Online File Caching on Multiple Caches in Latency-Sensitive Systems

Guopeng Li, Chi Zhang, Hongqiu Ni, and Haisheng Tan(✉)

LINKE Lab and the CAS Key Lab of Wireless-Optical Communications, University of Science and Technology of China (USTC), Hefei, China
{guopengli,gzhnciha,nhq0806}@mail.ustc.edu.cn, hstan@ustc.edu.cn

Abstract. Motivated by the presence of multiple caches and the non-negligible fetching latency in practical scenarios, we study the online file caching problem on multiple caches in latency-sensitive systems, e.g., edge computing. Our goal is to minimize the total latency for all file requests, where a file request can be served by a hit locally, fetching from the cloud data center, a delayed hit, relaying to other caches, or bypassing to the cloud. We propose a file-weight-based algorithm, named `OnMuLa`, to support delayed hits, relaying and bypassing. We conduct extensive simulations on Google' trace and a benchmark YCSB. The results show that our algorithms significantly outperform the existing methods consistently in various experimental settings. Compared with the state-of-the-art scheme supporting multiple caches and bypassing, `OnMuLa` can reduce the latency by 14.77% in Google's trace and 49.69% in YCSB.

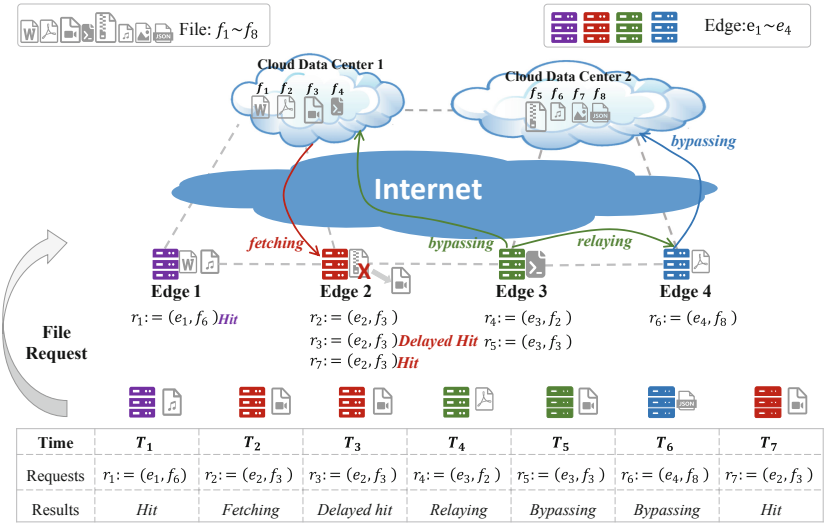
Keywords: Cache · Mobile edge computing · Delayed hits

1 Introduction

In computer architecture, *cache* is designed to address the gap between memory access latency and processor processing speed [9]. Today, the concept of cache has been extended to different areas and has played an important role in improving system performance. The structure located between different types of hardware and used to eliminate the impact caused by the disparities of access time can be called a *cache*, such as disk cache, CDNs cache, Web cache and DNS cache. Taking advantage of temporal locality, storing some files that are about to be accessed into cache is one way to improve performance by using cache [18]. In this context, in order to minimize the total cost, one of the first and most important problems is to select which files to store in the cache and which files to replace when the cache is full, *i.e.*, the *online file caching problem*. In the traditional online file caching problem, we are given a cache with a specified size k and a sequence of file requests, where each file has a specified size and a specified retrieval cost. The goal of the traditional online file caching is usually to minimize the cache misses or the total retrieval cost of file retrievals by maintaining

