Dependency-Aware Online Caching

Julien Dallot*, Amirmehdi Fesharaki†, Maciej Pacut*, Stefan Schmid*
*TU Berlin, Germany, †Sharif University of Technology, Iran

***Abstract*—We consider a variant of the online caching problem where the items exhibit *dependencies* among each other: an item can reside in the cache only if all its dependent items are also in the cache. The dependency relations can form any directed acyclic graph. These requirements arise in systems such as CacheFlow (SOSR 2016) that cache forwarding rules for packet classification in IP-based communication networks.**

First, we present an optimal randomized online caching algorithm which accounts for dependencies among the items. Our randomized algorithm is $O(\log k)$-competitive, where k is the size of the cache, meaning that our algorithm never incurs the cost of $O(\log k)$ times higher than even an optimal algorithm that knows the future input sequence.

Second, we consider the bypassing model, where requests can be served at a fixed price without fetching the item and its dependencies into the cache — a variant of caching with dependencies introduced by Bienkowski et al. at SPAA 2017. For this setting, we give an $O(\sqrt{k \cdot \log k})$-competitive algorithm, which significantly improves the best known competitiveness. We conduct a small case study, to find out that our algorithm incurs on average 2x lower cost.

I. Introduction

Performance of most computer systems today rely on how well the caching algorithms manage their caches to avoid cache misses. Existing caching algorithms treat cached items as independent, however in some applications, items exhibit *dependencies* (modeled with a directed acyclic graph) among each other: an item can reside in the cache only if all its dependent items are also in the cache.

For example, dependency-aware caching has applications in communication networks, in IP packet classification [1] in network routers and switches (see Section I-A), where the goal is to cache the set of heavy hitter rules. We elaborate on the setting later in this section. In this paper, we present an algorithm with provable performance guaranties, thus proposing a rigorous, carefully analyzed alternative to the CacheFlow system [2].

Designing caching algorithms poses interesting challenges that can be overcome with a principled approach. In particular, as shifts in the traffic patterns (and hence also in the heavy hitter rules) may not be predictable, which calls for algorithms that operate in an online manner and adjust the cache in reaction to the traffic they see. A well-established method to deal with uncertainty of the future is the framework of online algorithms and competitive analysis [3], and the classic caching algorithms were often designed and analyzed for this setting [3, Ch. 3, 4] [4, Ch. 3]. Ideally, these online algorithms achieve a good competitive ratio: intuitively, without knowing the future demand, their performance is almost as good as a clairvoyant optimal offline algorithm.

To design algorithms that efficiently manage a cache in presence of dependencies, we generalize the well-known caching problem [5] to respect dependencies. The dependencies can form an arbitrary directed acyclic graph, see Figure 1 for an illustration. The objective of the algorithm is to minimize the number of fetches.

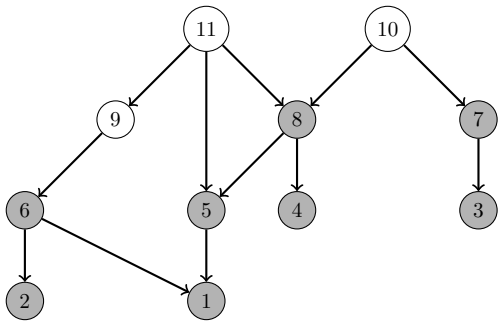


Fig. 1. Example directed acyclic graph of dependencies among items. An arrow from u to v means that if u is in the cache then also v must be in the cache. The size of the cache is $k = 8$, the items 9, 10 and 11 are not in the cache while the grayed items $1, 2, \ldots, 8$ are in the cache. If a request to 9 would arrive, we must choose what item to evict while keeping a feasible cache (8 or 7 are the only choices here) before we fetch 9.

Introducing dependencies among the cached items unravels algorithm design challenges unseen in the classic caching problem. Consider the situation when a cache miss occurs, and we need to choose a single item to evict. Then, we cannot freely pick any item from the cache to evict, as other items in the cache may depend on them. Hence, simply using a classic dependency-unaware algorithm such as Random Mark [6] leads to infeasible solutions, as other items may depend on the item chosen to be evicted. Furthermore, how to decide which item to evict? If we have two items that could be evicted, should we base the decision upon the number of descendants of these items? If we favor to evict an item with more descendants, then we would have more flexibility to choose items for future evictions. These questions become even more challenging when items can have multiple parents, and the dependency graph is not a tree.

The main contribution of our paper is an optimal randomized online *Bucketing* for the dependency-aware caching. Our analysis shows that our generalized algorithm is $O(\log k)$-competitive, where k is the size of the cache. We dive deeper into the analysis of our randomized algorithm with a parameterized analysis of the competitive ratio, revealing how the topology of the dependency graph influences the resulting competitive ratio, linking its competitiveness to the maximum independent set in the dependency graph.

As an additional contribution, we generalize our randomized algorithm to the setting with bypassing [7], [8], where requests can be served for a fixed price without fetching to the cache. In the context of packet classification, bypassing delegates classification of a packet to the centralized controller, similarly to the design introduced in CacheFlow [2]. This variant of caching with dependencies and bypassing was introduced by Bienkowski et al. at SPAA 2017, where the authors gave an $O(k \cdot h(T))$-competitive [9] where $h(T)$ is the height of the dependency graph T. By generalizing our Bucketing procedure, we design an algorithm that is $O(\sqrt{k \cdot \log k})$-competitive. Our result hence significantly improves the best known competitiveness, reaching the sublinear dependency on the cache size k.

A. Motivation: Packet Classification

Packet classification is a fundamental task in communication networks [1]. For example, each packet incoming to a router is matched against a set of predefined rules (e.g. does the packet match an IP range?) to determine how the router should handle the packet (e.g. send via a port); a packet may match multiple rules, and among them, the router uses the highest priority rule. The router then handles the packet according to the action associated with the rule.

The number of rules a router needs to store is growing rapidly for several reasons [10]. This introduces significant memory requirements, requiring more and more expensive and power-hungry hardware [11]. To address this problem, a natural approach, studied in systems such as CacheFlow [2] and TreeCaching by Bienkowski et al. at SPAA 2017 [9], is to store only a small subset of the rules at the router and the rest in a cheaper but potentially slower memory, e.g., at a (software-defined) network controller, essentially a two-level caching hierarchy. However, we cannot blindly use existing caching algorithms due to dependencies that arise among the rules with overlapping patterns. Precisely, to assure that the router correctly forwards packets with its cached subset of rules, if two rules overlap, the higher priority rule must be present in the cache when the lower priority rule is present.

