Asymptotically Tight Approximation for Online File Caching With Delayed Hits and Bypassing

Haisheng Tan, *Senior Member, IEEE*, Yi Wang, Chi Zhang, Guopeng Li, Haohua Du, Zhenhua Han, Shaofeng H.-C. Jiang, and Xiang-Yang Li, *Fellow, IEEE*

***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 for the same missing file before the fetching process 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 present a general reduction that turns a traditional file caching algorithm into one that can handle delayed hits. Based on this reduction, we propose an efficient online file caching algorithm, called `CaLa`, with an asymptotically tight competitive ratio as $O(Z \log K)$, where Z is the maximum fetching latency of any file and K is the cache size. Extensive simulations on the production data trace from Google and the Yahoo benchmark illustrate that `CaLa` can reduce the latency by up to 8.48% compared with the state-of-the-art schemes dealing with delayed hits without bypassing, and this improvement increases to 26.00% if bypassing is allowed. Furthermore, by upgrading the method for estimating files' weights in `CaLa`, we propose `CaLa+`, which further reduces the total latency by more than 5%.**

***Index Terms*—Approximation algorithms, content distribution networks, cache storage, edge computing.**

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 [2].

However, in practical applications such as Content Delivery Networks (CDNs) [3] and Mobile Edge Computing (MEC) [4], due to the long physical distance, the latency for fetching missing files from the remote data center can be more than $100ms$ [5], [6], whereas the average inter-time for two consecutive file requests could be as low as $1\mu s$ [7], *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 cannot 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* [7]. Moreover, traditional cache models [2], [8], [9], [10], [11] 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 file misses, hits, delayed hits and bypassing, where there is 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

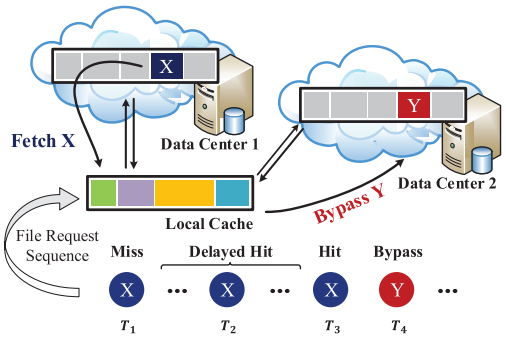


Fig. 1. An example of an online file caching system, where a file request may be served at the local cache or bypassed directly to the remote data center.

Received 25 November 2023; revised 26 August 2024; accepted 28 February 2025; approved by IEEE TRANSACTIONS ON NETWORKING Editor G. Joshi. This work was supported in part by the 2030 National Key AI Program of China under Grant 2021ZD0110400, in part by NSFC under Grant 62132009 and Grant 62102016, in part by the Fundamental Research Funds for the Central Universities at China, and in part by the Startup Fund from Peking University. A preliminary version of this work titled "Online File Caching in Latency-Sensitive Systems with Delayed Hits and Bypassing" was published in IEEE INFOCOM 2022, London, United Kingdom, May, 2022 [DOI: 10.1109/INFOCOM48880.2022.9796969]. *(Corresponding author: Chi Zhang.)*

Haisheng Tan, Yi Wang, Guopeng Li, and Xiang-Yang Li are with the LINKE Laboratory and the CAS Key Laboratory of Wireless-Optical Communications, University of Science and Technology of China (USTC), Hefei 230027, China (e-mail: hstan@ustc.edu.cn; wangyi1024@mail.ustc.edu.cn; guopengli@mail.ustc.edu.cn; xiangyangli@ustc.edu.cn).

Chi Zhang is with the School of Computer Science and Information Engineering, Hefei University of Technology, Hefei 230009, China, and also with the LINKE Laboratory and the CAS Key Laboratory of Wireless-Optical Communications, University of Science and Technology of China (USTC), Hefei 230027, China (e-mail: zhangchi@hfut.edu.cn).

Haohua Du is with the School of Cyber Science and Technology, Beihang University, Beijing 100191, China (e-mail: duhaohua@buaa.edu.cn).

Zhenhua Han is with Microsoft Research Asia (MSRA), Shanghai 200232, China (e-mail: hzhua201@gmail.com).

Shaofeng H.-C. Jiang is with the Center on Frontiers of Computing Studies, Peking University, Beijing 100871, China (e-mail: shaofeng.jiang@pku.edu.cn).

Digital Object Identifier 10.1109/TON.2025.3549289

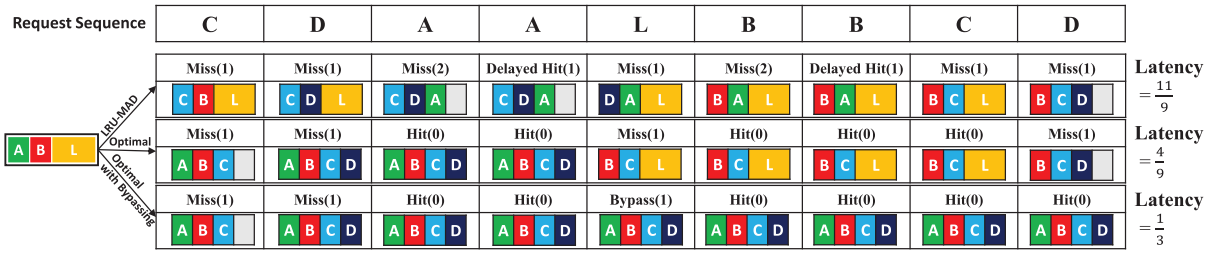


Fig. 2. An example in general file caching with delayed hits and bypassing, where files have heterogeneous size and latency. The size of cache $K = 4$, files' size $s_A = s_B = s_C = s_D = 1$, $s_L = 2$, fetching latency $z_A = z_B = 2$, $z_C = z_D = z_L = 1$. The average latency of LRU-MAD, Optimal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

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.

So far, although noted in the literature (*e.g.*, [12]), research on online file caching with delayed hits is still limited. A representative work addressing this issue is the online paging problem[1] studied in [7], where the authors highlighted the importance of delayed hits in high-throughput systems. They proposed MAD, an online solution that integrates the aggregate delay of files into existing practical caching algorithms such as LRU [13], ARC [14] and LHD [15]. However, in cloud-based applications, file sizes vary significantly, with differences up to thousands of times as observed in Google product traces [16]. Hence, the fetching costs of various files could be quite different, and the general file caching problem should be investigated. Moreover, bypassing should also be considered in cloud-based systems. The following motivating example illustrates that the existing schemes fail to tackle 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 that will be requested, with sizes $s_A = s_B = s_C = s_D = 1$, $s_L = 2$, and fetching latencies $z_A = z_B = 2$, $z_C = z_D = z_L = 1$. 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 $C, D, A, A, L, B, B, C, D$. Upon the arrival of the first request for *C*, one of A, B or L has 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 the 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 requests for *C* and *D* will also be missed, and the average latency of LRU-MAD is $11/9$. 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, *A* and *D* will be evicted since *A* does not appear in subsequent requests and *D* has lower fetching latency than *B*. The average latency of optimal is $4/9$. If bypassing is allowed, the optimal will bypass all the requests for *L* since its 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 delayed hits and bypassing. We proposed a novel framework to effectively transform *any* existing algorithm in classic file caching models, *i.e.*, 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 using 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 both the file size and fetching latency are 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 a competitive ratio of $O(ZK)$. Furthermore, the randomized version of `CaLa` is $O(Z \log K)$-competitive. To the best of our knowledge, `CaLa` is the first online algorithm with competitive ratios for the online general file caching problem with delayed hits and bypassing. Besides, we propose `CaLa+` which improves the method for estimating the weights of files during fetching in `CaLa`(in Sec. III).
- We conduct extensive simulations on Google's production trace and the Yahoo 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 8.48% without bypassing, which will be increased to 26.00% if allow bypassing. Moreover, `CaLa+` decreases the latency by more than 5% compared to `CaLa` (in Sec. IV).

[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*.