T. N. Dinh and M. Li (Eds.): CSoNet 2022, LNCS 13831, pp. 292–304, 2023.
https://doi.org/10.1007/978-3-031-26303-3_26

files in the cache [21]. However, in practical scenarios such as Mobile Edge Computing(MEC) [16] and Content Delivery Networks (CDNs) [6], due to the long physical distance, the latency for fetching a file from the cloud data center can be up to 100 ms, with the increase in network bandwidth and system throughput, 1M file requests can be arrived in a second, *i.e.*, the average inter-time for two consecutive file requests could be as low as 1μs [2]. During the period when a missed file is retrieved from the cloud data center, the subsequent requests for the same file can not be served immediately, which is not a *hit*, and is also different from a *miss*, which is called a ***delay hit*** [23]. In MEC and CDNs, in addition to local hit and fetching files from cloud to local, file requests can also be served by *bypassing* and *relaying*: 1) In cloud-based scenarios, the request can be sent to and served at the remote cloud, which is called a *bypassing*. 2) Since there is more than one edge server or PoP (regarded as cache) in MEC and CDNs, respectively, when the requested file is not already stored in the cache that the request arrives, the request can be sent to a nearby cache that has the same file, which is called a *relaying*. An example of online file caching in MEC is demonstrated in Fig. 1.



Time	T_1	T_2	T_3	T_4	T_5	T_6	T_7
Requests	$r_1 := (e_1, f_6)$	$r_2 := (e_2, f_3)$	$r_3 := (e_2, f_3)$	$r_4 := (e_3, f_2)$	$r_5 := (e_3, f_3)$	$r_6 := (e_4, f_8)$	$r_7 := (e_2, f_3)$
Results	*Hit*	*Fetching*	*Delayed hit*	*Relaying*	*Bypassing*	*Bypassing*	*Hit*

Fig. 1. An example in mobile edge computing system, where r_1 is served at e_1 locally, r_2 triggers the fetching and evict operation, r_3 is delayed served at e_2, r_4 is relayed to e_4, r_5 and r_6 are bypassed to the cloud data center, and r_7 is served at e_2 locally.

Motivating Example. As shown in Fig. 1, there are four edge servers and two cloud data centers connected through Internet, and eight different types of files with different sizes are stored in the cloud data center. Before T_1, each edge server has stored some files. $r := (e, f)$ represents the request arriving on an edge server e for file f. In order to demonstrate the ways (***hit, fetching, delayed hits, relaying and bypassing***) for serving requests on multiple caches

in latency-sensitive systems, we have designed eight requests as an example. **1) *hit***: If file f is already stored on e, the request can be served locally with no latency (as r_1 and r_7 illustrated in the figure). **2) *fetching***: If file f has not been stored in e, edge server can fetch file from the cloud, due to the non-negligible fetching latency, file will not be stored in the edge server as soon as the fetching operation is triggered (as r_2 illustrated, and we assume the fetching operation will be finished at T_6). **3) *delayed hits***: During the file fetching period, the subsequent requests can not be served until the fetching operation is finished and suffer delayed serve latency (as r_3 illustrated). **4) *relaying***: If file f has not been stored on server e, but a nearby server e' has f, we can relay and serve the request r on e' with relaying latency (as r_4 illustrated). **5) *bypassing***: When file f does not exist in the entire multiple caches system, in addition to fetching f from the cloud data center to the cache, we can bypass the request to and serve it at the cloud data center with bypassing latency (as r_5 and r_6 illustrated). Moreover, since the capacity of the edge server is limited, where some existing files might be replaced if the edge server is full. For r_2, when the fetching operation is triggered, we should check if there is enough empty space to store f_3. The capacity of cache e_2 does not allow it to store both f_3 and f_5 because both of them are large files, and we choose to evict f_5 from e_2.

Several algorithms have been proposed for the online file caching problem in the past, such as recency-based LRU [15], frequency-based LFU [4], recency and frequency based ARC [11], learning-based LeCaR [17], Camul [16] that uses marking methods in multiple caches system and CaLa [23] which handle weights and supports bypassing, but none of them studied the model as above. CaLa was designed for the one cache system, Camul focuses on fixed relaying and fetching cost without considering the non-uniform size of various files and the non-negligible fetching latency. In this work, we study the online file caching problem on multiple caches in latency-sensitive systems.

