Online File Caching in Latency-Sensitive Systems with Delayed Hits and Bypassing

Chi Zhang*, Haisheng Tan*, Guopeng Li*, Zhenhua Han†, Shaofeng H.-C. Jiang‡, and Xiang-Yang Li*
*CAS Key Lab of Wireless-Optical Communications, University of Science and Technology of China, China
†Microsoft Research Asia, China ‡Peking University, Beijing, China

IEEE INFOCOM 2022 - IEEE Conference on Computer Communications | 978-1-6654-5822-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/INFOCOM48880.2022.9796969

***Abstract*—In latency-sensitive file caching systems such as Content Delivery Networks (CDNs) and Mobile Edge Computing (MEC), the latency of fetching a missing file to the local cache can be significant. Recent studies have revealed that successive requests of the same missing file before the fetching completes could still suffer latency (so-called delayed hits).**

Motivated by the practical scenarios, we study the online general file caching problem with delayed hits and bypassing, i.e., a request may be bypassed and processed directly at the remote data center. The objective is to minimize the total request latency. We show a general reduction that turns a traditional file caching algorithm to one that can handle delayed hits. We give an $O(Z^{3/2}\log K)$-competitive algorithm called `CaLa` with this reduction, where Z is the maximum fetching latency of any file and K is the cache size, and we show a nearly-tight lower bound $\Omega(Z\log K)$ for our ratio. Extensive simulations based on the production data trace from Google and the Yahoo benchmark illustrate that `CaLa` can reduce the latency by up to 9.42% compared with the state-of-the-art scheme dealing with delayed hits without bypassing, and this improvement increases to 32.01% if bypassing is allowed.

I. Introduction

Online file caching is a fundamental problem widely studied in computer and networking systems. The conventional objective of file caching is to minimize the cache misses or the total cost of file retrievals. In general, an exquisite online file caching algorithm should provide a lower average file access latency, resulting in a better user experience. When all files have uniform size and uniform fetch cost (*i.e.*, the paging problem), intuitive algorithms such as *Least Recently Used* (LRU) and *First In First Out* (FIFO) can achieve a competitive ratio of $O(K)$ with respect to minimizing the number of misses, where K is the cache size [1].

However, in practical applications such as Content Delivery Networks (CDNs) [2] and Mobile Edge Computing (MEC) [3], due to the long physical distance, the latency for fetching missing files from the remote data center can be more than $100ms$ [4], [5], whereas the average inter-time for two consecutive file requests could be as low as $1\mu s$ [6], *e.g.*, 1M file requests per second. An interesting case appears. During the period when a missed file is retrieved from the remote data center, the subsequent requests for this file can not be served immediately, and thus should not be simply treated as a hit. This case is also different from a simple miss as the requests can be served as a hit after the file is fetched to local servers. Hence we called this case *a delayed hit* [6]. Moreover, traditional cache models [1], [7], [8] assume all the missing files have to be fetched and stored in the local cache before being accessed, while in the scenario of cloud related applications, file requests can be sent to and served directly at the remote cloud, which we call *bypassing*. Fig. 1 illustrates online file caching in a cloud-based system with the file misses, hits, delayed hits and bypassing, where there are a local cache server and multiple remote data centers. The first request for X arrives at T_1. Since X is not stored in the cache, it is a miss and triggers fetching X from Data Center 1, and X will not be ready in the local cache until time T_3 due to the fetching latency. Then, another request for X arrives at T_2 ($T_1 < T_2 < T_3$), which will be buffered and served at T_3, which is a delayed hit. The third request for X arrives at T_3 is a hit. For the request for Y arrives at T_4, we choose to bypass this request directly to avoid space allocation in the cache.

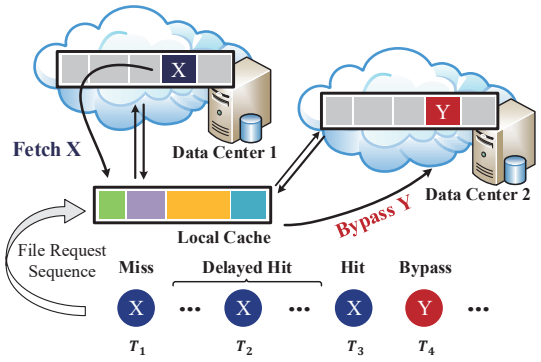


Fig. 1. An example of an online file caching system, where a file request may be served at the local cache or bypassed directly to the remote data center.

So far, few results have been published on online file caching with delayed hits although it has been noted in the literature [9]. A representative work is the online paging problem[1] studied in [6], where the authors novelly revealed the importance of delayed hits in systems with large throughput, and proposed MAD, an online solution incorporating files' aggregate delay into existing practical caching algorithms such as LRU [10], ARC [11] and LHD [12]. However, the file sizes in cloud-based applications vary dramatically, *i.e.*, up to

[1]Throughout this paper, *paging* represents the special case of the caching problem where the size and fetching cost are both uniform for each file. *Weighted paging* means the uniform file size but non-uniform fetching costs. *File caching* means non-uniform file sizes and the fetching cost can be uniform or non-uniform. When the size and fetching cost are both non-uniform, we call it *the general file caching*.

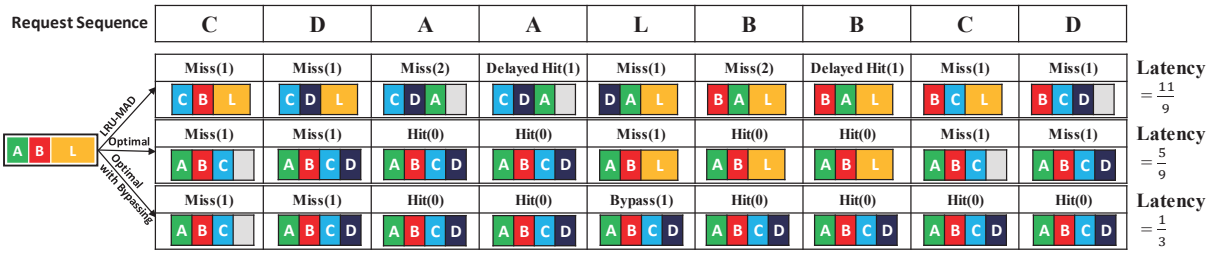


Fig. 2. An example in general file caching with delayed hits and bypassing, where files have heterogeneous size and latency. The size of cache $K = 4$, files' size $s_A = s_B = s_C = s_D = 1$, $s_L = 2$, fetching latency $z_A = z_B = 2$, $z_C = z_D = z_L = 1$. The average latency of LRU-MAD, Optimal and Optimal with bypassing are $11/9$, $5/9$ and $1/3$, respectively.

thousands of times in Google product traces [13]. Hence, the fetching cost of various files could be quite different, and the general file caching problem should be investigated. Moreover, in cloud-based systems, bypassing should also be taken into account. The following motivating example illustrates that the existing schemes fail in tackling online general file caching with delayed hits and bypassing.