II. Problem Formulation

A. System Model

Cache System. Motivated by applications such as CDN 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 that all file sizes are integers. Naturally, the sum of sizes of files stored in the cache can never 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 cannot get future information and no assumption is made about the arrival patterns. Each request 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, this request is called a *hit* and it can be served immediately with no latency. Otherwise, it is a *miss* and has to suffer a latency to fetch this file from the remote data center; alternatively, we might forward this request to get the file from the remote data center, *i.e.*, *bypassing* the request. We set the latency to fetch f taking z_f time slots, *i.e.*, this request cannot be served until time $T + z_f$. We also set serving a request by bypassing taking z_f slots, because it also needs to interact with the remote data center similar to fetching. When fetching a file, we need to decide which files in the cache should be evicted if the cache is already out of space. Before file f is fetched and stored in the cache, all requests that require 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*. Besides, if a file f is evicted when it is still being fetched from the remote data center, the latency of every request of f during this fetching process will be increased to z_f, as these requests will be served by bypassing. Here, evicting a file during fetching means deleting its incomplete part which has already arrived in the cache and releasing the space preparing for its remaining part. We call this case, evicting during fetching, as *EDF*.

B. Problem Formulation

To formulate the online caching problem in this paper (Problem P), we first define variables in Table I below.

Optimization Goal. The goal of this problem is to minimize the total latency of all requests:

$$\min \sum_{f \in \mathcal{F}} \sum_{i} d(f, i). \tag{1}$$

[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. Thus, each file can be requested at most once in each slot.

TABLE I
Notation Summary

Notation	Definition
$\mathcal{F}$	Set of all files.
K	Cache size.
s_f	Size of file f.
z_f	Fetching latency of file f.
$x(f,t)$	State of file f at time t. (-1: fetching, 0: in cache, 1: out of cache)
$t(f,i)$	Time when file f is requested for the i-th time.
$v(f,i)$	Fetch start time if f is being fetched when the i-th request arrives.
$y(f,i)$	Indicator: 0 if the request is served normally, 1 if eviction occurs during fetching.
$d(f,i)$	Latency of the i-th request for file f.

State Definition. Variable $x(f,t)$ denotes the state of file f at time t, *i.e.*, $x(f,t) = 1$, 0, and -1 indicate that at time t, file f is out of cache, in cache, and during fetching, respectively.

$$x(f,t) \in \{-1, 0, 1\}, \quad \forall f, t. \tag{2}$$

$$x(f,0) = 1, \quad \forall f. \tag{3}$$

Cache Capacity Constraint. Constraint 4 ensures that the cache space used is always less than or equal to the total cache size K at any time.

$$\sum_{f \in \mathcal{F}} s_f \min\{1 - x(f,t), 1\} \le K, \quad \forall t. \tag{4}$$

State at Request Arrival. For concision, we use $x'(f,i)$ to represent the state of file f when the i-th request of f arrives.

$$x'(f,i) = x(f, t(f,i)), \quad \forall f, i. \tag{5}$$

Fetch Start Time. If the request arrives while the file is being fetched, $v(f,i)$ is the fetch start time.

$$v(f,i) = t(f,i-1) - z_f + d(f,i-1), \quad \text{if } x'(f,i) = -1. \tag{6}$$

EDF Indicator. If $x'(f,i) = -1$ and the i-th request of file f is an EDF, $y(f,i)$ is 1; else, $y(f,i)$ is 0.

$$y(f,i) = \begin{cases} 1, & \text{if } \prod_{k=t(f,i)+1}^{v(f,i)+z_f} (x(f,k-1) - x(f,k) + 2) = 0, \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

If the request of file f at time t is an EDF case, it implies that there must be a time slot k after t and before $v(f,i)+z_f$ such that the file was being fetched in the slot prior to k and is out of the cache at time k, *i.e.*, $x(f,k-1) = -1$, $x(f,k) = 1$, $x(f,k-1) - x(f,k) + 2 = 0$.

Latency Calculation. If a requested file is in the cache, the latency of the request is 0 (Eqn. 8). If it is out of the cache, the latency is z_f (Eqn. 9). When the requested file is being fetched, the latency depends on $y(f,i)$ (Eqn. 10). Specifically,

if $y(f,i) = 1$, it means this file will be evicted before the fetching process is completed, which causes z_f latency. By contrast, if $y(f,i) = 0$, it indicates the fetching process ends with the file arriving in the cache. In this case, the latency $d(f,i)$ is calculated as $t(f,i-1)+d(f,i-1)-t(f,i)$, which is the time when file f will arrive at the cache minus the current time.

$$d(f,i) = 0, \quad \text{if } x'(f,i) = 0, \tag{8}$$

$$d(f,i) = z_f, \quad \text{if } x'(f,i) = 1, \tag{9}$$

$$d(f,i) = \begin{cases} v(f,i) + z_f - t(f,i), \\ \qquad\qquad \text{if } x'(f,i) = -1, y(f,i) = 0, \\ z_f, \qquad\quad \text{if } x'(f,i) = -1, y(f,i) = 1. \end{cases} \tag{10}$$

C. Problem Hardness

When no bypassing is considered, Problem P has been proven to have a lower bound of the competitive ratio of $\Omega(ZK)$ [17] for deterministic algorithms. By using two kinds of request groups: pure and bursty requests similar to [17], we construct the request sequence to prove our general caching problem P has the following lower bounds for deterministic and randomized solutions.

Theorem 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: We define two kinds of request groups, *pure* and *bursty* requests, similar to [17]. 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 a total of $K+1$ 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, each bursty request is missed. For each bursty request, the latency caused by it is at least $\frac{Z(Z+1)}{2}$. So the total latency of $\mathcal{A}$ is $Z + K\frac{Z(Z+1)}{2}$.

By contrast, the optimal offline algorithm always keeps the files that will be requested in the following K bursty requests in cache after r_{K+1}^p, which causes these K bursty requests all hit. So the latency of optimal is only Z caused by r_{K+1}^p.

Thus, for deterministic algorithms, the lower bound of file caching with delayed hits is $\Omega(ZK)$.

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 the cache. Since there is only one file not in the cache, we have the following equation: $\sum_i p_i = 1$. 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 the 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 $Z + \frac{Z(Z+1)}{2}H_K$, 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 and we assume this pure request does not affect files' markers. Let u be the number of unmarked files. According to the definition of subphase, u is different at the beginning of each subphase in a phase. Besides, let $\mathcal{M}$ be the set of marked files. And we define: $P = \sum_{f_i \in \mathcal{M}} p_i$. For each subphase, we prove that its expected latency is at least $\frac{Z(Z+1)}{2u}$ as follows:

- If $P = 0$, there must be an unmarked file f_i with $p_i \geq 1/u$. Let this subphase contain a single request to r_i^b. The probability that r_i^b misses is at least $1/u$ and the latency that will be caused if r_i^b misses is $\frac{Z(Z+1)}{2}$. So the expected latency of this request is at least $\frac{Z(Z+1)}{2u}$.
- If $P > 0$, then continuously require r_i^b until the total expected latency of this subphase exceeding $\frac{Z(Z+1)}{2u}$, where $f_i \in \mathcal{M}$ and $p_i > 0$. This works because this subphase can only end with a request of an unmarked file, while f_i is a marked file. No matter how many r_i^b we require, these bursty requests always happen in the current subphase.

Finally, we accumulate the total latency of all subphases in this phase. Naturally, u takes all the integers between 1 and K, thus, the total latency of $\mathcal{A}$ in this phase is

$$Z + \sum_{1 \leq u \leq K} \frac{Z(Z+1)}{2u} = Z + \frac{Z(Z+1)}{2} H_K. \tag{11}$$

By contrast, as each phase contains requests for exactly K different files, the optimal offline algorithm can always keep these K files in the cache, which causes the total latency of optimal is only Z caused by r_{K+1}^p.

Thus, for randomized algorithms, the lower bound of file caching with delayed hits is $\Omega(Z \log K)$. □

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`

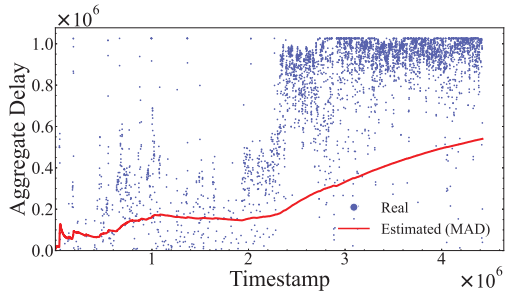


Fig. 3. The estimated aggregate delay [7] *vs.* real aggregate delay calculated by Eqn. 12.