The structure of the dependency graph depends on how general the packet forwarding rules are. In case of prefix rule matching for a single field (destination IP), the dependency graph is a tree [9]. However, in multi-field matching, commonly used in OpenFlow [12], the dependency can have a more general form of a DAG, even if the IP fields are matched by the longest prefix rule. Also, even for single-field matching, the wildcard rule can result in DAG dependencies [2].

For more background and details on such network architectures in general and on the technical setup of caching classification rules, we refer to prior works [2], [13], [9], [14].

B. Related Work

Due to their wide applications, algorithms for caching (often also called paging) were studied for decades, and here we overview the results from the perspective of competitive analysis. The seminal paper of Sleator and Tarjan [5] originated the concept of competitive analysis of online algorithms. In their paper, an upper bound of k-competitiveness was established for a family of marking algorithms that included commonly studied Least Recently Used and FIFO algorithms, and it was shown that no deterministic algorithm can be better than k-competitive. Randomization helps: Fiat et al. [6] showed that an algorithm RandomMark is $2H_k$-competitive and no randomized algorithm can be better than H_k-competitive. Two algorithms [15], [16] match this lower bound, hence competitiveness of online caching is fully understood in this model. Some existing work on caching uses similar terminology to dependencies, but the models differ substantially from ours. First, dependency-aware caching was studied from the practical perspective in the context of parallel processing systems [17], but the dependencies are required only at the fetch time, and can immediately be evicted afterward. Second, online caching was considered under a restriction called access graph [18], a request at time t can be followed only by a subset of requests at $t + 1$, consistently with a given access graph. In their work, the cache state is unrestricted, and in contrast, in our model we restrict feasible cache configurations rather than feasible input sequences.

Our paper is the most related to work of Bienkowski et al. [9], where they introduce *online tree caching*, a caching variant that can be viewed as online dependency-aware caching with the dependency graphs restricted to binary trees. In their model the requests can be *bypassed* [7], [8]: the request for item not present in the cache can be served from the slow memory, incurring a fixed cost. The algorithm presented in their paper attains the competitive ratio of $O(k \cdot h(T))$, where $h(T)$ is the height of the dependency tree.

In the context of packet classification in communication networks, an algorithm CacheFlow [2] was proposed to deal with dependencies among the rules. The algorithm splits the packet classification rules to minimize overlap, and uses the estimated past rule popularity statistics to determine the best cache configuration. Empirical evaluations of CacheFlow demonstrate the applicability of the approach in the context of packet classification. Other worth-mentioning attempts to improve caching forwarding rules tried to avoid involving the controller by using cooperation between switches [13] or optimizing rules storage with dynamic compression algorithms [19], [20] for more restricted scenarios.

C. Contributions

First, we consider the dependency-aware caching problem in the most natural setting *without bypassing*, where after requesting an item, it must be placed in the cache along with its dependencies. For this setting, we develop optimal deterministic and randomized algorithms. The optimal deterministic algorithm, Recursive LRU is k-competitive, and the randomized algorithm Bucketing is $2H_k$-competitive, where H_k is the k-th Harmonic number. Further, we characterize how the competitive ratio depends on the topology of the dependency graph. Our randomized algorithm is $2H_{\min\{k,\ell\}}$-competitive, where ℓ is the size of the maximum independent

set in the transitive closure of the dependency graph, k is the size of the cache, and H_k is the k-th Harmonic number.

We complement this result by showing that no randomized algorithm can be better than $H_{\min\{k,\ell\}}$-competitive. Hence, our algorithm is asymptotically optimal from the competitive standpoint. We highlight that the lower bound holds even for the simplest dependency structure of the tree, hence the algorithm is optimal for the simplest case of prefix rule matching (cf. Section I-A).

Next, we consider a related variant of dependency-aware caching where requests can be *bypassed* [7], [8], meaning that for a fixed cost an algorithm could avoid potentially costly fetch of the requested item and its dependencies. This setting was already studied in the literature in the context of caching packet classification rules [1]: Bienkowski et al. [9] proposed an $O(k \cdot h(T))$-competitive algorithm, where $h(T)$ is the height of the dependency tree (their algorithm restrict dependencies to trees). We significantly improve upon this result by developing a $(6\sqrt{k \cdot H_{\min\{k,\ell\}}})$-competitive algorithm, a ratio that is independent of the height of the dependency graph. We note that in contrast to this result, the competitive ratio of our algorithm can only improve when we account for properties of the dependency graph.

Finally, we perform an empirical case study comparing the TreeCaching algorithm of Bienkowski et al. [9] with our randomized algorithm, finding that it on average performs 2x better in terms of the average cost per request.

II. Preliminaries

A. Model

We introduce the *online dependency-aware caching* problem, defined as follows. Our task is to manage a two level memory hierarchy, consisting of a slow memory which stores the universe $\mathcal{U}$ of n items, and a fast memory which stores at most k items, where $k \geq 1$ is a parameter.

At the beginning we are given a set of dependencies among the items, given as an arbitrary directed acyclic graph (DAG) $G = (\mathcal{U}, E)$, which restricts the set of feasible caches. For each item x, the set of its *dependencies* are the items reachable from x in G, excluding the item x itself. At any time, an item can be present in the cache only if all its dependencies are in the cache. We assume that each item has at most $k - 1$ dependencies. If G has no edges, the problem is equivalent to the classic online caching problem [21].

In the online manner, we receive a sequence σ of requests to the items. If a requested item is not in the cache, we must *fetch* this item into the cache alongside with any of its dependency that may not be in the cache. As the size of the cache is limited, we may need to evict other items to fetch the requested item and its dependencies. The goal of the algorithm is to minimize the number of fetches.

B. Notation

Let x and y be two items of the universe $\mathcal{U}$. We say that y is a *descendant* of x (or x is an *ancestor* of y) if there exists a directed path from x to y in G. By $T(x)$ we denote the set of all descendants of x. In other words, $T(x)$ is the set that contains x and all its dependencies.

For the rest of the paper, we fix a total order τ among the items such that τ is consistent with an arbitrary reversed topological order of the items in G. For example, the numbers in the items of Figure 1 are a reversed topological order and the order $\leq$ on the item's associated numbers is a valid order τ between the items.

For any directed acyclic graph D, we define its maximum independent set as follows. First, we take the transitive closure D^T of D. Then, we take the symmetric closure D^S of D^T. We say that the maximum independent set of D is the maximum independent set of D^S.