Motivating Example. As shown in Fig. 2, there are 5 different files A, B, C, D and L will be requested, where files' size $s_A = s_B = s_C = s_D = 1$, $s_L = 2$, and fetching latency $z_A = z_B = 2$, $z_C = z_D = z_L = 1$. The latency to bypass a request is the same as fetching this file. The cache size is 4. Initially, there are A, B and L in the cache. The sequence of file requests that will arrive is $C, D, A, A, L, B, B, C, D$. When the first request for C arrives, one of A, B or L have to be evicted to make room for C. According to the guidelines of least recently used, LRU-MAD will evict A and put C into cache. For the same reason, B is replaced by D. Then two consecutive requests for A will cause a miss and a delayed hit respectively. After L is requested, C is evicted from the cache and L is stored in the cache. The following two consecutive requests for B will also cause a miss and a delayed hit respectively. Finally, the last request for C and D will also be missed and the average latency of LRU-MAD is therefore $11/9$. By contrast, the optimal solution will evict the larger file L when the first request for C arrives and the subsequent requests for A will be two hits. When L is requested, C and D will be evicted since they have lower fetching latency. The average latency of optimal is $5/9$. If bypassing is allowed, the optimal will bypass all the requests for L since L is with a larger size and lower fetching latency. The average latency of optimal with bypassing is $1/3$.

In this paper, we study the online general file caching problem with bypassing. We proposed a novel framework to effectively transform *any* existing algorithm in classic file caching models, *e.g.*, without delayed hits considered, to a solution for our model with delayed hits. The main idea is to find an estimated weight for each file, which can express the total cost caused by this file's miss, and run the classic algorithm with the estimated weights of all files. Our contributions are summarized as follows.

- We investigate a practical online general file caching problem with bypassing to minimize the total latency of file requests, where the file size and fetching latency are both non-uniform. We first prove the lower bound $\Omega(ZK)$ and $\Omega(Z \log K)$ of this problem in deterministic and randomized algorithms, where Z is the maximum of the file's fetching latency and K is the cache size (in Sec. II).
- We derive a deterministic online algorithm, called `CaLa`, with the competitive ratio of $O(Z^{3/2}K)$. Furthermore, the randomized version of `CaLa` is $O(Z^{3/2} \log K)$-competitive. To the best of knowledge, `CaLa` is the first online algorithm with competitive ratios for the online general file caching problem with delayed hits and bypassing (in Sec. III).
- We conduct extensive simulations on Google's production trace and Yahoo! Cloud Serving Benchmark. The results show that compared with LRU-MAD, the state-of-the-art algorithm that deals with delayed hits, `CaLa` can reduce the latency by up to 9.42% without bypassing, which will be increased to 32.01% if bypassing is allowed (in Sec. IV).

II. Problem Formulation

Cache System. Motivated by applications as CDNs and MEC, we consider the online general file caching model, a local cache server and remote data centers. Let K be the cache size, and $\mathcal{F} = \{f_1, f_2, \ldots, f_N\}$ be the set of all kinds of files, where each file $f_i \in \mathcal{F}$ $(1 \le i \le N)$ has size s_{f_i} and fetching latency z_{f_i}. We also use s_i to represent s_{f_i} and use z_i to represent z_{f_i} for concision when there is no ambiguity. Set $Z = \max_i z_i$. Without loss of generality, we assume all file sizes are integers. Naturally, the sum of sizes of files stored in the cache can not exceed K, *i.e.*, $\sum_{f \text{ in cache}} s_f \le K$.

File Request Model. Let $\mathcal{R} = (r_1, r_2, \ldots)$ be the sequence of file requests, arriving in an online manner, *i.e.*, we can not get the future information and no assumption is made on the arrival patterns. Each r requests to access a specific file $f \in \mathcal{F}$. Time is divided into slots of unit size. Multiple different kinds of file requests might come within one time slot, while each file $f \in \mathcal{F}$ can be requested at most once in each slot [2].

Transmission Latency. When a request arrives at time T, if the requested file f is already in the local cache, then this request is called a *hit* and can be served immediately with no latency. Otherwise, it has to suffer a latency to fetch this file from the remote data center; alternatively, we might forward

[2]We set the time slot small enough so that the minimal interval of two consecutive requests on the same file is at least one time slot.

this request to get the file from the remote data center, *i.e.*, *bypassing* the request. We set the latency to fetch a file f taking z_f time slots, *i.e.*, this request can not be served until time $T+z_f$. We also set serving a request by bypassing taking z_f slots as it also needs to interact with the remote data center similar to fetching. When fetching a file, we need to decide which files should be evicted if the cache is already full. Before file f is fetched and stored in the cache, all requests that require file f at time slot $t' \in \{T+1, T+2, \ldots, T+z_f-1\}$ can only be served at time $T+z_f$ and suffer a latency of $z_f-(t'-T)$, which are *delayed hits*. The objective of this problem is to minimize the total latency of all requests.

Problem Formulation. Let $\mathcal{F}_t$ be set of files requested at time t, and $d(f,i)$ be the latency of the i-th request of f. Variable $x(f,i)$ indicates whether f is out of the cache after the i-th request of f, *e.g.*, $x(f,i)=0$ means f is in the cache or in the period of fetching after the i-th request of f, and $x(f,i)=1$ means f is out of the cache after the i-th request of f. Let $r(f,t)$ denote the number of requests for file f until time t. Let $\tau(f,i)$ be the time from $(i-1)$-th request of f to i-th request of f. We formulate the online general file caching problem with delayed hits and bypassing as follows:

Problem P:

$$\min \sum_{f\in\mathcal{F}} \sum_{i} d(f,i)$$

$$s.t. \sum_{f\in\mathcal{F}} s_f(1-x(f,r(f,t))) \le K, \qquad \forall t$$

$$x(f,i)\in\{0,1\}, \qquad \forall f,i$$

$$x(f,0)=1, \qquad \forall f$$

$$d(f,i)=z_f, \qquad \text{if } x(f,i-1)=1, \forall f,i$$

$$d(f,i)=\max\{d(f,i-1)-\tau(f,i),0\}, \qquad \text{if } x(f,i-1)=0, \forall f,i$$

Problem Hardness. When no bypassing is considered, Problem P has been proven to have a lower bound of the competitive ratio of $\Omega(ZK)$ [14] for deterministic algorithms. By using two kinds of request groups: pure and bursty requests similar to [14], we construct the request sequence to prove our general caching problem P has the following lower bounds for deterministic and randomized solutions.

Lemma 1. *All the deterministic online algorithms for problem P have a lower bound of the competitive ratio of $\Omega(ZK)$ to minimize the total latency, and all the randomized have a lower bound of the competitive ratio of $\Omega(Z\log K)$.*

Proof. Please refer to Appendix A. □

III. Online Algorithm

In this section, we first propose a parameter to measure the total latency caused by a file's miss, called *estimated weight*, to address the potential impact of the fetching process (Sec. III-A). Then, we present our algorithm `CaLa` (Algorithm 2) in detail in Sec. III-B. We also analyze the performance of `CaLa` in Sec. III-C, and prove that the deterministic version of `CaLa` is $O(Z^{3/2}K)$-competitive and the randomized version of `CaLa` is $O(Z^{3/2}\log K)$-competitive.