(Algorithm III-B) in detail in Sec. III-B. We also analyze the performance of CaLa in Sec. III-C, and prove that the deterministic and the randomized version of CaLa is $O(ZK)$-competitive and $O(Z \log K)$-competitive, respectively.

A. Estimated Weight

Aggregate Delay-Based Weight. The central challenge of this problem is how to deal with delayed hits. In the design of MAD [7], it uses the aggregate delay to capture the total latency caused by a file's miss:

$$\text{AggDelay}(f,T) = z_f + \sum_{1\le\tau\le z_f-1} (z_f-\tau)\eta(f,T+\tau), \tag{12}$$

where z_f is the fetching latency of file f, $\eta(f,T+\tau)=1$ if f is requested at time $T+\tau$; else, $\eta(f,T+\tau)=0$. The aggregate delay of file f cannot be directly calculated in practice since it requires the future information of the next z_f time slots. MAD uses the average aggregate delay(AAD) of all the past requests for f to estimate the aggregate delay of the next request for f.

Conservative Upper-Bound Weighting. 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.

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 $O(Z)\times$ extra cost.

Combined Weighting Approach. 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 of using the upper bound is conservative. To get a trade-off between these two kinds of methods, CaLaintroduces a hybrid weight function to represent the weight of each file:

$$W_f(T) = (1-\gamma)\,\text{AggDelay}(f,T) + \gamma z_f^2, \tag{13}$$

where parameter γ is called *conservative parameter* and is a tuning parameter that balances:

- Aggressive estimation ($\gamma = 0$), which relies on past observed delays.
- Conservative estimation ($\gamma = 1$), which assumes worst-case latency.

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

```
1 Input Parameter γ, file f, time T
2 if f is not in the cache then
3     f.cumulativeDelay ← f.cumulativeDelay + z_f;
4     f.fetchingTime ← T;
5     f.numFetching ← f.numFetching + 1;
6 if f.state = OCCUPY then
7     f.cumulativeDelay ←
      f.cumulativeDelay + (z_f − (T − f.fetchingTime));
8 f.averageAggDelay ← f.cumulativeDelay / f.numFetching;
9 return (1 − γ) · f.averageAggDelay + γ · z_f^2;
```

First, we introduce the algorithm to update estimated weights (Algorithm 1). This method mainly adopted Algorithm 1 in [7]. 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 $\mathcal{A}$ chooses to evict files $\mathcal{F}_{\text{evict}}$ in the cache to make room for file f, then CaLa will conduct the decision by evicting the files immediately (Line 14 to Line 17) and storing the new file f (Line 18 to Line 20). If $\mathcal{A}$ chooses to bypass this request, then CaLa also bypasses it (Line 25). When a file finishes its fetching, we serve all the buffered requests together (Line 6 to Line 10).

We use a modified version of Landlord, *i.e.*, Landlord with bypassing (LLB) [18], as the kernel of CaLa. Landlord [10] is an $O(K)$-competitive online algorithm for the 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

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;
  $T \leftarrow T + 1$;

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 $\mathcal{A}(z_i^2) \le c \cdot \text{OPT}'(z_i^2)$.

Lemma 1: $\text{ALG}(z_i) \le \mathcal{A}(z_i^2)$.

Proof: We define the 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 contains a single fetching of f_i followed by zero or more delayed hits of f_i. For special cases of ALG, if f_i is evicted during fetching, let all requests to f_i in the period from starting fetching to evicting be a fetching group. Since each fetching group of f_i at most includes z_i requests of f_i and the latency of each request to f_i is at most z_i, the fetching latency of each fetching group of f_i of $\text{ALG}(z_i)$ is no larger than z_i^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 either in a fetching group or a bypassing. Since CaLa follows the operations of $\mathcal{A}$, the fetching group of $\text{ALG}(z_i)$ is exactly the same as $\mathcal{A}(z_i^2)$. Thus, $\text{ALG}(z_i) \le \mathcal{A}(z_i^2)$. □

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 $(\beta w_1, \beta w_2, \ldots, \beta w_n)$. Then we have $\text{OPT}'(\mathcal{I}_2) = \beta \text{OPT}'(\mathcal{I}_1)$.

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 $w_1 \le w'_1, w_2 \le w'_2, \ldots, w_n \le w'_n$. Then we have $\text{OPT}'(\mathcal{I}_1) \le \text{OPT}'(\mathcal{I}_2)$.

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

Lemma 2: $\text{OPT}'(z_i^2) \le Z \cdot \text{OPT}'(z_i)$.

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

Lemma 3: $\text{OPT}'(z_i) \le \text{OPT}(z_i)$.

Proof: We define the *n*-th fetching window of f_i as the time interval during which f_i is being fetched from the remote data center for the *n*-th time, which begins from the time when f_i is requested and not found in the cache for the *n*-th time, denoted as $t^b(i,n)$, and ends at $\min\{t^f(i,n), t^e(i,n)\}$. Here, $t^f(i,n) = t^b(i,n) + z_i$ and $t^e(i,n)$ is the time when f_i is evicted from cache for the *n*-th time. We assume the *j*-th request of f_i appears in the $N_{i,j}$-th fetching window of f_i. For conciseness, let $t^f_{i,j}$ represent $t^f(i, N_{i,j})$ and $t^e_{i,j}$ represent $t^e(i, N_{i,j})$. We set $\mathcal{B}$ as an algorithm that meets the following conditions: Firstly, $\mathcal{B}$ and OPT have the same fetching groups, which means they have the same main structures including content admission and eviction decision. Secondly, there is only one difference between $\mathcal{B}$ and OPT, which is the way how they compute the total latency. For the *j*-th request of file f_i, we assume the time slot when this request arrives is $T_{i,j}$. Besides, we define the increase of the total latency caused by the *j*-th request of file f_i in algorithm OPT be $\text{OPT}_{i,j}$ and define the increase of the total latency caused by this request in algorithm $\mathcal{B}$ be $\mathcal{B}_{i,j}$. If the *j*-th request of f_i hits, $\mathcal{B}_{i,j} = \text{OPT}_{i,j} = 0$. If the *j*-th request of f_i missed, $\mathcal{B}_{i,j} = \text{OPT}_{i,j} = z_i$. If the *j*-th request of f_i delayed hits, $\mathcal{B}_{i,j}$ is always zero. But $\text{OPT}_{i,j}$ is z_i when $t^e_{i,j} < t^f_{i,j}$ which means it is an EDF case and $\text{OPT}_{i,j}$ is $t^f_{i,j} - T_{i,j}$ when $t^e_{i,j} > t^f_{i,j}$ which means this fetching completes normally. Thus, $\mathcal{B}_{i,j}$ is always less than or equal to $\text{OPT}_{i,j}$, for any *i*, *j*. We define $\mathcal{B}(z_i)$ as the total latency of algorithm $\mathcal{B}$. According to the above definition of $\mathcal{B}$, $\mathcal{B}(z_i) \le \text{OPT}(z_i)$. Besides, we find $\mathcal{B}$ computes the total latency in the same way as algorithms in the model without delayed hits. The model with delayed hits and the model without delayed hits only differ in the method of how they compute the total latency. So the solution of $\mathcal{B}$ is

a feasible solution in the model without delayed hits. Then, as $\mathrm{OPT}'(z_i)$ is the total latency of the offline optimal solution of general file caching with bypassing and without delayed hits, $\mathrm{OPT}'(z_i) \leq \mathcal{B}(z_i)$. Thus, $\mathrm{OPT}'(z_i) \leq \mathrm{OPT}(z_i)$. □

By combining Lemma 1, Lemma 2 and Lemma 3 together, we have the following theorem.

Theorem 2: If there is an online file caching algorithm $\mathcal{A}$ with bypassing that is c-competitive, `CaLa` is $O(Zc)$-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 [18], we have the following corollary.