III. Dependency-Aware Caching without Bypassing

In this section, we study the problem of online caching with dependencies in the randomized setting. We present a randomized algorithm Bucketing and prove that its expected competitive ratio is always below $2H_{\min\{k,\ell\}}$. In Appendix III-C we show an asymptotically tight lower bound which proves Bucketing is the best possible.

A. The Randomized Algorithm Bucketing

In this section we present our randomized algorithm called Bucketing. The design of the algorithm relies on the notion of *bucket* define hereafter.

Definition 1. *A bucket is a subset of items with the following properties:*

- *it is not empty*
- *all its items are cached*
- *evicting its item with maximum τ (called maximum item) leaves a feasible cache*

We say that a bucket is *frozen* if it used to be a bucket but does not match the definition anymore.

Having a bucket ensures that one of its items — the one with maximum τ — is ready for eviction. Hence, our algorithm Bucketing uses buckets as its only eviction interface; its main tasks then are to maintain a pool of buckets and choose a bucket when an eviction is needed. Here follows a description of the algorithm.

Bucketing operates in *phases*. The first phase starts with the first request σ_1. At the start of a phase, Bucketing generates a new pool of buckets. To do so, it iterates over each maximal cached item x (i.e. cached item without cached ancestor) and inserts $T(x)$ as a bucket in the pool (see an illustration in Figure 2). Bucketing now has a non-empty pool of buckets.

Whenever a request arrives, Bucketing starts by removing the requested item and its dependencies from all buckets. To satisfy the request, multiple items might be missing: in order to fetch them, Bucketing considers the missing items one by one in increasing τ. When it considers a missing item to fetch, the cache may however be full. In that case, Bucketing chooses one bucket uniformly at random in the pool and evicts its

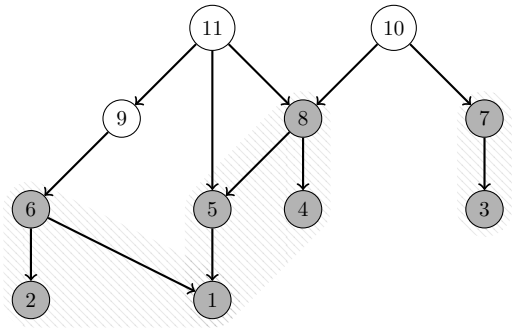


Fig. 2. Example of buckets formed at the beginning of a phase. We have three buckets, depicted by dashed sets, with maximum items (i.e. candidates for eviction) 6, 8 and 7. The item 1 is in two buckets.

maximum item out of the cache (and also remove the evicted item from all buckets). Bucketing then fetches the considered item.

At some point, it may happen that some subset of items in our pool is no longer a bucket. The reason may be that the subset has become empty, or that its maximum item cannot be evicted due to dependency relations. In that situation, we say that the bucket is *frozen*. Whatever the reason that caused it, if a bucket froze then Bucketing immediately removes that bucket from the pool. Finally, a new phase begins when the pool becomes empty.

Above we gave an almost complete presentation of Bucketing. There are however still edge cases that need to be properly addressed for correctness, for instance what to do if a new phase starts in the middle of serving a request. For a complete description we refer to the pseudo-code in algorithm 1.

If G has no edges, our problem is equivalent to classic caching. In that case, it is worth noting that our algorithm Bucketing is then equivalent to the well-known, optimal random-mark algorithm by Fiat et al [6]. In a sense, a bucket in our algorithm directly generalizes an unmarked nodes in the random-mark algorithm.

Algorithm 1: Randomized algorithm Bucketing

procedure *ResetBuckets()*:
  $\Gamma := \emptyset$
  foreach *cached item x with no cached ancestor* **do**
    $\Gamma := \Gamma \cup \{T(x)\}$
  return Γ
▷ Fetch w, evict an item beforehand if necessary
procedure *EvictAndFetch(w, Γ)*:
  if *w not in the cache* **then**
    if *the cache is full* **then**
      choose $B \in \Gamma$ uniformly at random, let $y \in B$ with maximum τ
      Remove y from all buckets
      Remove all frozen buckets from Γ
      Evict y from the cache
    Fetch w into the cache
  return Γ
▷ Main Algorithm
algorithm *Bucketing*:
  $\Gamma \leftarrow$ ResetBuckets()
  when *a new request v arrives* **do**
    for *w in $T(v)$ ordered in increasing τ* **do**
      Remove the items of $T(v)$ from all buckets
      Remove all frozen buckets from Γ
      if $\Gamma = \emptyset$ **then**
        $\Gamma \leftarrow$ ResetBuckets()
        Remove the items of $T(v)$ from all buckets
        Remove all frozen buckets from Γ
      $\Gamma \leftarrow$ EvictAndFetch(w, Γ)
    Serve the request to v

B. Proof for the competitiveness of Bucketing

In this section, we prove that Bucketing holds an expected competitive ratio of $H_{\min\{k,\ell\}}$ against the oblivious adversary.

1) Upper Bound for the Bucketing Algorithm: In this subsection, we derive an upper bound on the expected cost (number of fetches) of Bucketing during any phase. Fix any input sequence σ and directed acyclic graph G. In the next paragraph we make our analysis framework more precise.

In order to better synchronize our analysis with the actions of Bucketing, we will use a more fine-grained yet equivalent request sequence than σ. We call this sequence σ_τ and construct it as follows. Iterate over the requests of σ and replace each x by $T(x)$ sorted in increasing τ. We refer to the items of σ_τ as *pseudo-requests*. As Bucketing considers the items to fetch in the same order as in σ_τ, the following correspondence holds: to any instant when Bucketing considers a given item w corresponds a unique pseudo-request $w \in \sigma_\tau$. We can therefore without ambiguity refer to specific actions that Bucketing performed after receiving a given pseudo-request $w \in \sigma_\tau$, such as an eviction or a fetch.

We partition the sequence σ_τ into subsequences separated by bucket regeneration events (as defined above). We call those subsequences *phases* and will upper-bound the cost of Bucketing on each of them.

Let P be a phase, and let m be the number of buckets that Bucketing generated at the beginning of P. We further break P into subsequences called *fragments*. The fragments make up a partition of P so that the number buckets in the pool stays the same during each fragment. In other words, the fragments are separated by freeze events. If two or more buckets freeze at the same time (*i.e.*, due to the same pseudo-request), we order these freeze events arbitrarily. Hence, we can without ambiguity name each fragment after the number of buckets that are already frozen as it begins: we call the fragments $F_0, F_1 \dots F_{m-1}$.

Let $i \in [0, m-1]$ and $b \in [1, m]$, we define X_i^b as the random variable equal to the number of items that were evicted during F_i in the b-th bucket to freeze. $X_i = \sum_{b=1}^{m} X_i^b$ is the total number of evicted items during F_i. The following lemma

then holds.