Our Contribution. In this work, we study the online file caching problem with relaying and bypassing on *multiple* caches in latency-sensitive systems, and propose an online algorithm to minimize the total relaying, fetching, delayed hits and bypassing latency. Our contributions are summarized as follows.

- We investigate a practical online file caching problem with relaying and bypassing on multiple caches in latency-sensitive system to minimize the total latency of all file requests (Sect. 3).
- We propose an online algorithm, called `OnMuLa`, to support delayed hits, relaying and bypassing. To the best of knowledge, `OnMuLa` is the first online algorithm for the online general file caching problem with delayed hits on multiple caches system (Sect. 4).
- We conduct extensive simulations on Google's trace and YCSB. Compared with Camul, the state-of-the-art algorithm that deals with multiple caches and bypassing, in default settings, `OnMuLa` can reduce the latency by 14.77% in Google's trace and 49.69% in YCSB (Sect. 5).

2 Related Work

Online file caching problem is often mentioned in computer and network systems. It can be described as follows, after being given a cache of k slots that each store one file, if the file has been cached in a slot when a request for a file arrives online, the request is served with no cost. If the file has not been cached, it has to be fetched into the cache with fetching cost. The online file caching algorithms maintain the contents of k slots to minimize the total fetching cost for all requests. LRU [15] is a classic algorithm widely used in practical scenarios. Considering the non-uniform fetching cost, Young *et al.* [21] proposed Landlord. In order to support bypassing, Landlord is extended to Landlord with Bypassing (LLB) [7]. Investigating general online caching problem on multiple caches with relaying and bypassing, Camul [16] has a high hit ratio with lower total cost than previous works.

Table 1. Related work of online caching problems

Algorithms	File size & Fetching cost	Delayed hits	Bypassing	Multiple caches
LRU [15]	Uniform	✘	✘	✘
Landlord [21]	Non-uniform	✘	✘	✘
LLB [7]	Non-uniform	✘	✔	✘
MAD [2]	Uniform	✔	✘	✘
CaLa [23]	Non-Uniform	✔	✔	✘
Camul [16]	Uniform	✘	✔	✔
`OnMuLa` [this work]	Non-Uniform	✔	✔	✔

In latency-sensitive systems, few works have paid attention to *delayed hits* so far. A representative work is the online paging problem studied in [2], Atre *et al.* creatively reveal the importance of delayed hits, and propose MAD, an online algorithm that combines file aggregation delays into existing caching algorithms (such as LRU and ARC). Zhang *et al.* [23] proposed CaLa, a general framework that imitates an existing file caching algorithm to get guaranteed performance in their work. Besides, caching problem has attracted a wide range of works from emerging application areas, such as mobile system [10,13], CDNs [3,20], container caching [8,12], and deep learning system [1,22]. For example, Beckmann *et al.* [3] proposed LHD to predict the hit density of each object to filter objects that have a small contribution to the cache hit rate. Yan *et al.* [20] proposed a timer-based mechanism that can optimize the mean caching latency. This work addresses the non-uniform file size and fetching latency, delayed hits, relaying and bypassing on multiple caches. We summarize some related results in Table 1.

3 Problem Formulation

Motivated by latency-sensitive scenarios such as MEC and CDNs, we consider the online general file caching model with multiple caches and cloud data centers. Let $\mathcal{F} = \{f_1, f_2, \ldots, f_N\}$ be the set of all kinds of files, and we assume all files are available in the cloud data center. Each file f_i has size s_{f_i} and fetching latency t_{f_i}. Without loss of generality, we assume all file sizes are integers. $\mathcal{E} = \{e_1, e_2, \ldots, e_M\}$ represents the set of the caches in the system, the size of each cache e_i is K_i. Naturally, the sum of sizes of files stored in each cache can not exceed the size of cache, *i.e.*, $\sum_{f \text{ in cache } e_i} s_f \leq K_i$. Let $\mathcal{R} = (r_1, r_2, \ldots)$ be the sequence of file requests, a request r is a pair $(e, f) \in \mathcal{E} \times \mathcal{F}$, meaning a file f on cache e is requested. All requests arrive in an online manner, *i.e.*, we can not get future information and no assumption is made on the arrival patterns. 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. In the multiple caches system, when a request $r := (e, f)$ arrives at time T, the following 5 types of operations may be performed. The objective of this problem is to minimize the total relaying, bypassing, fetching and delayed hits latency to serve all requests.