Corollary 1: By setting $\gamma = 1$, the deterministic version of `CaLa` is $O(ZK)$-competitive, and the randomized version of `CaLa` is $O(Z \log K)$-competitive.

D. `CaLa+`

`CaLa+`improves weight estimation by incorporating EDF. When a request for file f is a miss, delayed hits will be introduced potentially during f's fetching process. Additionally, if the cache is full at the time of this request, the latency of a file f' evicted to make space for f will be increased to $z_{f'}$, which is also the consequence of this miss. To address the above, provided that this file begins fetching at $T_s(f,T)$, we define the EDF cost as:

$$\mathrm{C_{EDF}}(f,T) = z_f N(f,T) - \mathrm{ADR}(f,T), \tag{14}$$

where $N(f,T)$ is the number of requests for f during this fetching, $\mathrm{ADR}(f,T)$ is the aggregate delay of f during the period from $T_s(f,T)$ to T, given by:

$$\mathrm{ADR}(f,T) = \sum_{0 \leq \tau \leq T - T_s(f,T)} (z_f - \tau)\eta(f, T_s(f,T)+\tau). \tag{15}$$

Thus, `CaLa+`modifies the weight formula during the fetching process to:

$$W_f(T) = (1-\gamma)\,\mathrm{AggDelay}(f,T) + \gamma z_f^2 + \alpha\,\mathrm{C_{EDF}}(f,T), \tag{16}$$

where α is a tuning parameter to control the impact of EDF.

Modified Weighting Algorithm. According to Eqn. 16 and Eqn. 14, we have a new algorithm to update the estimated weights (Algorithm 3 as follows. On the basis of Algorithm 1, Algorithm 3 incorporates $\mathrm{C_{EDF}}$ into the weight calculation. If f is not in the cache when a new request for f arrives, we start a new fetching period just like Algorithm 1 (Line 3 to Line 4). Meanwhile, we initialize the number of requests for f in this fetching period to 1 and set its aggregate latency in this fetching period as z_f (Line 5 to Line 6). If the status of f is OCCUPY, which means f is already in a fetching period, we accumulate the latency of the delayed hit to this fetching and increase the request count for this period (Line 10 to Line 11).

Specifically, Algorithm 3 changes the way we calculate historical average AggDelay at time T, which is called AAD in

Algorithm 3 EstimatedWeight+

```
1  Input Parameter γ, α, file f, time T
2  if f is not in the cache then
3      f.fetchingTime ← T;
4      f.numFetching ← f.numFetching + 1;
5      f.aggDelay ← z_f;
6      f.aggNum ← 1;
7      f.lastAverageAggDelay ← f.averageAggDelay;
8      f.averageAggDelay ← f.lastAverageAggDelay +
       (f.aggDelay − f.lastAverageAggDelay) / f.numFetching;
9  if f.state = OCCUPY then
10     f.aggDelay ←
       f.aggDelay + (z_f − (T − f.fetchingTime));
11     f.aggNum ← f.aggNum + 1;
12     f.averageAggDelay ← f.lastAverageAggDelay +
       (f.aggDelay − f.lastAverageAggDelay) / f.numFetching;
13 if f.state = IN then
14     return (1 − γ) · f.averageAggDelay + γ · z_f^2;
15 else
16     return (1 − γ) · f.averageAggDelay + γ · z_f^2 +
17     α · (f.aggNum · z_f − f.aggDelay);
```

Eqn. 17. Instead of dividing accumulative delay by the number of fetches, the new method updates AAD using the following approach (Line 7 to Line 8, Line 12):

$$\mathrm{AAD}_n^f = \mathrm{AAD}_{n-1}^f + \frac{1}{n}(\mathrm{ADR}(f,T) - \mathrm{AAD}_{n-1}^f), \tag{17}$$

where n is the number of times how much file f is fetched.

Then, if the status of file f is IN, its estimated weight will be calculated using Eqn. 13 (Line 14); otherwise, its estimated weight is calculated according to Eqn. 16 (Line 17).

Finally, in Algorithm 2, we change the function which estimates weights of files from $EstimatedWeight(f,T)$ to $EstimatedWeight + (f,T)$ (in Line 12). This updated algorithm is named `CaLa+`.

E. Summary of Weighting Methods

Overall, the weight of a file quantifies its importance by estimating the impact of its miss on total latency, serving as a priority metric for decisions on fetching, evicting, or bypassing. By using weights, we can leverage existing weight-based algorithms and integrate diverse methods.

Table II summarizes the weighting approaches discussed above. We use AggDelay and $\mathrm{C_{EDF}}$ to quantify the effects of delayed hits and EDFs, improving the accuracy of latency loss estimation due to file misses.

IV. Evaluation

We evaluate the performance of `CaLa` on two datasets: (1) the production trace from Google [16], (2) the system benchmark of YCSB workloads from Yahoo [19], which is used widely in previous works (*e.g.*, [4], [20], [21]). We compare

TABLE II
COMPARISON OF DIFFERENT WEIGHTING METHODS

Weight Type	Formula	Pros	Cons
Aggregate Delay	$\text{AggDelay}(f,T)$	Captures historical delayed hits	Future requests deviates from history
Conservative Bound	z_f^2	Guarantees worst-case performance	Overestimates impact
Hybrid (CaLa)	$(1-\gamma)\text{AggDelay}+\gamma z_f^2$	Balances estimation	Needs tuning of γ
EDF-Aware (CaLa+)	$(1-\gamma)\text{AggDelay}+\gamma z_f^2+\alpha\,\text{C}_{\text{EDF}}$	Reduces EDF loss	Requires tuning of α

CaLa and CaLa+ with several state-of-the-art methods, *i.e.*, LRU [2], LRU-MAD [7], Landlord [10], and Landlord with bypassing [18]. The details of experiment results are shown in Sec. IV-C and we highlight our key findings as follows.

- Compared to LRU-MAD, the state-of-the-art algorithm deals with delayed hits, CaLa can reduce latency by up to 8.48% without bypassing. This reduction will be increased to 26.00% if bypassing is allowed. Additionally, the latency improvement from CaLa to CaLa+ is more than 5%.
- CaLa 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.
- If the cache size is relatively small (*e.g.*, sum of 0.1% to 0.5% of the active files), when bypassing is allowed, CaLa and CaLa+ outperform other algorithms significantly.

A. Methodology

We set the cache size in a way similar to [7], 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 the top 1% of active files. For CaLa, the default value of γ is set to 0.1. For CaLa+, the default value of α is set to 10.

Workloads. There are $4.4M$ and $2.8M$ requests in the Google's production trace and the YCSB benchmark, respectively. Note that the request sequence patterns of these two traces are quite different. The requests in Google's trace for the same file usually arrive continuously, while the requests in the YCSB benchmark arrive individually. To express this more clearly, we define the *request locality* of a sequence, calculated as the ratio of the number of requests that are followed by the ones requiring the same file among the number of total requests. The request locality of the Google trace and the YCSB benchmark are 0.7058 and 0.0025, respectively. This is the main reason why CaLaand CaLa+ perform better on the Google trace than the YCSB benchmark. For Google's production trace, we use "RAM Used" as the size of the file. The size of the file in the 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 $35K$ [22]. The average default latency of files is set to $100ms$ (*i.e.*, the average value of z_f for files is 1000), which is the approximate latency to fetch files from remote data center [6]. Since both traces lack information of the file's fetching latency, we randomly generate a latency uniformly distributed within $(0, Z_{\text{upper}})$ 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 by 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 better performance.

B. Baseline Algorithms

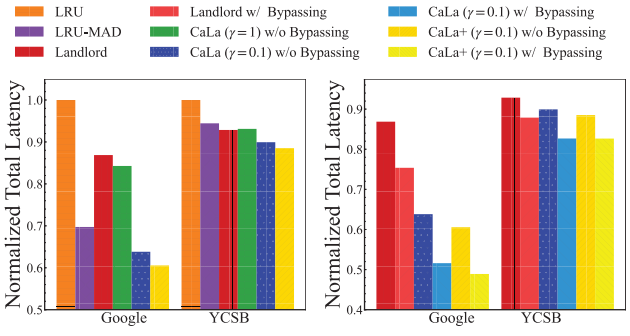
We compare the performance of our proposed algorithms CaLa ($\gamma = 1$) without bypassing, CaLa without bypassing, CaLa with bypassing, CaLa+ without bypassing and CaLa+ with bypassing with the following baselines. Our proposed algorithms without bypassing use Landlord as their kernel, while algorithms with bypassing use LLB.