Lemma 2. *$\forall i \in [0, m-1]$, $\forall b \in [i+1, m]$ it holds*

$$\mathbb{E}[X_i^b] = \frac{\mathbb{E}[X_i]}{m-i}$$

Proof. We first show that, during any fragment i, each non-frozen bucket experiences the same number of evictions in expectation. To this end, we label the successive pseudo-requests in fragment F_i with integer numbers $t = 1, 2, 3, \ldots$. We define the following random variables: for any t, let $\mathcal{E}(t)$ be 1 if the fragment F_i has a t-th pseudo-request and if that t-th pseudo-request leads to an eviction, and 0 otherwise. We also define random variables $X_i^b(t)$ which equals 1 if an eviction occurs in the b-th bucket when pseudo-request t arrives, and 0 otherwise. (Notice that $X_i^b(t)$ equals 0 if the fragment F_i does not have a t-th pseudo-request.)

Then it holds

$$\begin{aligned}\mathbb{E}[X_i^b(t)] = &\ \mathbb{E}[X_i^b(t)|\mathcal{E}(t)=1] \cdot \mathbb{P}[\mathcal{E}(t)=1] \\ &+\mathbb{E}[X_i^b(t)|\mathcal{E}(t)=0] \cdot \mathbb{P}[\mathcal{E}(t)=0] \\ = &\ \frac{1}{m-i} \cdot \mathbb{P}[\mathcal{E}(t)=1] + 0\end{aligned}$$

Hence, the expected value of $X_i^b(t)$ does not depend on the index b. Therefore, for all $b, b' \in [i+1, m]$, it holds that $\mathbb{E}[X_i^b(t)] = \mathbb{E}[X_i^{b'}(t)]$.

Notice that for all $b \in [i+1, m]$ it holds $X_i^b = \sum_t X_i^b(t)$. Hence, each non-frozen bucket indeed in expectation experiences the same number of evictions. More formally, for all $b, b' \in [i+1, m]$, we have $\mathbb{E}[X_i^b] = \mathbb{E}[X_i^{b'}]$.

Finally, we prove the claim of the lemma. We have that $X_i^b = 0$ for all $b \in [1, i]$ since the buckets it refers to are frozen when fragment F_i begins. Hence, it holds $\mathbb{E}[X_i] = \sum_{b=i+1}^{m} \mathbb{E}[X_i^b]$, and $\forall b \in [i+1, m]$ we have $\mathbb{E}[X_i^b] = \frac{\mathbb{E}[X_i]}{m-i}$. □

Let $x \in \mathcal{U}$ be an item that was fetched during phase P. We say that this fetch is *clean* if x was not in the cache at the beginning of P, otherwise it is *stale*. Let C be the number of clean fetches during P and S be the number of stale fetches during P. The total cost during phase P is then $C + S$.

We characterize the phase in more details. Let $i \in [0, m-1]$, C_i and S_i are respectively the number of clean and stale fetches during fragment F_i. Zooming more, let $b \in [1, m]$, S_i^b is the number of stale fetches during fragment i in the b-th bucket to freeze — we say that a stale fetch is *in* a given bucket if the fetched item was previously evicted because that bucket was randomly chosen to perform the eviction. Finally, $S^b = \sum_{i=0}^{m-1} S_i^b$ is the total number of stale fetches in the b-th bucket to freeze throughout the phase P.

Lemma 3. *Let $b \in [1, m]$, it holds that $\mathbb{E}[S^b] \leq \sum_{i=0}^{b-1} \frac{\mathbb{E}[C_i + S_i]}{m-i}$.*

Proof. For a stale fetch to occur, the fetched item in question must have been evicted previously in the phase. Hence, for a given bucket and across the whole phase, the number of stale fetches is no greater than the number of evictions; more formally, $\forall b \in [1, m]$ it holds

$$S^b \leq \sum_{i=0}^{m-1} X_i^b.$$

Past fragment F_{b-1}, bucket b becomes frozen and will experience no more evictions for the rest of the phase; it therefore holds that $\forall i \in [b, m-1], X_i^b = 0$. Using this remark and Lemma 2, we get

$$\mathbb{E}[S^b] \leq \sum_{i=0}^{b-1} \mathbb{E}[X_i^b] = \sum_{i=0}^{b-1} \frac{\mathbb{E}[X_i]}{m-i}. \tag{1}$$

Moreover, after any eviction directly occurs a fetch: during any period of time, the number of evictions therefore is no larger than the number of fetches. Taking fragments as periods of time and noticing that any fetch is either clean or stale, it therefore holds that $\forall i \in [0, m-1], X_i \leq C_i + S_i$. From this inequality we directly derive

$$\sum_{i=0}^{b-1} \frac{\mathbb{E}[X_i]}{m-i} \leq \sum_{i=0}^{b-1} \frac{\mathbb{E}[C_i + S_i]}{m-i}, \tag{2}$$

which proves the claim combined with (1). □

Lemma 4. *For any $b \in [1, m]$ it holds that*

$$\mathbb{E}[S^b] \leq \frac{\mathbb{E}[C]}{m-b+1}$$

Proof. By induction on b, we prove the following inequality:

$$\sum_{i=0}^{b-1} \frac{\mathbb{E}[C_i + S_i]}{m-i} \leq \frac{1}{m-b+1} \left(\sum_{i=0}^{b-1} \mathbb{E}[C_i] - \sum_{j=1}^{b-1} \sum_{i=b}^{m-1} \mathbb{E}[S_i^j] \right) \tag{3}$$

which directly proves the claim along with using Lemma 3. Inequality (3) clearly holds for $b = 1$ since $S_0 = 0$. Let $b \in [1, m]$, we suppose that the claim holds for b. Then,

