilon_3$ measure of cache states with exactly an $\epsilon_2 + \epsilon_3$ measure of cache states that have an excess in that reserve constraint. More precisely, if C is a cache state that violates the reserve constraint for agent $i \in \mathcal{I}$, then we match an $\alpha > 0$ measure of C with another cache state C' that satisfies $|C' \cap \mathcal{U}(i)| \geq k_i + 1$. Such a matching exists because the fractional solution y' satisfies all reserve constraints and (by the end of the first phase we ensured that) the distribution μ satisfies the reserve constraint in expectation: for every cache state C with $|C \cap \mathcal{U}(i)| < k_i$, there must exist another cache state C' with $|C' \cap \mathcal{U}(i)| > k_i$. We move an arbitrary page $p' \in \mathcal{U}(i) \cap (C' \setminus C)$ from C' to C. In exchange for p', we move an arbitrary page $q' \in (\mathcal{U} \setminus \mathcal{U}(i)) \cap (C \setminus C')$ from C to C' that does not violate any reserve constraints for the state C. The choice of q' is well-defined because the size of C is $k+1$ right after p' is moved from C' to C, and we also have $|C \cap \mathcal{U}(i)| = k_i$, so there must exist some other agent $j \neq i$ satisfying $|C \cap \mathcal{U}(j)| > k_j$. The cost incurred in this phase is at most $2(\epsilon_2 + \epsilon_3) \leq 4\epsilon$.
   At the end of this step, all cache states have size exactly k and satisfy all reserve constraints. The marginal probabilities in the resulting distribution μ' matches y'.

This completes the description of our rounding scheme. In the first step, we incurred a total cost of exactly ϵ while in the second and third steps, we incurred a total cost of at most $\epsilon_1 \leq \epsilon$ and $2(\epsilon_2 + \epsilon_3) \leq 4\epsilon$. Since the fractional algorithm incurs a cost of ϵ, the theorem follows. ◀

References

1. Dimitris Achlioptas, Marek Chrobak, and John Noga. Competitive analysis of randomized paging algorithms. *TCS*, 234(1-2):203–218, 2000.
2. Anna Adamaszek, Artur Czumaj, Matthias Englert, and Harald Räcke. An $O(\log k)$-competitive algorithm for generalized caching. In *SODA*, pages 1681–1689, 2012.
3. Nikhil Bansal, Niv Buchbinder, and Joseph Naor. Towards the randomized k-server conjecture: A primal-dual approach. In *SODA*, pages 40–55, 2010.
4. Nikhil Bansal, Niv Buchbinder, and Joseph Naor. A primal-dual randomized algorithm for weighted paging. *Journal of the ACM (JACM)*, 59(4):1–24, 2012.
5. Nikhil Bansal, Niv Buchbinder, and Joseph Naor. Randomized competitive algorithms for generalized caching. *SICOMP*, 41(2):391–414, 2012.
6. Nikhil Bansal, Christian Coester, Ravi Kumar, Manish Purohit, and Erik Vee. Learning-augmented weighted paging. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 67–89. SIAM, 2022.
7. Sorav Bansal and Dharmendra S. Modha. CAR: Clock with adaptive replacement. In *3rd USENIX Conference on File and Storage Technologies (FAST 04)*, San Francisco, CA, March 2004. USENIX Association. URL: `https://www.usenix.org/conference/fast-04/car-clock-adaptive-replacement`.
8. L. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.
9. Leah Epstein, Csanád Imreh, Asaf Levin, and Judit Nagy-György. Online file caching with rejection penalties. *Algorithmica*, 71(2):279–306, 2015.
10. Amos Fiat, Richard M Karp, Michael Luby, Lyle A McGeoch, Daniel D Sleator, and Neal E Young. Competitive paging algorithms. *J. Algorithms*, 12(4):685–699, 1991.
11. Sharat Ibrahimpur, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua Wang. Caching with reserves, 2022. `arXiv:2207.05975`.

12 Wu Kan, Tu Kaiwei, Patel Yuvraj, Sen Rathijit, Park Kwanghyun, Arpaci-Dusseau Andrea, and Remzi Arpaci-Dusseau. NyxCache: Flexible and efficient multi-tenant persistent memory caching. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 1–16, Santa Clara, CA, February 2022. USENIX Association. URL: `https://www.usenix.org/conference/fast22/presentation/wu`.

13 Mayuresh Kunjir, Brandon Fain, Kamesh Munagala, and Shivnath Babu. Robus: fair cache allocation for data-parallel workloads. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 219–234, 2017.

14 Lyle A McGeoch and Daniel D Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6(1-6):816–825, 1991.

15 Nimrod Megiddo and Dharmendra S Modha. {ARC}: A {Self-Tuning}, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, 2003.

16 Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. {FairRide}:{Near-Optimal}, fair cache sharing. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 393–406, 2016.

17 Yinghao Yu, Wei Wang, Jun Zhang, and Khaled Ben Letaief. Lacs: Load-aware cache sharing with isolation guarantee. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 207–217. IEEE, 2019.