- *Hit*: If the requested file f is already cached in cache e, then this request is served locally with no latency, *i.e.*, called a *hit*.
- *Relaying*: If e does not have f stored but another cache e' does, the request may be relayed and processed at e' with relaying latency t_r.
- *Bypassing*: A request may be bypassed to the cloud with bypassing latency t_b. Note that in this case, the file is not necessarily fetched into the cache.
- *Fetching*: When file f does not exist in the entire system, f may be fetched to the cache with fetching latency t_f, *i.e.*, request r can not be served until time $T + t_f$. Once f is cached, we need to decide which files should be replaced if the cache is already full.
- *Delayed Hits*: During the fetching period, *i.e.*, from T to $T + t_f$, before file f is actually stored in the cache, all requests that require file f on cache e at time slot $t' \in \{T+1, T+2, \ldots, T+t_f-1\}$ can only be served at time $T + t_f$ and suffered a latency of $t_f - (t' - T)$, which are *delayed hits*.

4 Algorithm Design

In this section, we first propose a method to measure the importance of each file, called *file weight* (Sect. 4.1). Then, we present our algorithm `OnMuLa` and its version without bypassing `OnMuLa`$^-$ in Algorithm 2.

4.1 File Weight

The central challenge of this problem is how to deal with delayed hits in multiple caches system. The method in [23] does not capture the relaying operation in the multiple caches scenario, in order to avoid the impact of the lack

Algorithm 1: Update Weight

Input *Parameter* γ*, cache* e*, file* f*, latency* l
if f_e.state = OUT **then**
  f_e.cumulativeDelay $\leftarrow$ f_e.cumulativeDelay + t_f;
  f_e.numFetching $\leftarrow$ f_e.numFetching + 1;
if f_e.state = FETCHING **then**
  f_e.cumulativeDelay $\leftarrow$ f_e.cumulativeDelay + l;
f_e.aggregateDelay $\leftarrow \frac{f_e.\text{cumulativeDelay}}{f_e.\text{numFetching}}$;
f_e.weight $= (1-\gamma) * f.aggregateDelay + \gamma * t_f^2$;

of consideration for relaying, as Eqn. 1 shown (assume start fetching f into e at time T) and AggDelay(e, f) = CumuDelay(e, f)/ # of fetching f into e, we refine the method to calculate CumuDelay to estimate the actual latency in the multiple caches system. Not only *delayed hits* and *fetching* of request on cache e contribute to CumuDelay(e, f), but the *relaying* from other caches also affect CumuDelay(e, f). In order to get a trade-off between CumuDelay and the upper bound of the total latency caused by the file's miss, t_f^2, we use $W_{f_e} = (1-\gamma)\text{AggDelay}(e, f) + \gamma t_f^2$ to represent the weight of each file on each cache, where parameter γ is used to adjust between these two methods.

$$\begin{aligned}\text{CumuDelay}(e, f) = t_f + &\sum_{1 \le \tau \le t_f - 1} (t_f - \tau)[f \text{ is requested at } T + \tau] \\ &+ \sum (\max(t_f - \sigma, 0) + t_r)[\text{relay from } e' \text{to } e \text{ at } T + \sigma].\end{aligned} \tag{1}$$