$$
\begin{aligned}
&\sum_{i=0}^{b} \frac{\mathbb{E}[C_i + S_i]}{m-i} = \sum_{i=0}^{b-1} \frac{\mathbb{E}[C_i + S_i]}{m-i} + \frac{\mathbb{E}[C_b] + \mathbb{E}[S_b]}{m-b} \\
= &\sum_{i=0}^{b-1} \frac{\mathbb{E}[C_i + S_i]}{m-i} \\
&+ \frac{\mathbb{E}[C_b] + \mathbb{E}[S^b] + \sum_{j=1}^{b-1} \mathbb{E}[S_b^j] - \sum_{i=b+1}^{m-1} \mathbb{E}[S_i^b]}{m-b} \\
\leq &\sum_{i=0}^{b-1} \frac{\mathbb{E}[C_i + S_i]}{m-i} \\
&+ \frac{\mathbb{E}[C_b] + \sum_{i=0}^{b-1} \frac{\mathbb{E}[C_i+S_i]}{m-i} + \sum_{j=1}^{b-1} \mathbb{E}[S_b^j] - \sum_{i=b+1}^{m-1} \mathbb{E}[S_i^b]}{m-b} \\
= &\frac{m-b+1}{m-b} \cdot \left(\sum_{i=0}^{b-1} \frac{\mathbb{E}[C_i + S_i]}{m-i} \right) \\
&+ \frac{\mathbb{E}[C_b] + \sum_{j=1}^{b-1} \mathbb{E}[S_b^j] - \sum_{i=b+1}^{m-1} \mathbb{E}[S_i^b]}{m-b} \\
\leq &\frac{\sum_{i=0}^{b-1} \mathbb{E}[C_i] - \sum_{j=1}^{b-1} \sum_{i=b}^{m-1} \mathbb{E}[S_i^j]}{m-b} \\
&+ \frac{\mathbb{E}[C_b] + \sum_{j=1}^{b-1} \mathbb{E}[S_b^j] - \sum_{i=b+1}^{m-1} \mathbb{E}[S_i^b]}{m-b} \\
= &\frac{1}{m-b} \cdot \left(\sum_{i=0}^{b} \mathbb{E}[C_i] - \sum_{j=1}^{b} \sum_{i=b+1}^{m-1} \mathbb{E}[S_i^j] \right)
\end{aligned}
$$

We used the identity $S_b = S^b + \sum_{j=1}^{b-1} S_b^j - \sum_{i=b+1}^{m-1} S_i^b$ in line 2, Lemma 2 in line 3 and the induction hypothesis in line 5. The induction holds hence the claim is true for all $b \in [1, m]$. □

Theorem 5. *In each phase,* Bucketing *fetches in expectation no more than $H_{\min\{k,\ell\}} \cdot C$ items, where C is the number of clean fetches in the phase, ℓ is the size of the maximum independent set in the transitive closure of the dependency graph and k is the size of the cache.*

Proof. We first examine the number of stale fetches in the last bucket to freeze. Until it freezes, they are no fetches in it (otherwise it would have frozen earlier); but when it finally freezes, the phase is over and no more fetches will occur. Hence, there are no stale fetches in the last bucket to freeze, *i.e.*, $S^m = 0$. Summing the inequalities from Lemma 4 for all buckets $b \in [1, m-1]$ and taking into account the last bucket's specificity gives us the following

$$\mathbb{E}[S] \leq \left(\sum_{b=1}^{m-1} \frac{1}{m-b+1} \right) \cdot \mathbb{E}[C].$$

Adding the expected total number of clean fetches on both members, noticing that C does not depend on the random choices of Bucketing (i.e. that $\mathbb{E}[C] = C$) and that the initial number of buckets m is smaller than both k and ℓ ends the proof. □

2) A Lower Bound for any Algorithm in a Phase: Let OPT be an optimal algorithm for the input sequence σ. Similarly like Fiat et al. [6], we can prove that OPT has an (amortized) cost at least C in every phase. Only one technicality prevents us from directly applying their proof; it arises when the phase ends while Bucketing have not yet fetched all the dependencies of the next request. Then, there is no guarantee that OPT must have paid to fetch those items during the current phase. Intuitively, the lower-bound however still applies since those items will be paid by OPT during the next phase anyway.

One method to present a rigorous proof is to compare with a specific optimal offline algorithm OPT_τ that performs the same evictions and fetches as OPT for each request, but in a specific order: for each request, all the evictions happen first and then the fetches happen following the τ order. OPT_τ pays the same cost as OPT and it is feasible. More importantly, OPT_τ is synchronized with Bucketing which allows to directly apply the proof from Fiat et al.

3) Bounding the Competitive Ratio: The following theorem ends the analysis of our randomized algorithm Bucketing.

Theorem 6. *The algorithm* Bucketing *is $2H_{\min\{k,\ell\}}$-competitive against the oblivious adversary, where ℓ is the size of the maximum independent set in the transitive closure of the dependency graph and k is the size of the cache.*

Proof. Directly from the above upper and lower-bounds. □

This result is tight with the lower bound that we present in the next subsection.

C. A Lower Bound for Dependency-Aware Caching without Bypassing

In this section we give a lower bound for the competitive ratio of any randomized online algorithm for caching with dependencies. The competitive ratio depends on the topology of the graph, namely the maximum independent set (for the definition of the maximum independent set in directed graphs we refer to Section II-B). This lower bound asymptotically matches the upper bound given in Section 1.

Theorem 7. *No randomized online algorithm can achieve a competitive ratio better than $H_{\min\{k,\ell\}}$ against the oblivious adversary, where k is the size of the cache and ℓ is the size of the maximum independent set of the dependency graph.*

We defer the proof of this theorem to the full version of the paper due to page restrictions, and we sketch it below. We consider a dependency graph consisting of $\ell - 1$ isolated items and a single chain of length $k - \ell + 2$. Then, we construct a probability distribution for requesting the maximum nodes in the graph, similarly to the lower bound of H_k for caching by

Motwani and Raghavan [22]. To bound the competitive ratio, we consider the k-phase partition of this input sequence, argue that in each phase any deterministic algorithm pays $H_{\min\{k,\ell\}}$, whereas an optimal algorithm pays 1. We derive the bound for randomized algorithms with the Yao's principle [23].

D. Deterministic Algorithm Recursive LRU

To finish the section on the model variant without bypassing, we present a simple optimal deterministic algorithm. This completes the picture for this model: both our randomized and deterministic results are tight.

We define a natural deterministic algorithm for online dependency-aware caching, recursive LRU, which invokes the classic LRU algorithm for all dependencies of the requested item. Precisely, we split the algorithm's logic into two loops. The first loop updates the timestamps top-to-bottom to assure that the evictions will take place in the correct order. The second loop fetches dependencies bottom-to-top, to assure that at each intermediate cache state all items have their dependencies in the cache, consistently with the model definition.

We claim that this algorithm is k-competitive. The lower bounds for the classic online caching imply lower bounds for the more general dependency-aware caching problem, hence the lower bound of k given by Sleator and Tarjan [5] implies that DET is optimal.

Theorem 8. *DET is k-competitive.*

We defer the proof of this theorem to the full version of the paper due to page restrictions.