A. Estimated Weight

The central challenge of this problem is how to deal with delayed hits. In the design of MAD [6], they use the aggregate delay to capture the total latency caused by a file's miss. The aggregate delay of f at T can be calculated using Eqn. 1.

$$\begin{aligned}&\text{AggDelay}(f,T)\\ =&z_f+\sum_{1\le\tau\le z_f-1}(z_f-\tau)[f \text{ is requested at } T+\tau].\end{aligned} \tag{1}$$

The aggregate delay of file f can not be directly calculated in practice since it requires the future information of the next z_f time slots. MAD use the average aggregate delay of all the past requests for f to estimate the aggregate delay of the next request for f. However, this estimation is not always accurate hence the performance of MAD is not guaranteed. We show the gap between the estimated value and the real aggregate delay in Fig. 3. It shows that the estimated aggregate delay deviates from the real value.

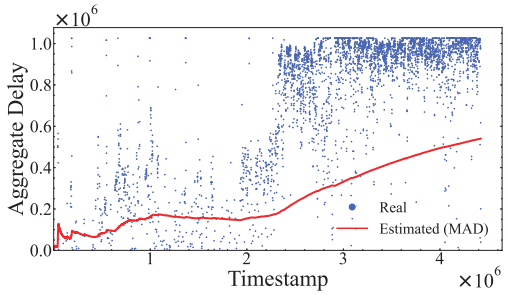


Fig. 3. The estimated aggregate delay [MAD, in SIGCOMM'20] *vs.* real aggregate delay calculated by Eqn. 1.

To avoid the impact of misestimation, we use the upper bound of the total latency caused by the file's miss, *i.e.*, z_i^2, to estimate the actual latency caused by this miss. As we will prove later, this estimation could preserve the competitive ratio of the internal algorithm within $O(Z^{3/2})\times$ extra cost.

Although using the upper bound to estimate the total latency of a miss yields algorithms with a performance guarantee, the actual performance of this method might be poor since it may give too much weight to some infrequently requested files. In general, the method of predicting the actual aggregate delay is radical and the method using the upper bound is conservative. To get a trade-off between these two kinds of methods, we use the following equation to represent the weight of each file:

$$W_f(T)=(1-\gamma)\,\text{AggDelay}(f,T)+\gamma z_f^2 \tag{2}$$

, where parameter γ is called *conservative parameter* and used to adjust between these two methods.

B. `CaLa`

The core part of `CaLa` is quite simple, which imitates the existing general file caching algorithm $\mathcal{A}$ while constantly

updating the weight of each requested file. By this means, CaLa can eliminate the impact of delayed hits while retaining the character of the original algorithm.

Algorithm 1: EstimatedWeight

Input *Parameter* γ*, file* f*, time* T
if f is not in the cache **then**
  f.cumulativeDelay $\leftarrow$ f.cumulativeDelay $+ z_f$;
  f.fetchingTime $\leftarrow T$;
  f.numFetching $\leftarrow$ f.numFetching $+ 1$;
if f.state $=$ OCCUPY **then**
  f.cumulativeDelay $\leftarrow$ f.cumulativeDelay $+ (z_f - (T - f.\text{fetchingTime}))$;
f.aggregateDelay $\leftarrow \dfrac{f.\text{cumulativeDelay}}{f.\text{numFetching}}$;
return $(1-\gamma) * f$.aggregateDelay $+ \gamma * z_f^2$;

Algorithm 2: CaLa

Input *Fetching Latency* z_i*, online caching algorithm* $\mathcal{A}$
Initialize the cache $\mathcal{C} \leftarrow \varnothing$,$\mathcal{C}_{\mathcal{A}} \leftarrow \varnothing$;
Fetching files $\mathcal{F}_{\text{fetching}} \leftarrow \varnothing$, element $(f,t) \in \mathcal{F}_{\text{fetching}}$ means file f will arrive at time t;
Timer $T \leftarrow 0$;
while True **do**
  for $(f,t) \in \mathcal{F}_{\text{fetching}}$ **do**
    if $t = T$ **then**
      if f.state $=$ OCCUPY **then**
        f.state $\leftarrow$ IN;
      Serve all the buffered requests for f;
  while new request for file f arrive at T **do**
    $W_f(T) \leftarrow \text{EstimatedWeight}(f, T)$;
    Let f arrive at $\mathcal{A}$ with weight $W_f(T)$;
    if $\mathcal{A}$ evicts $\mathcal{F}_{\text{evict}} \subseteq \mathcal{F} \setminus \{f\}$ from $\mathcal{C}_{\mathcal{A}}$ and puts f into $\mathcal{C}_{\mathcal{A}}$ **then**
      for $f' \in \mathcal{F}_{\text{evict}}$ **do**
        Evict f' from $\mathcal{C}$;
        f'.state $\leftarrow$ OUT;
      Put f into $\mathcal{C}$;
      f.state $\leftarrow$ OCCUPY;
      $\mathcal{F}_{\text{fetching}} \leftarrow \mathcal{F}_{\text{fetching}} \cup \{(f, T + z_f)\}$;
    if $\mathcal{A}$ bypasses f **then**
      if $(f,t) \in \mathcal{F}_{\text{fetching}}, \exists t > T$ **then**
        Buffer this request;
      else
        Bypass this request;
  $T \leftarrow T + 1$;

First, we introduce the algorithm to update estimated weights (Algorithm 1). This method is mainly adopted Algorithm 1 in [6]. When a new request for f arrives, if f is not in the cache then a new fetching period starts (Line 3 to Line 5). If the status of f is OCCUPY, it means that f is already in a fetching period, then we will accumulate the latency of this request to this fetching (Line 7). Then the aggregate delay of f is updated (Line 8) and the estimated weight of f can also be calculated (Line 9).

The details of CaLa are described in Algorithm 2. Initially, the cache of both CaLa and $\mathcal{A}$ are initialized (Line 2). When a new request for file f arrives, calculate its weight $W_f(T)$ by calling Algorithm 1 and send this request to $\mathcal{A}$ (Line 12 to Line 13). If $\mathcal{A}$ choose to evict some files $\mathcal{F}_{\text{evict}}$ in the cache to make room for storing file f, then CaLa evicts all the files in $\mathcal{F}_{\text{evict}}$ (Line 14 to Line 17) and reserve space for f (Line 18 to Line 20). If $\mathcal{A}$ choose to bypass this request, then CaLa also bypasses it (Line 25). When a file finishes its fetching, serve all the buffered requests (Line 6 to Line 10).

We use a modified version of Landlord, *i.e.*, Landlord with bypassing (LLB) [15], as the kernel of CaLa. Landlord [8] is an $O(K)$-competitive online algorithm for general file caching problem. It maintains a credit for each file to determine whether it should be evicted. Similar to Landlord, LLB also maintains a non-negative credit for each file. When a request for file f arrives, LLB will first set the credit of f as w_f, where w_f is the fetch cost of f. In the design of CaLa, we set $w_f = W_f(T)$ (Line 13 in Algorithm 2). Let G be a set of files consisting of all files in the cache and f. Then for all the files $g \in G$, decrease their credit by Δ times their size and delete zero-credit files in G, where Δ is the minimum value to zero the credit of a file, until the sum of files in G is no larger than K. If f remains in G in the end then fetch f to the cache, otherwise, bypass the request for f.