4.2 Design of `OnMuLa`

The main algorithm is defined in Algorithm 2, and the weight update algorithm is defined in Algorithm 1. Initially, the caches in the system are initialized to empty. We use IN, FETCHING and OUT to represent the state of file f on cache e, where $f_e.state$ = IN means f is already cached in e and $f_e.state$ = FETCHING means f is already in a fetching period. When a new request $r := (e, f)$ that requests for file f on cache e arrives at T, `OnMuLa` decides how to serve r based on $f_e.state$. **1)** ***hit***: If $f_e.state$ = IN, then serve f on cache e with no latency (Line 13 to Line 14). **2)** ***delayed hits***: If $f_e.state$ = FETCHING, r will be served until the fetching operation of f is finished and sufferd the latency $t - T$ (Line 15 to Line 17). **3)** ***relaying***: If there is a cache $e' \neq e$ has f in the multiple caches system, and the sum of waiting latency t_w and t_r is less than t_f, then relay r to e' with latency $t_w + t_r$. t_w represents the latency that r suffered on e' after be relayed to e' (Line 19 to Line 21). **4)** ***fetching*** and **5)** ***bypassing***: We use Landlord [21] and Landlord with Bypassing(LLB) [7] as the replacement policy in `OnMuLa`$^-$ and `OnMuLa`, respectively. It can be substituted with other algorithms which can handle weight and support bypassing. Landlord and LLB maintain a credit for each file to determine whether it should be evicted.

Specifically, in the implementation of OnMuLa$^-$and OnMuLa, let f_e.weight be the credit in Landlord and LLB. When a request $r := (e, f)$ has not been served after the above process (Line 13 to Line 21), OnMuLa$^-$ checks the remaining size of e and uses Landlord to replace the files in e if e does not have enough to cache f, then fetch f to e. OnMuLa will first suppose f in e^α, the copy of cache e, and set the credit of f (Line 25). If f_{e^α}.weight > 0 in the end of the replacement, then fetch f to e, otherwise, bypass the request.

For Line 15 to Line 17 in Algorithm 2, we prove Theorem 1, *i.e.*, if there is more than one copy of file f in the multiple caches system, the request should be delayed served on e instead of being relayed.

Theorem 1. *If there is more than one copy of file f (the state of f is* IN *or* FETCHING*) in multiple cache systems, when f_e.*state = FETCHING *and $f_{e'}$.*state = FETCHING *or* IN*, request $r := (e, f)$ should be delayed served on e instead of being relayed to e'.*

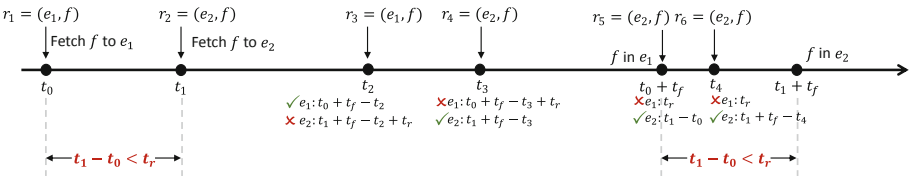


Fig. 2. An example of Theorem 1, after t_1, the request for f on e_2 should be served on e_2 instead of be relayed to e_1.

Proof. First, we prove Theorem 1 for OnMuLa$^-$. In the multiple caches system, as shown in Fig. 2, suppose there is no request for f before t_0. At t_0, r_1 requests f on e_1, however, $f_{e_1}.state$ = OUT, then start to fetch f into e_1, the fetching process will be finished at $t_0 + t_f$. At t_1, r_2 requests f on e_2, $f_{e_2}.state$ = OUT, $f_{e_1}.state$ = FETCHING, however, the condition in Line 19 of Algorithm 2 is not satisfied, $t_r + t_0 + t_f - t_1 > t_r$, *i.e.*, $t_1 - t_0 < t_r$, start to fetch f into e_2, the process will be finished at $t_1 + t_f$. Now, there is more than one copy of file f in the multiple caches system. Next, we use the converse method to prove Theorem 1. During the time period t_1 to $t_1 + t_f$, requests r_4, r_5 and r_6 request f on cache e_2. For r_4, we will relay the request to cache e_1 if and only if $t_0 + t_f - t_3 + t_r < t_1 + t_f - t3$, *i.e.*, $t_1 - t_0 > t_f$, however, this contradicts $t_1 - t_0 < t_r$. As a more direct example, for r_5, the condition is $t_r < t_1 - t_0$. For r_6, $t_r < t_1 + t_f - t_4$ should be satisfied, as Fig. 2 shown, it is impossible. For OnMuLa, since bypass is allowed, OnMuLa can fetch files into the multiple caches system or bypass the requests when the condition in Line 19 is not satisfied. If all requests are bypassed, file will not be cached in the system, the assumption of the theorem is not satisfied. If there is more than one copy of the file are cached in the system, as same as the above proof for OnMuLa$^-$. □