We note this simple design of recursively applying a known algorithm for the classic variant works for the deterministic case only, but does not lead to optimal competitiveness for the randomized algorithm.

IV. Dependency-Aware Caching with Bypassing

In this section, we consider a model of dependency-aware caching with *bypassing*, a similar setting to the classic caching with bypassing [7], [8]. When a request arrives, an algorithm may choose to either (1) serve it from the cache by fetching the requested item and its missing dependencies, or (2) to bypass it for a cost of 1. The first option, serving a request from the cache, may be more costly if the number of the dependencies are large, however the subsequent requests to the same item are free (until the cache changes). The second option, bypassing a request, always cost 1 per bypassed request, but does not require to fetch the dependencies of the requested item.

Introducing bypassing unravels new challenges to design a competitive online algorithm. Whether it is more beneficial to fetch or to bypass a request depends on the future requests. Hence, bypassing brings uncertainty to the online algorithm while it clearly benefits an offline algorithm.

In this section, we show how to use the randomized eviction procedure of the last section to take advantage of bypassing capabilities while providing strong worst case guaranties. We present the algorithm BucketingBypass and prove it attains a competitive ratio of $6\sqrt{k \cdot H_{\min\{k,\ell\}}}$ against the oblivious adversary, where k is the size of the cache, ℓ is the size of the maximum independent set in the transitive closure of the dependency graph and H_k is the k-th Harmonic number.

A. The Randomized Algorithm Bucketing with Bypassing

1) Presentation of the algorithm BucketingBypass: We present our randomized algorithm BucketingBypass that deals with the case when bypassing is allowed. Just like Bucketing, BucketingBypass operates in phases. The first phase starts with the first request σ_1. At the start of a phase, BucketingBypass initializes a pool of buckets exactly like Bucketing.

Whenever a request arrives, say to an item v, we distinguish between multiple cases. If v is already in the cache, BucketingBypass does the same as the algorithm Bucketing without bypassing: it removes $T(v)$ from all buckets and serves the request. Otherwise, in case v is not in the cache, BucketingBypass selects a item w successor of v positioned just above the cache frontier; it chooses the one item with minimum τ to avoid ambiguity. BucketingBypass then lists the items that are both successor of w and in a bucket — let S be the set of those items. If S contains strictly more than $\sqrt{k/H_{\min\{k,\ell\}}}$ items then BucketingBypass removes the $\sqrt{k/H_{\min\{k,\ell\}}}$ ones with minimum τ from the buckets. It bypasses v and waits for the next request. Otherwise, if S contains fewer than $\sqrt{k/H_{\min\{k,\ell\}}}$ items then BucketingBypass removes those items from the buckets, applies the eviction procedure of Bucketing if the cache is full and finally fetches w into the cache. If $w = v$ then BucketingBypass serves the request to v, else it bypasses it. The algorithm uses the procedures ResetBuckets and EvictAndFetch defined in Algorithm 1. For more details, the pseudocode of BucketingBypass can be found in Algorithm 2.

2) Upper-bound on the cost of BucketingBypass: In the rest of this section, we bound the competitive ratio of BucketingBypass. Like in the previous analysis without bypassing allowed, we define *phases* as sequences of consecutive requests between two calls to the function ResetBuckets().

Lemma 9. *For each phase, BucketingBypass pays at most $2H_{\min\{k,\ell\}} \cdot C + \sqrt{k \cdot H_{\min\{k,\ell\}}}$ in expectation, where C is the number of clean requests during the considered phase.*

Proof. Let P be a phase. We partition the requests of P into two subsequences P_f and P_b depending on how BucketingBypass handles them. A request goes in P_b if the algorithm handles it with lines 11 to 13, it goes in P_f otherwise. We now prove the two following inequalities which directly imply the claim:

(1) BucketingBypass pays at most $\sqrt{k \cdot H_{\min\{k,\ell\}}}$ for the requests in P_b.
(2) BucketingBypass pays at most $2H_{\min\{k,\ell\}} \cdot \mathbb{E}[C]$ for the requests in P_f

We first prove the subclaim (1). At the start of the phase the buckets together contain k items, this number decreases until it hits zero at the end of the phase. Each time a request of P_f is issued, BucketingBypass removes $\sqrt{k/H_{\min\{k.l\}}}$ items from

Algorithm 2: BucketingBypass algorithm with bypassing

```
1  algorithm BucketingBypass:
2      Γ ← ResetBuckets()
3      when a new request v arrives do
4          if v is in the cache then
5              Remove the items of T(v) from all buckets
6              remove all frozen buckets from Γ
7              serve v
8          else
9              Let w be the uncached item of T(v) with
                minimum τ
10             Let S be the set of items that are
                simultaneously in T(w) and in some
                bucket
11             if |S| > sqrt(k/H_{min{k,ℓ}}) then
12                 Remove the sqrt(k/H_{min{k,ℓ}}) items of S
                    with minimum τ from all buckets in Γ
13                 Remove all frozen buckets from Γ
14                 bypass v
15             else
16                 Remove the items of T(w) from all
                    buckets
17                 Remove all frozen buckets from Γ
18                 if Γ = ∅ then
19                     Γ ← ResetBuckets()
20                     Remove the items of T(w) from all
                        buckets
21                     Remove all frozen buckets from Γ
22                 Γ ← EvictAndFetch(w, Γ)
23                 If w = v then serve request v else
                    bypass it
```

the buckets. Hence, P_b contains at most $k/\sqrt{k/H_{\min\{k.l\}}} = \sqrt{k \cdot H_{\min\{k.l\}}}$ requests. Since BucketingBypass pays 1 for each request in P_b, it pays at most $\sqrt{k \cdot H_{\min\{k.l\}}}$.

In order to prove subclaim (2), we construct an alternative request sequence to P_f and run Bucketing on it. We show that Bucketing does the same changes of internal variables and pays at least half the cost of BucketingBypass on P_f dealing with that alternative request sequence.
Let v be a request of P_f. If v is in the cache when requested, then BucketingBypass behaves like Bucketing and pays 0. Otherwise, if v is not in the cache then everything behaves as if w was the requested item in a framework without the possibility to bypass and fetching costs where doubled. Finally, the interferences of the requests in P_b (that only shrink buckets) can be modeled by inserting in P_f fake requests of cached items. We then apply Theorem 5 and the subclaim follows. □

B. A Lower Bound for Any Offline Algorithm and Bounding the Ratio

Let OPT be an optimal offline algorithm for the request sequence σ. Based on OPT, we define another offline algorithm OPT_τ as follows. Let σ_i be a request. If OPT bypasses σ_i then OPT_τ bypasses it. Otherwise, if OPT does not bypass σ_i, OPT_τ fetches and evicts the same items as OPT but in a specific order. First, OPT_τ performs all the evictions that OPT does. Then, OPT_τ performs all the fetches that OPT does sorted in increasing τ. OPT_τ maintains a feasible cache and pays the same cost as OPT, OPT_τ is an optimal offline algorithm.