C. Analysis

To facilitate the proof, we define the following notations. Let $\text{ALG}(z_i)$ and $\text{OPT}(z_i)$ be respectively the total latency incurred by CaLa and offline optimal solution in the model of general file caching with delayed hits and bypassing if the latency to fetch f_i or bypass request for f_i is z_i. Let $\mathcal{A}(z_i)$ and $\text{OPT}'(z_i)$ be respectively the total cost of online algorithm $\mathcal{A}$ and offline optimal solution of general file caching with bypassing, where $\mathcal{A}$ is c-competitive and z_i is the cost to fetch or bypass f_i. Similarly, $\mathcal{A}(z_i^2)$ and $\text{OPT}'(z_i^2)$ are the total cost when the cost to fetch or bypass f_i is z_i^2. Clearly, we have $\mathcal{A}(z_i^2) \leq c \cdot \text{OPT}'(z_i^2)$.

Lemma 2. $\text{ALG}(z_i) \leq \mathcal{A}(z_i^2)$.

Proof. We define fetching group of f_i as all requests to f_i from a fetching of f_i to the next fetching or bypassing of f_i. Clearly, each fetching group of f_i only contain a single fetching of f_i followed by zero or more delayed hits of f_i. Since each fetching of f_i at most causes $z_i - 1$ delayed hits, the fetching latency of each fetching group of f_i of $\text{ALG}(z_i)$ is at most $z_i(z_i+1)/2$. On the other hand, the fetching latency of each fetching group of f_i of $\mathcal{A}(z_i^2)$ is exactly z_i^2. For each bypassing of f_i in $\mathcal{A}$, the corresponding latency of f_i in CaLa is no larger than z_i and the cost in $\mathcal{A}$ is z_i^2. By definition, each request in $\mathcal{A}$

is ether in a fetching group or a bypassing. Since CaLa follows the operations of $\mathcal{A}$, the fetching group of $\mathrm{ALG}(z_i)$ is exactly the same as $\mathcal{A}(z_i^2)$. Thus, $\mathrm{ALG}(z_i) \leq \mathcal{A}(z_i^2)$. □

Fact 1. For general file caching with bypassing, let $\mathcal{I}_1$ and $\mathcal{I}_2$ be two input sequences that request the same files, where the cost to fetch files in $\mathcal{I}_1$ are $(w_1, w_2, \ldots, w_n)$ and the cost to fetch files in $\mathcal{I}_2$ are $(\alpha w_1, \alpha w_2, \ldots, \alpha w_n)$. Then we have $\mathrm{OPT}'(\mathcal{I}_1) = \alpha \mathrm{OPT}'(\mathcal{I}_2)$.

Fact 2. For general file caching with bypassing, let $\mathcal{I}_1$ and $\mathcal{I}_2$ be two input sequences that request the same files, where the cost for files in $\mathcal{I}_1$ are $(w_1, w_2, \ldots, w_n)$ and the cost for files in $\mathcal{I}_2$ are $(w'_1, w'_2, \ldots, w'_n)$ and assume $w_1 \leq w'_1, w_2 \leq w'_2, \ldots, w_n \leq w'_n$. Then we have $\mathrm{OPT}'(\mathcal{I}_1) \leq \mathrm{OPT}'(\mathcal{I}_2)$.

By using Fact 1 and Fact 2, we have the following lemma.

Lemma 3. $\mathrm{OPT}'(z_i^2) \leq Z \cdot \mathrm{OPT}'(z_i)$.

Then, we get the connection between the optimal of file caching and optimal of problem P by the following lemma.

Lemma 4. $\mathrm{OPT}'(z_i) \leq Z^{1/2}\mathrm{OPT}(z_i)$.

Proof. Similar to the proof of Lemma 2, define fetching group of f_i as all requests to f_i from a fetching of f_i to the next fetching of f_i. For each fetching group of f_i, it contains a fetching operation and zero or more delayed hits. Let m be the number of delayed hits in this fetching group ($0 \leq m \leq z_i - 1$) and $d_1, d_2, \ldots, d_m$ be the latency caused by these delayed hits respectively. Thus, the average latency caused by miss of delayed hits is at least $(z_i + \sum_{i=1}^{m} d_i)/(m+1) \geq (z_i + \sum_{i=1}^{m} i)/(m+1) \geq \sqrt{z_i}$ in $\mathrm{OPT}(z_i)$. Since the solution of $\mathrm{OPT}(z_i)$ is a feasible solution in the model without delayed hits and the bypassing latency in these two models are both z_i, we have $\mathrm{OPT}'(z_i) \leq Z^{1/2}\mathrm{OPT}(z_i)$. □

By combining Lemma 2, Lemma 3 and Lemma 4 together, we have the following theorem.

Theorem 1. *If there is an online file caching algorithm $\mathcal{A}$ with bypassing is c-competitive, CaLa is $O(Z^{3/2}c)$-competitive for the online file caching problem with heterogeneous fetching latency and bypassing by setting $\gamma = 1$.*

It should be noted that by using a similar method to prove, we can get the same result in the case without bypassing. Since there are deterministic $O(K)$-competitive online algorithm and randomized $O(\log K)$-competitive online algorithm for general file caching [15], we have the following corollary.

Corollary 1. *By setting $\gamma = 1$, the deterministic version of CaLa is $O(Z^{3/2}K)$-competitive, and the randomized version of CaLa is $O(Z^{3/2}\log K)$-competitive.*

IV. Evaluation

We evaluate the performance of CaLa on two datasets: (1) the production trace from Google [13], and (2) the system benchmark of YCSB workloads from Yahoo [16], which is used widely in previous works (*e.g.*, [3], [17], [18]). We compare CaLa with several state-of-the-art methods, *i.e.*, LRU [1], LRU-MAD [6], Landlord [8], and Landlord with bypassing [15]. The details of experiment results are shown in Sec. IV-C and we highlight our key findings as follows.

- Compare with LRU-MAD, the state-of-the-art algorithm deals with delayed hits. Among all settings, CaLa can reduce latency by up to 9.42% without bypassing, this reduction will be increased to 32.01% if bypassing is allowed.
- CaLa can achieve a similar hit ratio to LRU-MAD, and evicts more large files to make space for more frequent and high latency files. Furthermore, CaLa with bypassing achieves a higher hit ratio by bypassing infrequent requests to the remote data center.
- If the cache size is small (*e.g.*, sum of 0.1% to 0.5% of the active files), CaLa with bypassing outperforms other algorithms significantly by bypassing.

A. Methodology

We set the cache size in a way similar to [6], where the cache size is the sum of the sizes of the most active files. The default cache size is the sum of the sizes of top 1% active files. For CaLa, the default value of γ is set to 0.1.

Workloads. There are $4.4M$ and $2.8M$ requests in Google's production trace and YCSB benchmark, respectively. It should be noted that the request sequence patterns of these two traces are totally different. The requests in Google's trace for the same file are usually arriving continuously, while the requests in YCSB benchmark are arriving individually. To express this more clearly, we define the *request locality* of a sequence as the ratio of the number of requests that followed by requests require the same file and the number of total requests. The request locality of Google's trace and YCSB benchmark are 0.7058 and 0.0025, respectively. For Google's production trace, we use "RAM Used" as the size of the file. The size of the file of YCSB benchmark is generated with exponential distribution, and its mean value is set to be close to Google's trace. By default, we set the average inter-request time to $100\mu s$, *i.e.*, $10K$ requests arrive in a second. For reference, the peak number of requests per minute during a flash crowd is about $35K$ [19]. The average default latency of files is set to $100ms$ (*i.e.*, the average value of z_f for each file f is 1000), which is the approximate latency to fetch files from remote data center [5]. Since both traces lack the information of the file's fetching latency, we randomly generate a latency uniformly distributed within $(0, Z_{\mathrm{upper}})$ for each file, where Z_{upper} is $2\times$ the average fetching latency.