LRU [2]. 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 [7]. LRU-MAD is the state-of-the-art caching algorithm that deals with delayed hits by calculating each file's rank. The rank of a file is its aggregate delay 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 [7] all the files have the same fetching latency, LRU-MAD is aware of heterogeneous fetching latency because of the calculation of aggregate delay.

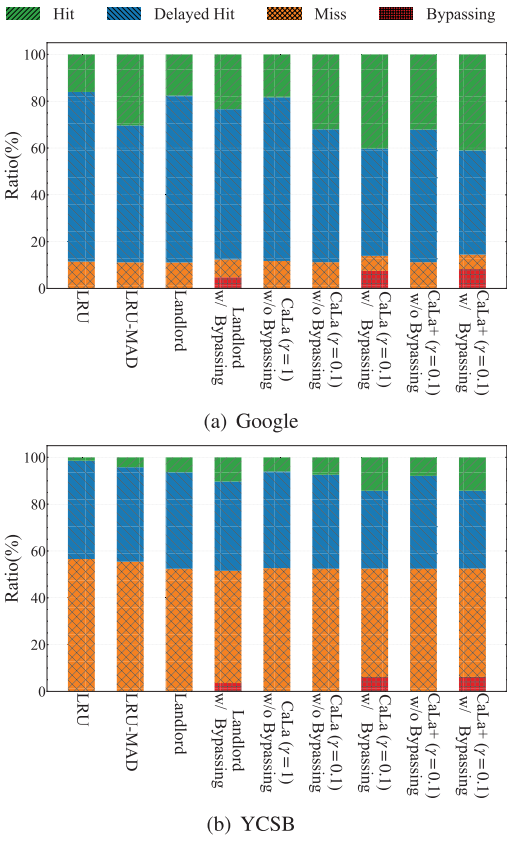
Landlord [10]. Landlord is a $O(K)$-competitive algorithm for online general file caching. The core of Landlord is to maintain a credit for each file and evict all the zero-credit files. Each file's credit is set to its cost (*i.e.*, fetching latency in this paper) when it is requested. Credit for all files in the cache will be decreased by a value proportional to the size of the file.

Landlord with Bypassing [18]. 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.



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

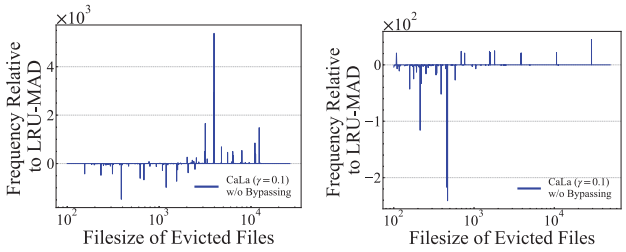
Fig. 4. Overall performance.



(a) Google

(b) YCSB

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



(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.

C. Experiment Results

Overall Result. We first evaluate the overall performance of CaLa ($\gamma = 1$) without bypassing, CaLa without bypassing and with bypassing, CaLa+ without bypassing and with bypassing and we 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. In Google's trace, the latency improvements of CaLa to LRU, LRU-MAD and Landlord are 36.18%, 8.48% and 26.53%, respectively. And the latency improvement of CaLa+ to CaLa is 5.11%. For the results in the YCSB benchmark, the latency improvements of CaLa to LRU, LRU-MAD and Landlord are 10.06%, 4.77% and 3.15%, respectively. The latency improvement of CaLa+ to CaLa is 3.42%. 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 19.14% and 8.07% latency on Google's trace and the YCSB benchmark, respectively. Bypassing can improve the performance of algorithms because it provides a mechanism to keep the files with low quotes out of the cache, which is similar to q-LRU and k-LRU [23]. We can see that the latency improvement of CaLa+ with bypassing to CaLa with bypassing is 5.21% on Google's trace but it is not obvious on the YCSB benchmark, because the requests of the YCSB benchmark are not bursty. We also find the performance of LRU-MAD is better than Landlord in Google's trace, while the opposite result is shown in the 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.

Ingredient of Latency. To explore the factors affecting performance, we analyze the ratios of hits, delayed hits, misses and bypasses for each algorithm in Fig. 5. Firstly, we find that the hit ratio generally determines an algorithm's performance. In the result from Google's trace, LRU-MAD has the highest hit ratio among algorithms without bypassing. CaLa also gets a hit ratio close to LRU-MAD, but performs better due to its tendency to evict larger files and allocate more space for high-latency files. Allowing bypassing further improves hit rates and performance. This phenomenon is more obvious in the YCSB benchmark, which has more requests for infrequent files that cause inevitable misses. By bypassing some 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. Besides, the hit ratio of CaLa+ is close to CaLawhether with or without bypassing.

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, while 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 that are evicted. The value on the y-axis represents the number of files evicted by CaLa compared with LRU-MAD. It shows that

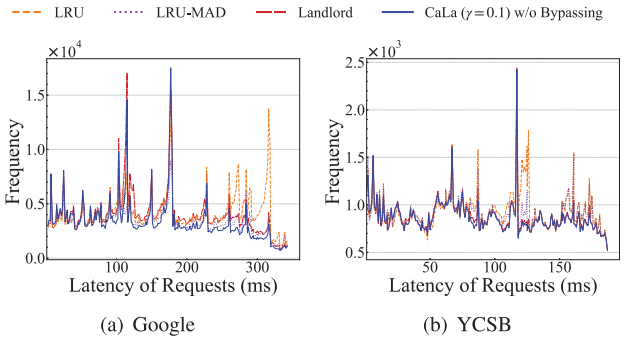


Fig. 7. Distribution of the latency of requests.

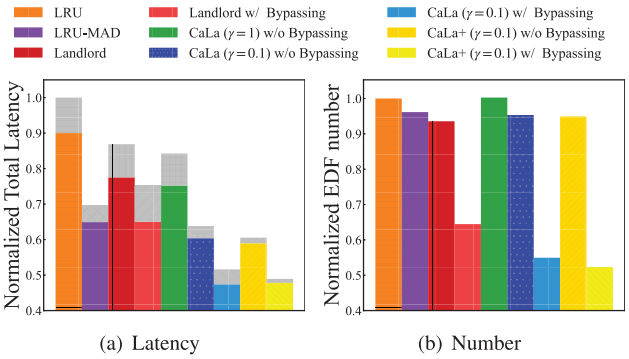


Fig. 8. The latency caused by EDFs and the number of EDFs.

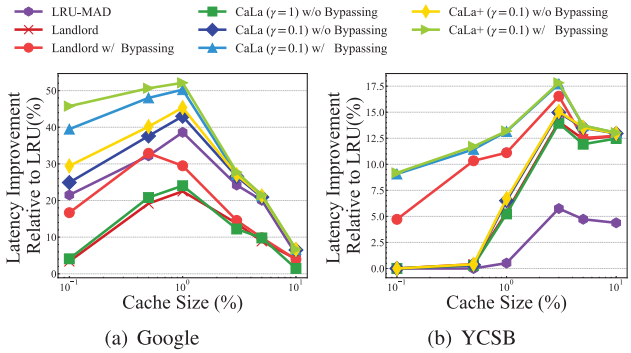


Fig. 9. Impact of cache size.

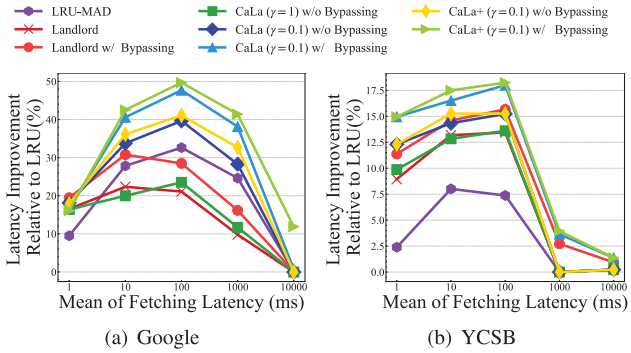


Fig. 10. Impact of fetching latency.