Similarly to the previous section, we compare the cost of BucketingBypass to the cost of OPT_τ. The following claim holds.

Lemma 10. *Let P be a phase, C is the number of clean fetches made by BucketingBypass during P. Let d_I and d_F be the number of items in OPT_τ's cache not in BucketingBypass's cache at the beginning and at the end of the phase. It holds:*

$$\mathsf{OPT}_\tau(P) \geq \frac{1}{2} \cdot \sqrt{\frac{H_{\min\{k,\ell\}}}{k}} \cdot (C - d_I + d_F).$$

We defer the proof of this claim to the full version of the paper due to page restrictions.

The derived amortized lower bound on OPT_τ (P) is proportional to the number of clean requests C. However, the upper bound on BucketingBypass (P) in Theorem 9 features an additive term which does not depend on C. As a last step before the final result, we give a second lower bound on OPT_τ (P) in case C is zero.

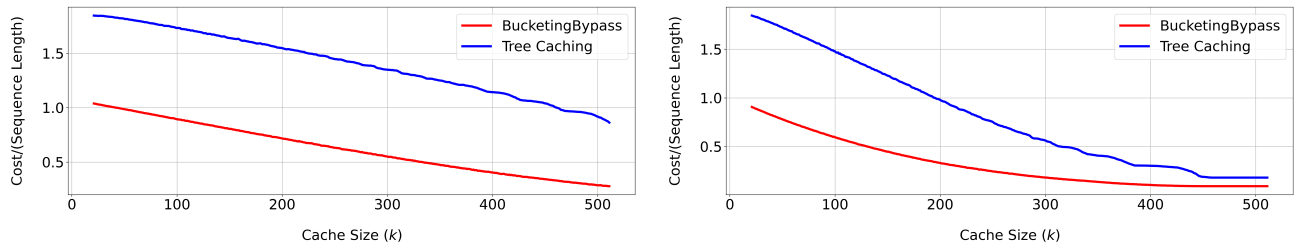
Lemma 11. *Let P be a phase. Let Φ_I and Φ_F be 1 if BucketingBypass's and OPT_τ's caches are not the same, respectively at the beginning and at the end of the phase. It holds:*

$$\mathsf{OPT}_\tau(P) \geq \frac{1}{2} \cdot (\mathbb{1}_{\mathsf{BucketingBypass}(P)>0} - \Phi_I + \Phi_F).$$

We defer the proof of this claim to the full version of the paper due to page restrictions.

Theorem 12. *The algorithm BucketingBypass is $6\sqrt{k \cdot H_{\min\{k,\ell\}}}$-competitive against the oblivious adversary, where ℓ is the size of the maximum independent set in the transitive closure of the dependency graph and k is the size of the cache.*

Proof. If BucketingBypass pays 0 in a phase then the claimed competitive ratio holds. Otherwise, we derive the claimed competitive ratio using the two amortized costs of



(a) Zipf distribution of requests among nodes

(b) Exponential distribution of requests among nodes

Fig. 3. Comparison of the BucketingBypass and TreeCaching algorithms in terms of cost per request for various cache sizes k and a binary tree with height $h(T) = 10$ and 1023 nodes. The left subfigure plot uses Zipf distribution parameterized by $a = 4$ and the right subfigure one uses geometric distribution parameterized by $p = \frac{10}{2^{10}}$.

OPT_τ found in Lemmas 10 and 11. It finally holds:

$$\frac{\mathsf{BucketingBypass}(P)}{\mathsf{OPT}_\tau(P)} \leq \frac{2H_m \cdot C + \sqrt{k \cdot H_{\min\{k,\ell\}}}}{1/2 \cdot \max\{\sqrt{\frac{H_{\min\{k,\ell\}}}{k}} \cdot C, \mathbb{1}_{\mathsf{BucketingBypass}(P)>0}\}} \leq 6\sqrt{k \cdot H_{\min\{k,\ell\}}} \quad \square$$

V. Evaluations

While the main objective of this paper is to develop a rigorously analyzed online algorithm with formal bounds on its performance, we additionally perform preliminary empirical evaluations of the performance of our BucketingBypass algorithm. The code of the evaluations is publicly available [24].

As a baseline, we compare with the algorithm TreeCaching of Bienkowski et al. [9]. Out of the two possible baselines, CacheFlow [2] and TreeCaching [9], the latter is closer in spirit to our solution, as both our system and TreeCaching optimize the same objective. Notably, CacheFlow design does not account for the cost of changing the cache, but rather it may periodically exchange entire cache, whereas striking the perfect balance between the cost and the benefits of cache exchanges is a fundamental design principle of our algorithm.

Recall that our algorithm attains the competitive ratio of $6\sqrt{k \cdot H_k}$, whereas TreeCaching attains the competitive ratio of $O(h(T) \cdot k)$, where $h(T)$ is the height of the tree. The analytical upper bounds favor our algorithm, with an improvement in terms of both parameters $h(T)$ and k, but the bounds concern the worst case, and in this section we compare these algorithms empirically.

To evaluate the algorithms, we use the following methodology. We construct a balanced binary tree with a height $h(T)$. To generate the requests, we conduct two experiments with different request distributions. The first experiment uses a Zipf distribution, a common distribution applied in this context [11], and the second experiment uses an exponential distribution, modelling highly skewed request pattern. We apply each distribution to the item heights: the height of each requested item is independently drawn from the distribution, resulting in higher probabilities for items at lower levels of the tree and lower probabilities for items at higher levels. Then, each for each level, an item from that level is chosen uniformly at random. This method may generate requests to items that can never fit in the cache, as they have more descendants than k. We prune these requests from our request sequence using rejection sampling.

For the first experiment, we use the Zipf distribution. The probability of an item with depth $1 \leq i \leq h(T)$ being requested, where $i = 1$ corresponds to the top item of the tree, is given by $Pr(i) = \frac{1}{2^{i-1}} \frac{(h(T)-i+1)^{-a}}{\zeta(a)}$, where ζ represents the Riemann Zeta function, and we use the parameter value $a = 4$. For the second experiment, we use the geometric distribution. Requesting an item with index $1 \leq i \leq 2^{h(T)} - 1$ has the following probability $Pr(i) = (1-p)^{i-1}p$, where we use $p = \frac{10}{2^{10}}$.