Metrics. The metrics used to evaluate the performance of algorithms is the total latency incurred of all requests, including the latency caused by misses, delayed hits or bypassing. Furthermore, we use the latency improvement relative to LRU to measure the performance of the algorithm when the parameters change, which can be calculated by

$$\text{Latency Improvement of A} = \frac{\text{Latency(LRU)} - \text{Latency(A)}}{\text{Latency(LRU)}}.$$

A higher latency improvement means a better performance.

B. Baseline Algorithms

We compare the performance of our proposed algorithms CaLa ($\gamma = 1$) without bypassing, CaLa without bypassing and CaLa with bypassing with the following baselines for minimizing the total latency.

LRU [1]. Least Recently Used is the most classic algorithm in the caching problem, which will evict the file that has not been used for the longest time. LRU is $O(K)$-competitive for the paging problem. Due to the locality of requests, LRU generally performs well in a production environment.

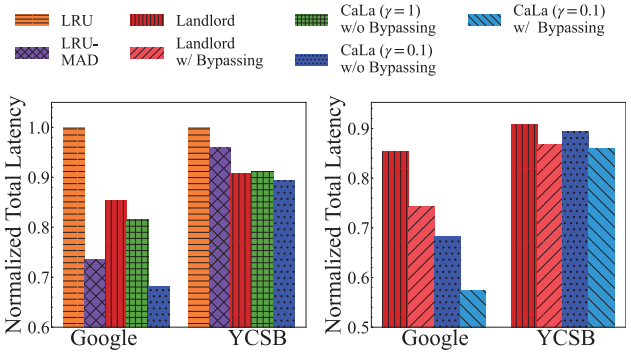
LRU-MAD [6]. LRU-MAD is the state-of-the-art caching algorithm that deals with delayed hits by calculating each files' rank. The rank of a file is the aggregate delay of this file divided by the time since its last request and LRU-MAD will evict the file with the lowest rank when the cache is out of space. Although in the system model of [6] all the file has the same fetching latency, LRU-MAD is aware of heterogeneous fetching latency because of the calculation of aggregate delay.

Landlord [8]. Landlord is an algorithm for online general file caching, which has a competitive ratio of $O(K)$. The core of Landlord is to maintain a credit for each file and evict all the zero-credit files. For each file, its credit is set to its cost (*i.e.*, fetching latency in this paper) when it is requested. Credit for all files in the cache will be decreased by a value proportional to the size of the file.

Landlord with Bypassing [15]. To support bypassing, Landlord with bypassing sets the credit of the new requested file to its cost first. Then decrease all the credit of files in the cache and the new requested file. Similar to Landlord, all the zero-credit files will be evicted. If the credit of the new requested file is decreased to zero, then bypass it.

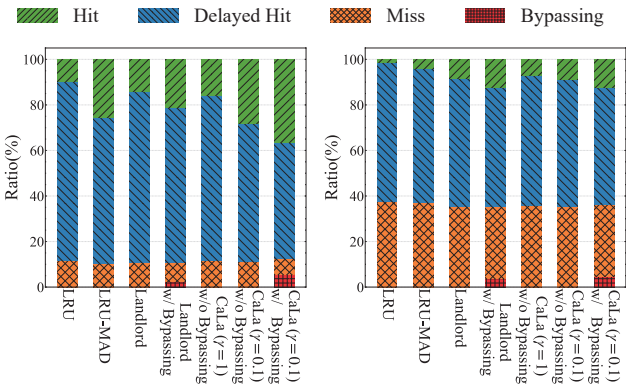
C. Experiment Results

Overall Result. We first evaluate the overall performance of CaLa ($\gamma = 1$) without bypassing, CaLa without bypassing and CaLa with bypassing, and compare them with LRU, LRU-MAD, Landlord and Landlord with bypassing, where parameters are set as default values. The experimental results are shown in Fig. 4, where the total latency of each algorithm is normalized so that the total latency of LRU is 1. Fig. 4(a) illustrates the results without bypassing. The latency improvements of CaLa to LRU, LRU-MAD and Landlord in Google's trace are 31.76%, 7.10% and 19.99%, respectively. For the results in YCSB benchmark, the latency improvements of CaLa are 10.58%, 6.81% and 1.42%, respectively. We show the result of latency improvement of bypassing in Fig. 4(b). It shows that if bypassing is allowed, compared with the situation without bypassing, CaLa reduces 15.79% and 3.87% latency on Google's trace and YCSB benchmark, respectively. The performance gain of bypassing can also be seen from the improvement of Landlord with bypassing compared to Landlord. We also found the performance of LRU-MAD is better than Landlord in Google's trace, while the opposite result is shown in YCSB benchmark. This phenomenon indicates aggregate delay captured burst requests and failed to handle the sequence without locality, and CaLa performs well in both cases.



(a) Total latency w/o bypassing. (b) Total latency w/ bypassing.

Fig. 4. Overall performance.

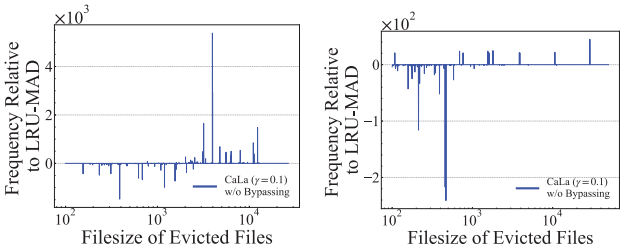


(a) Google (b) YCSB

Fig. 5. Ratio of hit, delayed hit, miss and bypassing.

Ingredient of Latency. To explore the factors that affect the algorithm performance, we show the ratio of hit, delayed hit, miss and bypassing of each algorithm in Fig. 5. Firstly, we can find that the hit ratio generally determines the final performance of an algorithm. In the result of Google's trace, the hit ratio of LRU-MAD is the highest among all the algorithms without bypassing. CaLa also gets a hit ratio close to LRU-MAD, but achieves a better performance. This is because CaLa tends to evict more large files and reserve more space for high latency files. When bypassing is allowed, the hit rate will be further improved and better performance will be obtained. This phenomenon is more obvious in the results of YCSB benchmark. There are always requests for infrequent files that cause inevitable misses. Thus the hit ratios are very low for all the algorithms without bypassing. By bypassing some of these requests, part of frequent files will not be evicted and hence the hit ratio of Landlord with bypassing and CaLa with bypassing are significantly increased.