CaLa evicts more large files, making more space left for files with high frequency and high latency.

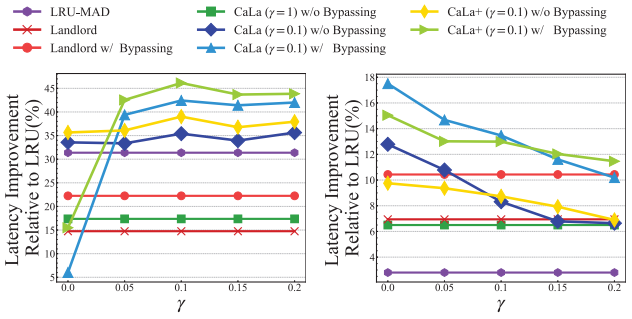
Latency of Requests. We plot the distribution of latency of requests in Fig. 7, including latency caused by bypassing, misses and delayed hits, where the height of a point for a specific latency represents the number of requests served at this latency. In the Google's trace, LRU has more high latency requests, resulting in its overall poor performance. With the help of aggregate delay, LRU-MAD is much better at reducing 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.

EDF Latency. Fig. 8 illustrates the differences in the latency caused by EDFs among the algorithms, focusing on Google's trace since the latency improvement of CaLa+ compared to CaLa is less obvious in the YCSB benchmark. In Fig. 8(a), the total height of every bar represents the overall latency, and the grey portion of every bar indicates the latency increase caused by all EDFs of these algorithms, referred to as EDF latency. We observe that CaLa+ has lower EDF latency than CaLa, explaining its total latency improvement of 5.11%, despite a less noticeable improvement in hit ratio. Specifically, Fig. 8(b) shows that bypassing effectively reduces the number of EDFs. Bypassing provides a new option when algorithms without bypassing have to evict files during fetching to serve new requests. Based on Figs. 8(a) and 8(a), we can conclude that the average latency of every EDF in CaLa+ without bypassing is less than CaLa without bypassing, indicating that CaLa+ can select files that incur less EDF latency to evict. This is because CaLa+ considers the EDF cost of evicting this file in its credit during fetching additionally, while CaLa only considers potential aggregate delay in the future.

D. Sensitivity Study

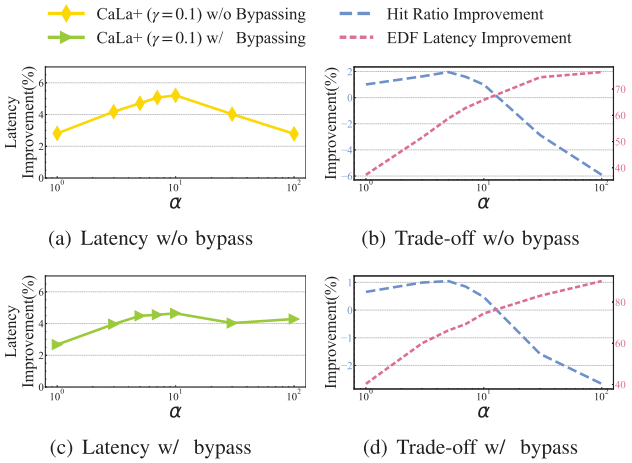
Impact of Cache Size. To investigate the impact of cache size, we change the cache size from 0.1% to 10% and measured the latency improvement of the algorithms relative to LRU, as shown in Fig. 9. First, when the cache size is small (*e.g.*, sum of 0.1% to 0.5% of the active files), CaLa with bypassing performs far beyond other algorithms. This is because bypassing can avoid evicting some frequently requested files and reduce misses and delayed hits. As the cache size gradually increases, the performance of CaLa approaches that of CaLa with bypassing. Notably, 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 frequent files can be cached and the performance of all algorithms tends to be the same.

Impact of Fetching Latency. We present the impact of fetching latency in Fig. 10, where the fetching latency changes from 10 to 100000 time slots. The performance of LRU-MAD, CaLa and CaLa+ starts increasing when the fetching latency becomes higher since their awareness of latency and delayed hits. In the YCSB benchmark, these algorithms perform sim-



(a) Google (b) YCSB

Fig. 11. Impact of γ.



(a) Latency w/o bypass (b) Trade-off w/o bypass

(c) Latency w/ bypass (d) Trade-off w/ bypass

Fig. 12. Impact of α (CaLa+ Relative to CaLa).

ilarly to the algorithms without delayed hits, as 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 result in misses or delayed hits. The fluctuations in Fig. 10 reflect the different sensitivity of various algorithms to the fetching latency.

Impact of γ. As shown in Fig. 11, for the Google's trace, the best performance is achieved when $\gamma = 0.1$, which shows that it is better to use a value of γ closer to the aggregate delay for burst requests. In the YCSB benchmark, the best performance can be obtained by setting $\gamma = 0$. However, CaLa with bypassing and CaLa+ with bypassing perform extremely badly when $\gamma = 0$ in the Google's trace, which indicates the misestimation of aggregate delay will seriously affects the results of bypassing, especially for burst requests.

Impact of α. Fig. 12 illustrates the impact of α on CaLa+latency. Fig. 12(a) and 12(c) show overall latency changes, while Fig. 12(b) and 12(d) provide insights into the underlying causes. The results show that CaLa+achieves optimal performance at $\alpha = 10$, whether with bypassing or without bypassing. As α changes from 1 to 100, we observe that the latency improvement of CaLa+relative to CaLainitially increases and then gradually declines, whether with bypassing or without bypassing. To explain this phenomenon, we analyze the changes in EDF latency and hit ratio of CaLa+ without bypassing and CaLa+ with bypassing under different α values (respectively relative to CaLa without bypassing and CaLa with bypassing). The results reveal a trade-off between reducing the EDF latency and maintaining the hit ratio, explaining the observed performance trend.

V. Related Works

Theoretical Results of Caching. The first systematic study of the performance analysis of caching algorithm is presented by Sleator and Tarjan [2], 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 [8] 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 [24] proposed a general method to transfer this problem to the uniform setting and gave an online algorithm with $O(\log^2 k)$-competitive when the fetch cost of a file equals 1 or its size. The tight deterministic k-competitive algorithm for weighted caching came from the results of the k-server problem on trees due to Chrobak et al. [25]. Bansal et al. [26] designed the first randomized O(log k)-competitive online algorithm for this problem. Jiang et al. [27] studied the weighted paging problem and gave a lower bound of $\Omega(\log k)$ in the PRP model. Then they proposed a stronger model called SPRP and gave an algorithm with 2-competitive. For general caching with nonuniform file size and fetch cost, Irani [28] proved the offline version is already NP-hard. Bar-Noy et al. [29] gave a 4-approximate algorithm for the offline version and Adamaszek et al. [30] showed a tight online algorithm with $O(\log k)$-competitive. Tan et al. [4] studied the caching variant in edge computing, where the system contains multiple caches and requests can be relayed to other caches, and gave an $O(\log k)$-competitive online algorithm. Lykouris and Vassilvitskii [31] 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, Rohatgi [32] improved the theoretical result to $O(1 + \min((\eta/OPT)/k, 1) \log k)$ and provided a lower bound of $\Omega(\log \min((\eta/OPT)/(k \log k), k))$. Los et al. [33] provided an $O(l)$-competitive deterministic and an $O(\log l)$-competitive randomized algorithm for a semi-online model of weighted paging, where l is the number of distinct weight classes.