We generate a random sequence of requests with a fixed length 5000 for both distributions. Then, we calculate the total cost of the BucketingBypass and TreeCaching algorithms on such sequence, for various cache sizes k. We repeat each sequence 10 times and take an average cost, and then we determine the average cost per request for both algorithms.

In Figure 3, we present the results of our evaluations of the mean cost per request between the BucketingBypass and TreeCaching algorithms. The BucketingBypass algorithm consistently induces significantly lower costs compared to the TreeCaching algorithm across all inspected cache sizes k. Our algorithm incurs on average 2x lower cost per requests than TreeCaching.

VI. Conclusions and Future Directions

This paper proposed several competitive caching algorithms that are aware of dependencies among items. We leave interesting avenues open for future studies. Most notably, it will be interesting to study dependency aspects in more general variants of caching, such as weighted caching [25], [26], [27] or file caching [28], [29].

The authors have provided public access to their code and/or data at https://github.com/foo/dependency-aware-caching.

Acknowledgements. This work was supported by the Austrian Science Fund (FWF) project I 5025-N (DELTA). We thank the anonymous INFOCOM reviewers for their valuable feedbacks on our work.

References

[1] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Netw.*, vol. 15, no. 2, pp. 24–32, 2001.

[2] N. P. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: Dependency-aware rule-caching for software-defined networks," in *Proceedings of the Symposium on SDN Research, SOSR 2016*, B. Godfrey and M. Casado, Eds. ACM, 2016, p. 6.

[3] A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[4] A. Fiat and G. J. Woeginger, Eds., *Online Algorithms, The State of the Art*. Springer, 1998.

[5] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Communications of the ACM*, vol. 28, no. 2, pp. 202–208, 1985.

[6] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young, "Competitive paging algorithms," *J. Algorithms*, vol. 12, no. 4, pp. 685–699, 1991.

[7] L. Epstein, C. Imreh, A. Levin, and J. Nagy-György, "Online file caching with rejection penalties," *Algorithmica*, vol. 71, no. 2, pp. 279–306, 2015.

[8] S. Irani, "Page replacement with multi-size pages and applications to web caching," *Algorithmica*, vol. 33, no. 3, pp. 384–409, 2002.

[9] M. Bienkowski, J. Marcinkowski, M. Pacut, S. Schmid, and A. Spyra, "Online tree caching," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2017, Washington DC, USA, July 24-26, 2017*. ACM, 2017, pp. 329–338.

[10] L. Cittadini, W. Mühlbauer, S. Uhlig, R. Bush, P. François, and O. Maennel, "Evolution of internet address space deaggregation: Myths and reality," *IEEE J. Sel. Areas Commun.*, vol. 28, no. 8, pp. 1238–1249, 2010.

[11] N. Sarrar, S. Uhlig, A. Feldmann, R. Sherwood, and X. Huang, "Leveraging zipf's law for traffic offloading," *Comput. Commun. Rev.*, vol. 42, no. 1, pp. 16–22, 2012.

[12] N. McKeown, T. E. Anderson, H. Balakrishnan, G. M. Parulkar, L. L. Peterson, J. Rexford, S. Shenker, and J. S. Turner, "Openflow: enabling innovation in campus networks," *Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[13] O. Rottenstreich, A. Kulik, A. Joshi, J. Rexford, G. Rétvári, and D. S. Menasché, "Data plane cooperative caching with dependencies," *IEEE Trans. Netw. Serv. Manag.*, vol. 19, no. 3, pp. 2092–2106, 2022.

[14] V. Addanki, M. Pacut, A. Pourdamghani, G. Rétvári, S. Schmid, and J. Vanerio, "Self-adjusting partially ordered lists," *INFOCOM*, 2023.

[15] L. A. McGeoch and D. D. Sleator, "A strongly competitive randomized paging algorithm," *Algorithmica*, vol. 6, no. 6, pp. 816–825, 1991.

[16] D. Achlioptas, M. Chrobak, and J. Noga, "Competitive analysis of randomized paging algorithms," *Theor. Comput. Sci.*, vol. 234, no. 1-2, pp. 203–218, 2000.

[17] Y. Yu, C. Zhang, W. Wang, J. Zhang, and K. B. Letaief, "Towards dependency-aware cache management for data analytics applications," *IEEE Trans. Cloud Comput.*, vol. 10, no. 1, pp. 706–723, 2022.

[18] A. Borodin, S. Irani, P. Raghavan, and B. Schieber, "Competitive paging with locality of reference," *J. Comput. Syst. Sci.*, vol. 50, no. 2, pp. 244–258, 1995.

[19] M. Bienkowski, N. Sarrar, S. Schmid, and S. Uhlig, "Competitive fib aggregation without update churn," in *2014 IEEE 34th International Conference on Distributed Computing Systems*, 2014, pp. 607–616.

[20] M. Bienkowski and S. Schmid, "Competitive fib aggregation for independent prefixes: Online ski rental on the trie," in *Structural Information and Communication Complexity*, T. Moscibroda and A. A. Rescigno, Eds. Cham: Springer International Publishing, 2013, pp. 92–103.

[21] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Commun. ACM*, vol. 28, no. 2, pp. 202–208, 1985.

[22] R. Motwani and P. Raghavan, "Randomized algorithms," in *Algorithms and Theory of Computation Handbook*, ser. Chapman & Hall/CRC Applied Algorithms and Data Structures series, M. J. Atallah, Ed. CRC Press, 1999.

[23] A. C. Yao, "Probabilistic computations: Toward a unified measure of complexity," in *18th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 1977.

[24] [Online]. Available: https://github.com/foo/dependency-aware-caching

[25] E. Cohen and H. Kaplan, "Lp-based analysis of greedy-dual-size," in *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1999, pp. 879–880.

[26] N. E. Young, "The k-server dual and loose competitiveness for paging," *Algorithmica*, vol. 11, no. 6, pp. 525–541, 1994.

[27] N. Bansal, N. Buchbinder, and J. Naor, "A primal-dual randomized algorithm for weighted paging," *J. ACM*, vol. 59, no. 4, pp. 19:1–19:24, 2012.

[28] N. E. Young, "On-line file caching," *Algorithmica*, vol. 33, no. 3, pp. 371–383, 2002.

[29] A. Adamaszek, A. Czumaj, M. Englert, and H. Räcke, "An $O(\log k)$-competitive algorithm for generalized caching," *ACM Trans. Algorithms*, vol. 15, no. 1, pp. 6:1–6:18, 2019.