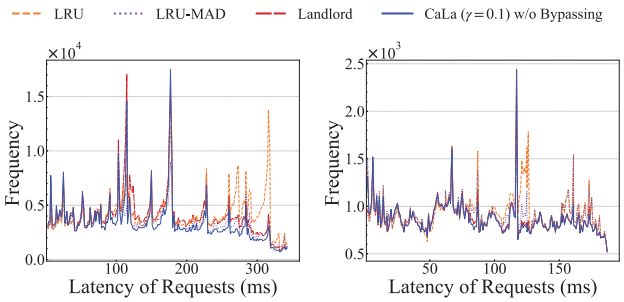
Size of Evicted Files. We investigate the sizes of evicted files of different algorithms in Fig. 6. The distributions of LRU and LRU-MAD are roughly close, whlie Landlord and CaLa are around close. Due to the limitation of space, we only plot the difference between LRU-MAD and CaLa. For each size value, we count the number of files with this size are evicted. The



(a) Google (b) YCSB

Fig. 6. Distribution of size of evict files, where the size is normalized so that the size of the smallest file is 1.

value on the y-axis represents the number of files evicted by CaLa compared with LRU-MAD. It shows that CaLa evicts more large files, making more space left for files with high frequency and high latency.



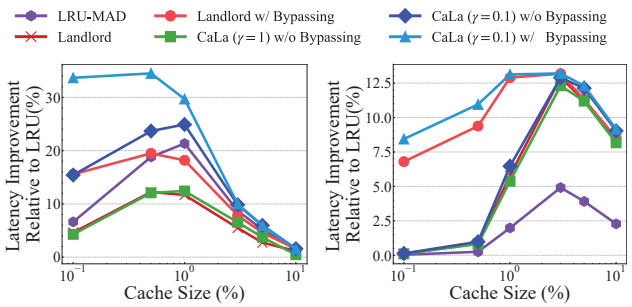
(a) Google (b) YCSB

Fig. 7. Distribution of the latency of requests

Latency of Requests. We plot the distribution of latency of requests in Fig. 7, including latency caused by bypassing, misses and delayed hits, where the height of a point for a specific latency represents the number of requests served at this latency. In the Google's trace, LRU has more high latency requests, resulting in its overall poor performance. With the help of aggregate delay, LRU-MAD is much better to reduce the number of high latency requests. However, in the YCSB benchmark, the distribution of LRU-MAD is close to LRU, which means aggregate delays are not that effective when the requests are not bursty. Besides, we can observe that CaLa can better avoid missing high latency files in both traces.

D. Sensitivity Study

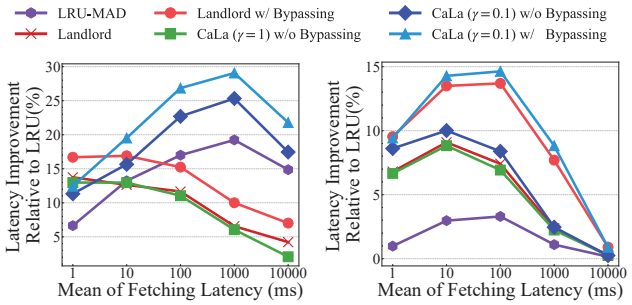
Impact of Cache Size. To investigate the impact of cache size, we change the cache size from 0.1% to 10% and show the results in Fig. 8, where the metrics to measure the performance of algorithms are latency improvement relative to LRU. First, when the cache size is small (*e.g.*, sum of 0.1% to 0.5% of the active files), CaLa with bypassing performs far beyond other algorithms. This is because bypassing can avoid evicting some frequently requested files in the cache and reduce the number of misses and delayed hits, and this situation happens more frequently when the cache size is small. As the cache size gradually increases, the performance of CaLa gradually



(a) Google (b) YCSB

Fig. 8. Impact of cache size.

catches up with CaLa with bypassing. It should be noted that due to the discreteness of files' size in the trace, for different algorithms the performance improvement brought by the additional cache size does not occur simultaneously as the cache size increases, which causes the fluctuations in performance curves. Finally, when the cache size is large enough, almost all the frequent files can be stored in the cache and the performance of all algorithms tends to be the same.

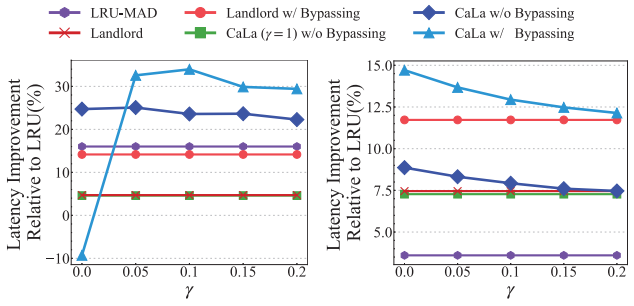


(a) Google (b) YCSB

Fig. 9. Impact of fetching latency.

Impact of Fetching Latency. We show the result of the impact of fetching latency in Fig. 9, where the fetching latency change from 10 to 100000 time slots. The performance of LRU-MAD, CaLa and CaLa with bypassing start increasing when the fetching latency becomes higher since their awareness of latency and delayed hits. In the result of YCSB benchmark, the performance of algorithms is more likely the case without delayed hits. The reason is that there are few requests with delayed hits, especially when the latency is relatively small. When the average fetching latency becomes very large, in both traces these algorithms tend to have similar performance, since almost all the requests are misses or delayed hits. The fluctuation of the curves in Fig. 9 reflects the different sensitivity of various algorithms to the fetching latency.

Impact of γ. As shown in Fig. 10, for the Google's trace, the best performance is achieved when $\gamma = 0.05$ or $\gamma = 0.1$, which shows that it is better to use a value of γ closer to the aggregate delay for burst requests. In the YCSB benchmark test, the best performance can be obtained by setting $\gamma = 0$. However, CaLa with bypassing performs extremely bad when



(a) Google (b) YCSB

Fig. 10. Impact of γ.

$\gamma = 0$ in Google's trace, which indicates that the estimation error of aggregate delay will seriously affect the results of bypassing, especially in the case of burst requests.

V. Related Works

Theoretical Results of Caching. The first systematic study of the performance analysis of caching algorithm is presented by Sleator and Tarjan [1], which shows that LRU and FIFO are $\frac{k}{k-h+1}$-competitive and no deterministic online algorithm can achieve a better competitive ratio. Here, k and h are the cache size of the online algorithm and offline optimal, respectively. Fiat *et al.* proposed the first online paging algorithm Marking [7] with $2H_k$-competitive and showed no randomized online algorithm could be better than H_k-competitive. For the caching problem with nonuniform file size, Irani [20] proposed a general method to transfer this problem to the uniform setting and gave an online algorithm with $O(\log^2 k)$ when the fetch cost of a file equals 1 or its size. Jiang [21] *et al.*studied the weighted paging problem and gave a lower bound of $O(\log k)$ in the PRP model. Then they proposed a stronger model called SPRP and gave an algorithm with 2-competitive. For the most general setting of nonuniform file size and fetch cost, Bar-Noy *et al.* [22] gave a 4-approximate algorithm for the offline version and Adamaszek *et al.* [23] showed a tight online algorithm with $O(\log k)$-competitive. Tan *et al.* studied the caching variant in edge computing, where the system contains multiple caches and the requests can be relayed to other cache, and gave an $O(\log k)$-competitive online algorithm for this problem. Lykouris and Vassilvitskii [24] first studied the online paging problem with machine learning advice and gave an algorithm with $O(1+\min(\sqrt{\eta/OPT}, \log k))$, where η is the total absolute loss and OPT is the cost of offline optimal. Based on this work, Rohatgi [25] improved the theoretical result to $O(1+\min((\eta/OPT)/k, 1)\log k)$ and provided a lower bound of $\Omega(\log\min((\eta/OPT)/(k\log k), k))$.