Algorithm 2: Main Algorithm

Input *Request* $r := (e, f)$, *the size of* f s_f, *Fetching Latency* t_f, *Bypass Latency* t_b, *Relay Latency* t_r, *bool* $\mathcal{B}$ *represents allow bypass or not* ;
$\mathcal{C} \leftarrow \varnothing$, $\mathcal{C}$ represents the set of files cached in e;
Fetching files $\mathcal{F}_{\text{f}} \leftarrow \varnothing$, $(e, f, t) \in \mathcal{F}_{\text{f}}$ means file f will arrive on e at time t;
Timer $T \leftarrow 0$;
while True **do**
  for $(e, f, t) \in \mathcal{F}_{\text{fetching}}$ **do**
    if $t <= T$ **then**
      if f_e.state = FETCHING **then**
        f_e.state $\leftarrow$ IN;
        $\mathcal{C} \leftarrow \mathcal{C} \cup \{f\}$;
      Serve all the buffered and relayed requests for f on e;
  while new request $r := (e, f)$ for file f on e arrive at T **do**
    if $f_e.state$ = IN **then** `// hit`
      serve f on e with no latency;
    if $f_e.state$ = FETCHING **then** `// delayed hit`
      delayed serve f on e at time t with latency $t - T$;
      UpdateWeight$(e, f, t - T)$
    if $f_e.state$ = OUT **then**
      if there is a cache e' has f and $t_r + t_w$ ¡ t_f **then** `// relay`
        relay $r := (e, f)$ to e' with latency $t_r + t_w$;
        UpdateWeight$(e', f, t_r + t_w)$;
      else
        Let cache e^{α} be a copy of e ;
        if $\mathcal{B}$ = Ture **then**
          Let f in e^{α}, UpdateWeight$(e^{\alpha}, f, 0)$;
        if remain size of $e^{\alpha} < s_f$ **then**
          $\mathcal{F}_{\text{evicts}} \leftarrow$ Replace(e^{α}, f);
          for $f' \in \mathcal{F}_{evicts}$ **do**
            Evict f' from e, $\mathcal{C} \setminus \{f'\}$;
            $f_{e'}$.state $\leftarrow$ OUT;
        if $f_{e^{\alpha}}$.weight > 0 or $\mathcal{B}$ = False **then**
          f_e.state $\leftarrow$ FETCHING;
          $\mathcal{F}_{\text{fetching}} \leftarrow \mathcal{F}_{\text{fetching}} \cup \{(e, f, T + t_f)\}$;
          UpdateWeight(e, f, t_f);
        else `// bypass`
          Bypass this request with latency t_b ;
  $T \leftarrow T + 1$;

5 Evaluation

In this section, we evaluate the performance of `OnMuLa` and `OnMuLa`$^-$on two datasets: (1) the production trace from Google [14], (2) the Yahoo! Cloud Serving Benchmark (YCSB) [5]. We compare `OnMuLa` and `OnMuLa`$^-$ with several caching algorithms *i.e.*, LRU [15], LRU-MAD [2], Landlord with Bypassing [7], Landlord [21], Camul [16], CaLa with Bypassing and CaLa [23]. The details of the results are shown in Sect. 5.2 and we highlight our key findings as follows.