Caching Algorithms in CDNs. Some works explore valuable features to optimize cache performance based on the actual production environment for CDNs. Hu et al. [34] used data locality to minimize the average response time of key-value caches. Beckmann et al. [15] 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. [35] proposed AdaptSize, an adaptive, size-aware cache admission policy for hot object cache in CDN. Berg et al. [36] showed CacheLib, a general-purpose caching engine, extracts a core set of common requirements and functionality from otherwise disjoint caching systems. Ye et al. [37] proposed a

TABLE III
THEORETICAL RESULTS OF CACHING PROBLEMS

Algorithms	File Size & Fetch Cost	Delayed Hits	Bypassing	Performance Guarantee	Type*
LRU [2]	Uniform	✘	✘	$O(K)$	D
Marking [8]	Uniform	✘	✘	$O(\log K)$	R
Landlord [10]	Non-uniform	✘	✘	$O(K)$	D
Adamaszek *et al.* [30]	Non-uniform	✘	✘	$O(\log K)$	R
Camul-det, Camul [4]	Uniform	✘	✔	$O(K)$ $O(\log K)$	D R
LLB [18]	Non-uniform	✘	✔	$O(K)$	D
MAD [7]	Uniform	✔	✘	-	D
CaLa [this work]	Non-uniform	✔	✔	$O(ZK)$ $O(Z\log K)$	D R

* D: Deterministic Algorithm, R: Randomized Algorithm.

learning framework to learn the joint cache size scaling and strategy adaptation policy for Elastic CDN. Song et al. [38] proposed HALP that achieves low CPU overhead and robust byte miss ratio improvement by augmenting a heuristic policy with machine learning. Chen et al. [39] developed Darwin, a learning-based CDN caching approach, to flexibly optimize different caching objectives. Zong et al. [40] 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. [41] proposed LRB to mimic the relaxed Belady's MIN algorithm by using Gradient Boosting Machines [42]. Akhtar et al. [43] described AviC that leverages properties of video delivery to design the eviction policy in CDN. Zhou et al. [44] introduced Bounded Linear Probing (BLP) balancing hit rate and lookup latency for network appliances. Yang et al. [45] presented C2DN to eliminate the miss ratio spikes caused by server unavailabilities. Garetto et al. [46] provided a first comprehensive analysis of similarity caching in different settings. Elsayed et al. [47] proposed a machine learning approach to estimate the optimal TTL values for large systems. Jin et al. [48] presented NetCache, a key-value store architecture that balances the load across storage nodes.

Caching Algorithms considering delayed hits. Atre et al. [7] studied the caching problem with delayed hits and proposed a heuristic to estimate the aggregate latency caused by a cache miss. Paper [17] introduced the lower bound of caching with delayed hits for deterministic solutions. Yan and Li [49] proposed a timer-based model considering delayed hits and a lightweight latency-aware caching algorithm named LA-Cache. Besides, the fetching delay of a caching file has also been considered on Time-to-Live (TTL) caches, known as non-zero download delay (non-ZDD) models [50], [51], [52], [53].

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 III.

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 several promising directions to more accurately estimate 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, learning-based methods such as multi-armed bandits and deep reinforcement learning 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 $O(Z)\times$ cost. This transformation is proven to be asymptotically 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, respectively. But the competitive ratio of CaLa and the lower bounds may still not reflect the performance in practice, since the parameter chosen 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 potential parameters that can better describe the request sequence.

Relating to TTL Caches. TTL caches set an independent expiration time for each individual content, such that we can decouple the eviction mechanisms of different files. Thus, TTL-based cache strategies typically have low time complexity and are simple to deploy. But few works about TTL caches provide competitive analysis. We believe that our method of estimating the weights of files and doing competitive analysis can also provide some insights for TTL caching, and applying the algorithm principles of this work to TTL caching to optimize non-ZDD performance is an interesting future work.

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 $O(ZK)$ and $O(Z\log K)$, respectively. Besides, we propose CaLa+ which takes evicting during fetching into account when estimating the weights of

files. We evaluate `CaLa` based on Google's trace and the YCSB benchmark. The experiment results show that compared with LRU-MAD, `CaLa` can reduce the latency by up to 8.48% without bypassing. Furthermore, this reduction will be increased to 26.00% if bypassing is allowed. And `CaLa+` can further reduce latency by about 5% compared to `CaLa`.

REFERENCES

[1] C. Zhang, H. Tan, G. Li, Z. Han, S. H.-C. Jiang, and X.-Y. Li, "Online file caching in latency-sensitive systems with delayed hits and bypassing," in *Proc. IEEE Conf. Comput. Commun. (IEEE INFOCOM)*, May 2022, pp. 1059–1068.

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

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

[4] 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 *Proc. IEEE Conf. Comput. Commun. (IEEE INFOCOM)*, Apr. 2019, pp. 244–252.

[5] R. Krishnan et al., "Moving beyond end-to-end path information to optimize CDN performance," in *Proc. 9th ACM SIGCOMM Conf. Internet Meas.*, Nov. 2009, pp. 190–201.

[6] X. Fan, E. Katz-Bassett, and J. Heidemann, "Assessing affinity between users and CDN sites," in *Proc. Int. Workshop Traffic Monitor. Anal.* Cham, Switzerland: Springer, Jan. 2015, pp. 95–110.

[7] N. Atre, J. Sherry, W. Wang, and D. S. Berger, "Caching with delayed hits," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, Jul. 2020, pp. 495–513.

[8] 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, Dec. 1991.

[9] D. Achlioptas, M. Chrobak, and J. Noga, "Competitive analysis of randomized paging algorithms," *Theor. Comput. Sci.*, vol. 234, nos. 1–2, pp. 203–218, Mar. 2000.

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

[11] N. Bansal, N. Buchbinder, and J. Naor, "Randomized competitive algorithms for generalized caching," *SIAM J. Comput.*, vol. 41, no. 2, pp. 391–414, Jan. 2012.

[12] D. Genbrugge and L. Eeckhout, "Memory data flow modeling in statistical simulation for the efficient exploration of microprocessor design spaces," *IEEE Trans. Comput.*, vol. 57, no. 1, pp. 41–54, Jan. 2008.

[13] M. V. Wilkes, "Slave memories and dynamic storage allocation," *IEEE Trans. Electron. Comput.*, vol. EC-14, no. 2, pp. 270–271, Apr. 1965.

[14] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. USENI FAST*, Mar. 2003, pp. 115–130.

[15] N. Beckmann, H. Chen, and A. Cidon, "LHD: Improving cache hit rate by maximizing hit density," in *Proc. USENIX NSDI*, Jan. 2018, pp. 389–403.

[16] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: Format + schemam," Google, Mountain View, CA, USA, Tech. Rep., Nov. 2011. [Online]. Available: https://github.com/google/cluster-data

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

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

[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, Jun. 2010, pp. 143–154.

[20] 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 *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2020, pp. 1077–1091.

[21] T. Yao et al., "MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM," in *Proc. USENIX ATC*, Jan. 2020, pp. 17–31.

[22] P. Wendell and M. J. Freedman, "Going viral: Flash crowds in an open CDN," in *Proc. ACM SIGCOMM Conf. Internet Meas. Conf.*, Nov. 2011, pp. 549–558.

[23] V. Martina, M. Garetto, and E. Leonardi, "A unified approach to the performance analysis of caching systems," in *Proc. IEEE Conf. Comput. Commun. (IEEE INFOCOM)*, Apr. 2014, pp. 2040–2048.

[24] S. Irani, "Page replacement with multi-size pages and applications to web caching," in *Proc. 29th Annu. ACM Symp. Theory Comput. (STOC)*, 1997, pp. 701–710.

[25] M. Chrobak, H. Karloof, T. Payne, and S. Vishwnathan, "New ressults on server problems," *SIAM J. Discrete Math.*, vol. 4, no. 2, pp. 172–181, May 1991.

[26] N. Bansal, N. Buchbinder, and J. Naor, "A primal-dual randomized algorithm for weighted paging," *J. ACM (JACM)*, vol. 59, no. 4, pp. 1–24, Aug. 2012.

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

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

[29] A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. Naor, and B. Schieber, "A unified approach to approximating resource allocation and scheduling," *J. ACM.*, vol. 48, pp. 1069–1090, Sep. 2001.

[30] A. Adamaszek, A. Czumaj, M. Englert, and H. Räcke, "An O(log k)-competitive algorithm for generalized caching," in *Proc. ACM Trans. Algorithms (TALG)*, Jan. 2012, vol. 15, no. 1, pp. 1–18.