Caching Algorithms in CDNs. Some works explore more valuable features to optimize cache performance based on the actual production environment than given performance guarantees. Hu *et al.* [26] uses data locality to minimize the average response time of key-value caches. Beckmann *et al.* [12] proposed the algorithm LHD to predict the hit density of each object to filter objects that have a small contribution to the cache hit rate. Berger *et al.* [27] proposed AdaptSize, an adaptive, size-aware cache admission policy for hot object cache in CDN. Berg *et al.* [28] showed CacheLib, a general-purpose caching engine, extracts a core set of common requirements and functionality from otherwise disjoint caching systems. Ye *et al.* [29] proposed a learning framework to learn the joint cache size scaling and strategy adaptation policy for Elastic CDN. Zong *et al.* [30] proposed Cocktail Edge Caching, which employed an ensemble of constituent caching policies and adaptively selected the best-performing approach to control the cache. Song *et al.* [31] proposed LRB to mimic the relaxed Belady's MIN algorithm by using Gradient Boosting Machines [32]. Akhtar *et al.* [33] described AviC, a caching algorithm that leverages properties of video delivery to design the eviction policy in CDN. Zhou *et al.* [34] introduced Bounded Linear Probing (BLP), a cache design by balancing hit rate and lookup latency for network appliances. Jin *et al.* [35] presented NetCache, a key-value store architecture that balances the load across storage nodes. Garetto *et al.* [36] provided a first comprehensive analysis of similarity caching in different settings. Atre *et al.* [6] first studied the caching problem with delayed hits and proposed a heuristic to estimate the latency caused by a miss. Paper [14] introduced the lower bound of caching with delayed hits for deterministic solutions.

In this work, we first extend the lower bound to randomized algorithms, then propose a general framework to transform an existing competitive algorithm for the general file caching problem to address delayed hits with a performance guarantee. We summarize the related theoretical results in Table I.

TABLE I
Theoretical Results of Caching Problems

Algorithms	File Size & Fetch Cost	Delayed Hits	Bypassing	Performance Guarantee	Type*
LRU [1]	Uniform	✘	✘	$O(K)$	D
Marking [7]	Uniform	✘	✘	$O(\log K)$	R
Landlord [8]	Non-uniform	✘	✘	$O(K)$	D
Adamaszek *et al.* [23]	Non-uniform	✘	✘	$O(\log K)$	R
Camul-det, Camul [3]	Uniform	✘	✔	$O(K)$ $O(\log K)$	D R
LLB [15]	Non-uniform	✘	✔	$O(K)$	D
MAD [6]	Uniform	✔	✘	-	D
`CaLa` [this work]	Non-uniform	✔	✔	$O(Z^{3/2}K)$ $O(Z^{3/2}\log K)$	D R

* D: Deterministic Algorithm, R: Randomized Algorithm.

VI. Discussion

More Accurate Estimated Weight. For the estimated weight, we only use a rough method, by setting a parameter γ, to linearly combine the aggregate delay and its upper bound. For various input sequences, the optimal value of γ might be completely different. There could be quite some promising direc-

tions to estimate more accurately the total latency of a request miss and find a better adaptive way to set the estimated weight. For example, the way to estimate aggregate delay should be highly correlated with time. Moreover, the combination could be more complex, *e.g.*, with more estimators but not just a simple linear combining. For more complicated applications, deep reinforcement learning based methods might work well. We leave this estimation improvement as our future work.

Portraying the Fetching Latency. In this work, we prove that CaLa can transform an existing file caching algorithm to handle delayed hits with extra $O(Z^{3/2})\times$ cost. This transformation is not tight since the lower bounds of file caching with delayed hits are $\Omega(ZK)$ and $\Omega(Z\log K)$ for deterministic algorithm and randomized algorithm. The gap between CaLa and the lower bound may not reflect the performance in practice, since the parameter chooses to portray the fetching latency, *i.e.*, Z, is just a rough estimate of the overall data. For example, when the latency of all files increases to Z, the theoretical performance bound of CaLa remains the same, while the actual total latency of CaLa may substantially increase. There might be other parameters that potentially better describe the request sequence.

VII. Conclusion

In this paper, we study the general online file caching problem with delayed hits and bypassing, where the objective is to minimize the total latency of all the requests. We first prove lower bound $\Omega(ZK)$ and $\Omega(Z\log K)$ for deterministic algorithms and randomized algorithms, respectively. Then we propose a general framework, *i.e.*, CaLa, which estimates the latency of each request and then imitates an existing file caching algorithm to get guaranteed performance. We prove that the deterministic version and randomized version of CaLa have a competitive ratio of $O(Z^{3/2}K)$ and $O(Z^{3/2}\log K)$, respectively. We evaluate CaLa based on Google's trace and YCSB benchmark. The experiment results show that compare with LRU-MAD, CaLa can reduce the latency by up to 9.42% without bypassing. Furthermore, this reduction will be increased to 32.01% if bypassing is allowed.

Acknowledgement

This work is partially supported by National Key R&D Program of China 2018YFB0803400, China National Funds for Distinguished Young Scientists No. 61625205, NSFC No. 61772489, 62132009, 62132018, Key Research Program of Frontier Sciences, CAS. No. QYZDY-SSW-JSC002, MST of China No. 2021YFA1000900, the Fundamental Research Funds for the Central U. in China, and the University Synergy Innovation Program of Anhui Province with No. GXXT-2019-024. Haisheng Tan is the corresponding author (email: hstan@ustc.edu.cn).

Appendix A
Proof of Lemma 1

Proof. We define two kinds of request groups, *pure* and *bursty* requests, similar to [14]. A pure request for f_i consists of $Z+1$ time slots, where the first slot requests f_i, and the following Z slots do not request any file. A bursty request for f_i consists of $2Z$ slots, where the first Z slots request file f_i, and the next Z slots do not request any file. If a pure or bursty request is hit, there will be no latency accrued. If a pure request is missed, the latency caused is Z; and, if a bursty request is missed, the latency caused is at least $\frac{Z(Z+1)}{2}$. Let r_i^p and r_i^b be pure and bursty request for f_i, respectively. Assume totally $K+1$ different files will be requested.

Deterministic Algorithm. Let $\mathcal{A}$ be a deterministic online algorithm for problem P. Without loss of generality we assume that files $f_1,\ldots,f_K$ are stored in the cache initially. First, the constructor requests r_{K+1}^p. Since the cache size is K, there is at least one file out of the cache whether bypassing is allowed or not. Then repeat bursty requests for K times. The j-th bursty request is $r_{i_j}^b$, where f_{i_j} is the file not in the cache of $\mathcal{A}$ just before j-th bursty request. Thus, for each bursty request, the latency caused by $\mathcal{A}$ is at least $\frac{Z(Z+1)}{2}$, and the total latency of $\mathcal{A}$ is $Z+K\frac{Z(Z+1)}{2}$. By contrast, the latency of optimal is only Z caused by r_{K+1}^p.