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

[32] D. Rohatgi, "Near-optimal bounds for online caching with machine learned advice," in *Proc. SIAM SODA*, Dec. 2019, pp. 1834–1845.

[33] D. Los, T. Sauerwald, and J. Sylvester, "Balanced allocations with heterogeneous bins: The power of memory," in *Proc. SIAM SODA*, Jan. 2023, pp. 4448–4477.

[34] X. Hu et al., "LAMA: Optimized locality-aware memory allocation for key-value cache," in *Proc. USENIX ATC*, Jul. 2015, pp. 57–69.

[35] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter, "Adaptsize: Orchestrating the hot object memory cache in a content delivery network," in *Proc. USENIX NSDI*, Mar. 2017, pp. 483–498.

[36] B. Berg et al., "The CacheLib caching engine: Design and experiences at scale," in *Proc. USENIX OSDI*, Jan. 2020, pp. 753–768.

[37] J. Ye, Z. Li, Z. Wang, Z. Zheng, H. Hu, and W. Zhu, "Joint cache size scaling and replacement adaptation for small content providers," in *Proc. IEEE Conf. Comput. Commun. (IEEE INFOCOM)*, May 2021, pp. 1–10.

[38] Z. Song et al., "HALP: Heuristic aided learned preference eviction policy for Youtube content delivery network," in *Proc. USENIX NSDI*, 2023, pp. 1149–1163.

[39] J. Chen et al., "Darwin: Flexible learning-based CDN caching," in *Proc. ACM SIGCOMM Conf.*, Sep. 2023, pp. 981–999.

[40] 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 *Proc. IEEE Conf. Comput. Commun. (IEEE INFOCOM)*, May 2021, pp. 1–10.

[41] Z. Song, D. S. Berger, K. Li, and W. Lloyd, "Learning relaxed belady for content distribution network caching," in *Proc. USENIX NSDI*, Jan. 2020, pp. 529–544.

[42] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *Ann. Statist.*, vol. 29, no. 5, pp. 1189–1232, Oct. 2001.

[43] Z. Akhtar et al., "AViC: A cache for adaptive bitrate video," in *Proc. 15th Int. Conf. Emerg. Netw. Experiments Technol.*, Dec. 2019, pp. 305–317.

[44] D. Zhou, H. Yu, M. Kaminsky, and D. G. Andersen, "Fast software cache design for network appliances," in *Proc. USENIX ATC*, Jan. 2020, pp. 657–671.

[45] J. Yang, A. Sabnis, D. S. Berger, K. Rashmi, and R. K. Sitaraman, "C2DN: How to harness erasure codes at the edge for efficient content delivery," in *Proc. USENIX NSDI*, 2022, pp. 1159–1177.

[46] M. Garetto, E. Leonardi, and G. Neglia, "Similarity caching: Theory and algorithms," in *Proc. IEEE Conf. Comput. Commun. (IEEE INFOCOM)*, Jul. 2020, pp. 526–535.

[47] K. S. Elsayed, F. Geyer, and A. Rizk, "Utility-driven optimization of TTL cache hierarchies under network delays," in *Proc. IFIP Netw. Conf. (IFIP Netw.)*, Jun. 2024, pp. 249–257.

[48] X. Jin et al., "NetCache: Balancing key-value stores with fast in-network caching," in *Proc. 26th Symp. Operating Syst. Princ.*, Oct. 2017, pp. 121–136.

[49] G. Yan and J. Li, "Towards latency awareness for content delivery network caching," in *Proc. USENIX ATC*, 2022, pp. 789–804.

[50] H. Dai, B. Liu, H. Yuan, P. Crowley, and J. Lu, "Analysis of tandem PIT and CS with non-zero download delay," in *Proc. IEEE Conf. Comput. Commun. (IEEE INFOCOM)*, May 2017, pp. 1–9.

[51] M. Ahmadi, J. Roberts, E. Leonardi, and A. Movaghar, "On the effectiveness of the PIT in reducing upstream demand in an NDN router," *Perform. Eval.*, vol. 138, Apr. 2020, Art. no. 102081.
[52] M. Dehghan, B. Jiang, A. Dabirmoghaddam, and D. Towsley, "On the analysis of caches with pending interest tables," in *Proc. 2nd ACM Conf. Inf.-Centric Netw.*, Sep. 2015, pp. 69–78.
[53] K. Elsayed and A. Rizk, "Time-to-live caching with network delays: Exact analysis and computable approximations," *IEEE/ACM Trans. Netw.*, vol. 31, no. 3, pp. 1087–1100, Mar. 2023.

Haisheng Tan (Senior Member, IEEE) received the B.E. degree (Hons.) in software engineering and the B.S. degree (Hons.) in management from the University of Science and Technology of China (USTC), and the Ph.D. degree in computer science from The University of Hong Kong (HKU). He is currently a Professor with USTC. He has published over 80 papers in prestigious journals and conferences, mainly in the areas of the AIoT and edge computing. His research interests include algorithms and networking. He received the Best Paper Award from WASA 2019, CWSN 2020, PDCAT 2020, and ICAPDS 2021.

Yi Wang received the B.Ec. degree from Shanghai University of Finance and Economics (SUFE) in 2020. She is currently pursuing the M.S. degree with the University of Science and Technology of China (USTC). Her main research interests include networking, edge computing, and algorithm design.

Chi Zhang received the B.Eng. degree in computer science and technology from the University of Science and Technology of China (USTC) with the honor of the Talent Program in Computer and Information Science and Technology in 2017 and the Ph.D. degree in computer science and technology from USTC in 2023. He is currently an Associate Professor with Hefei University of Technology. His main research interests include data center networking, cloud computing, and algorithms. In 2023, he received the prestigious Humboldt Research Fellowship awarded by the Alexander von Humboldt Foundation of Germany.

Guopeng Li received the B.Eng. degree in computer science and technology from Central South University in 2020. He is currently pursuing the Ph.D. degree with the University of Science and Technology of China (USTC). His main research interests include edge intelligence, LLM-based agent, and machine learning systems.

Haohua Du received the M.S. and Ph.D. degrees from the Department of Computer Science, Illinois Institute of Technology, USA, in 2015 and 2019, respectively. She is currently an Assistant Professor with the School of Cyber Science and Technology, Beihang University. Her research interests include wireless sensing, wireless communications, and other IoT applications.

Zhenhua Han received the B.Eng. degree in electronic and information engineering from the University of Electronic Science and Technology of China in 2014 and the Ph.D. degree from The University of Hong Kong (HKU). He is currently a Researcher with Microsoft Research Asia. Many of his works have been published in top venues, such as USENIX OSDI, IEEE INFOCOM, and IEEE/ACM TRANSACTIONS ON NETWORKING. His research interests include cloud computing, cluster scheduling, machine learning systems, online algorithms, and stochastic optimization.

Shaofeng H.-C. Jiang received the Ph.D. degree from The University of Hong Kong. He is currently an Assistant Professor with the Center on Frontiers of Computing, Peking University (PKU). Before he joined PKU, he was a Post-Doctoral Researcher with the Weizmann Institute of Science and then an Assistant Professor with Aalto University. His research interests include theoretical computer science, with a focus on algorithms for massive datasets, online algorithms, and approximation algorithms.

Xiang-Yang Li (Fellow, IEEE) received the bachelor's degree from the Department of Computer Science, Tsinghua University, in 1995, the bachelor's degree from the Department of Business Management, Tsinghua University, in 1995, and the M.S. and Ph.D. degrees from the Department of Computer Science, University of Illinois at Urbana-Champaign, in 2000 and 2001, respectively. He was a Full Professor with Illinois Institute of Technology, Chicago, IL, USA. He is currently a Full Professor and the Executive Dean of the School of Computer Science and Technology, University of Science and Technology of China, Hefei, China. He published a monograph *Wireless Ad Hoc and Sensor Networks: Theory and Applications*. His research interests include the artificial intelligent Internet of Things, mobile computing, data sharing and trading, and privacy. He is a Fellow of ACM. He is an ACM Distinguished Scientist.