Randomized Algorithm. Let $\mathcal{A}$ be a randomized online algorithm for problem P. When we construct the request sequence $\sigma_{\mathcal{A}}$ we can maintain a vector $\boldsymbol{p}=(p_1,p_2,\ldots,p_{K+1})$ of probabilities, where p_i is the probability that file f_i is not in cache. Since there is only one file not in the cache, we have $\sum_i p_i = 1$. Note that this vector of probabilities is valid whether bypassing is allowed or not. Similar to the marking algorithm, the constructor also maintains whether each file is *marked*, and divides the request sequence into several consecutive phases based on these markers. A file is marked when it was required in the current phase. When the number of marked files reaches $K+1$, a new phase starts and all files except the file just requested are set to *unmarked*. In general, each phase contains requests for exactly K different files and starts with a request requiring a file not required in last phase. Each phase then is divided into K subphases, where each subphase consists of several requests for marked files and ends with an unmarked file.

The sequence constructor can generate a sequence such that the expected latency of each phase to $\mathcal{A}$ is at least $\frac{Z(Z+1)}{2}H_K$, and the latency to the optimal is Z. Without loss of generality, we assume that files $f_1,\ldots,f_K$ are stored in the cache at the beginning of this phase. The first request in this phase is r_{K+1}^p. Let u be the number of unmarked files. Let $\mathcal{M}$ be the set of marked files. Let $P=\sum_{f_i\in\mathcal{M}} p_i$. If $P=0$ then there must be an unmarked file f_i with $p_i\geq 1/u$ and this subphase contains a single request to r_i^b. The expected latency of this request is at least $\frac{Z(Z+1)}{2}\frac{1}{u}$. If $P>0$ then continuously require r_i^b until the total expected latency of this subphase exceeding $\frac{Z(Z+1)}{2}\frac{1}{u}$ if this subphase ends with a request of an unmarked file, where $f_i\in\mathcal{M}$ and $p_i>0$. Finally, u takes all the integer between 1 and K thus the total latency of $\mathcal{A}$ is $\frac{Z(Z+1)}{2}H_K$, whereas the total latency of optimal is Z caused by r_{K+1}^p. □

REFERENCES

[1] 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.

[2] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl, "Globally distributed content delivery," *IEEE Internet Computing*, vol. 6, no. 5, pp. 50–58, 2002.

[3] H. Tan, S. H.-C. Jiang, Z. Han, L. Liu, K. Han, and Q. Zhao, "Camul: Online caching on multiple caches with relaying and bypassing," in *IEEE INFOCOM 2019*.

[4] R. Krishnan, H. V. Madhyastha, S. Srinivasan, S. Jain, A. Krishnamurthy, T. Anderson, and J. Gao, "Moving beyond end-to-end path information to optimize CDN performance," in *ACM IMC 2009*.

[5] X. Fan, E. Katz-Bassett, and J. Heidemann, "Assessing affinity between users and CDN sites," in *International Workshop on Traffic Monitoring and Analysis*. Springer, 2015, pp. 95–110.

[6] N. Atre, J. Sherry, W. Wang, and D. S. Berger, "Caching with delayed hits," in *ACM SIGCOMM 2020*.

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

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

[9] D. Genbrugge and L. Eeckhout, "Memory data flow modeling in statistical simulation for the efficient exploration of microprocessor design spaces," *IEEE Transactions on Computers*, vol. 57, no. 1, pp. 41–54, 2007.

[10] M. V. Wilkes, "Slave memories and dynamic storage allocation," *IEEE Transactions on Electronic Computers*, no. 2, pp. 270–271, 1965.

[11] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache." in *USENI FAST 2003*.

[12] N. Beckmann, H. Chen, and A. Cidon, "LHD: Improving cache hit rate by maximizing hit density," in *USENIX NSDI 2018*.

[13] C. Reiss, J. Wilkes, and J. Hellerstein, "Google cluster-usage trace," in *Technical Report*, 2011.

[14] P. Manohar and J. Williams, "Lower bounds for caching with delayed hits," *arXiv preprint arXiv:2006.00376*, 2020.

[15] 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.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *ACM SoCC 2010*.

[17] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, "Flatstore: An efficient log-structured key-value storage engine for persistent memory," in *ACM ASPLOS 2020*.

[18] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, H. Jiang, C. Xie, and X. He, "MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM," in *USENIX ATC 2020*.

[19] P. Wendell and M. J. Freedman, "Going viral: flash crowds in an open CDN," in *ACM/USENIX IMC 2011*.

[20] S. Irani, "Page replacement with multi-size pages and applications to web caching," in *ACM STOC 1997*.

[21] Z. Jiang, D. Panigrahi, and K. Sun, "Online algorithms for weighted paging with predictions," *arXiv preprint arXiv:2006.09509*, 2020.

[22] A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. Naor, and B. Schieber, "A unified approach to approximating resource allocation and scheduling," *Journal of the ACM (JACM)*, vol. 48, no. 5, pp. 1069–1090, 2001.

[23] A. Adamaszek, A. Czumaj, M. Englert, and H. Räcke, "An O(log k)-competitive algorithm for generalized caching," *ACM Transactions on Algorithms (TALG)*, vol. 15, no. 1, pp. 1–18, 2018.

[24] T. Lykouris and S. Vassilvitskii, "Competitive caching with machine learned advice," *arXiv preprint arXiv:1802.05399*, 2018.

[25] D. Rohatgi, "Near-optimal bounds for online caching with machine learned advice," in *SIAM SODA 2020*.

[26] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang, "LAMA: Optimized locality-aware memory allocation for key-value cache," in *USENIX ATC 2015*.

[27] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter, "Adaptsize: Orchestrating the hot object memory cache in a content delivery network," in *USENIX NSDI 2017*.

[28] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter *et al.*, "The cachelib caching engine: Design and experiences at scale," in *USENIX OSDI 2020*.

[29] Y. Jiahui, L. Zichun, W. Zhi, Z. Zhuobin, H. Han, and Z. Wenwu, "Joint cache size scaling and replacement adaptation for small content providers," in *IEEE INFOCOM 2021*.

[30] T. Zong, C. Li, Y. Lei, G. Li, H. Cao, and Y. Liu, "Cocktail edge caching: Ride dynamic trends of content popularity with ensemble learning," in *IEEE INFOCOM 2021*.

[31] Z. Song, D. S. Berger, K. Li, and W. Lloyd, "Learning relaxed belady for content distribution network caching," in *USENIX NSDI 2020*.

[32] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.

[33] Z. Akhtar, Y. Li, R. Govindan, E. Halepovic, S. Hao, Y. Liu, and S. Sen, "Avic: a cache for adaptive bitrate video," in *ACM CoNEXT 2019*.

[34] D. Zhou, H. Yu, M. Kaminsky, and D. Andersen, "Fast software cache design for network appliances," in *USENIX ATC 2020*.

[35] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *ACM SOSP 2017*.

[36] M. Garetto, E. Leonardi, and G. Neglia, "Similarity caching: Theory and algorithms," in *IEEE INFOCOM 2020*.