- Compared with Camul, the state-of-the-art caching algorithm in multiple caches system. With default settings, in Google's trace, `OnMuLa`$^-$ can reduce latency by 12.62%, this reduction will be increased to 14.77% if bypassing is allowed. In YCSB, `OnMuLa` reduces latency by 49.69% compared to Camul.
- The performance of the algorithm can vary significantly for different traces. For the one cache case, `OnMuLa`$^-$ and CaLa get poor performance in Google's trace, and conversely, work well in YCSB.
- If the cache size is small (e.g., sum of 0.001% to 0.01% of the popular files), `OnMuLa` outperforms other algorithms by bypassing.

5.1 The Experiment Settings

By default, we set 400 caches for Google's trace and 200 caches for YCSB. The default cache size is the sum of the sizes of top 0.01% popular files. We let relay latency $t_r = 0.002 \times$ fetching latency t_f [16], bypassing latency $t_b = t_f$ [23]. We set the average inter-request time to $10^{-4}s$ [19], and the average default fetching latency of files is set to $0.1s$ [2]. For `OnMuLa`, `OnMuLa`$^-$, CaLa and CaLa with Bypassing, the default value of γ is set to 0.1. The metrics used to evaluate the performance of algorithms is the total latency incurred of all requests. And we use the latency improvement relative to LRU to measure the performance of the algorithm when the parameters change, *i.e.*, Latency Improvement of A = (Latency(LRU) − Latency(A))/Latency(LRU), a higher latency improvement means better performance.

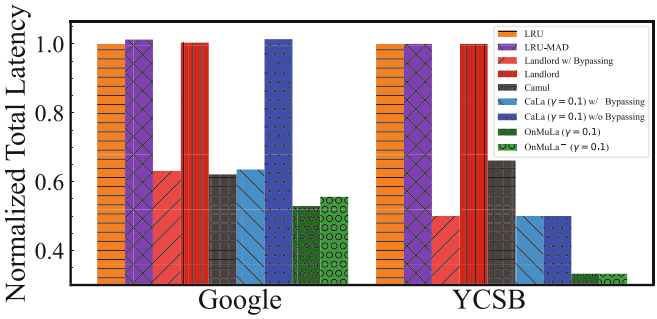


Fig. 3. Overall performance.

5.2 Experimental Results

Overall Performance. We first evaluate the overall performance of `OnMuLa` and `OnMuLa`$^-$, and compare them with the baselines, where parameters are set as default values. The experimental results are shown in Fig. 3. In Google's trace, the latency improvement of `OnMuLa`$^-$ to CaLa is 45.16%, and `OnMuLa` to Camul is 14.77%. For the results in YCSB, the latency improvement of `OnMuLa`$^-$to CaLa is 66.72%, and `OnMuLa` to Camul is 49.69%.

Sensitivity Analysis. In this part, we perform sensitivity analysis of the parameters in the experiment settings, including the number of caches, fetching latency, cache size and γ.

Impact of Number of Caches. Figure 4(a) and Fig. 4(b) illustrate the impact of the number of caches, which varies from 1 to 1000 for Google's trace and from 1 to 400 for YCSB. In Google's trace, `OnMuLa`$^-$ performs poorly in the one cache system, with the number of caches increasing, the performance of `OnMuLa`$^-$ becomes better. Due to the design for delayed hits and non-uniform file size, the performance of `OnMuLa` is better than Camul. In YCSB, for the one cache case, due to the design for non-uniform file size and delayed hits, `OnMuLa`$^-$ and CaLa avoid bringing large and infrequent files into cache. For the multiple caches case, with the increasing number of caches, the performance of `OnMuLa`$^-$, `OnMuLa` and Camul increase marginally. When the number of caches is large enough, 400 for Google's trace and 200 for YCSB, `OnMuLa` and `OnMuLa`$^-$ achieve the best performance among the nine algorithms.

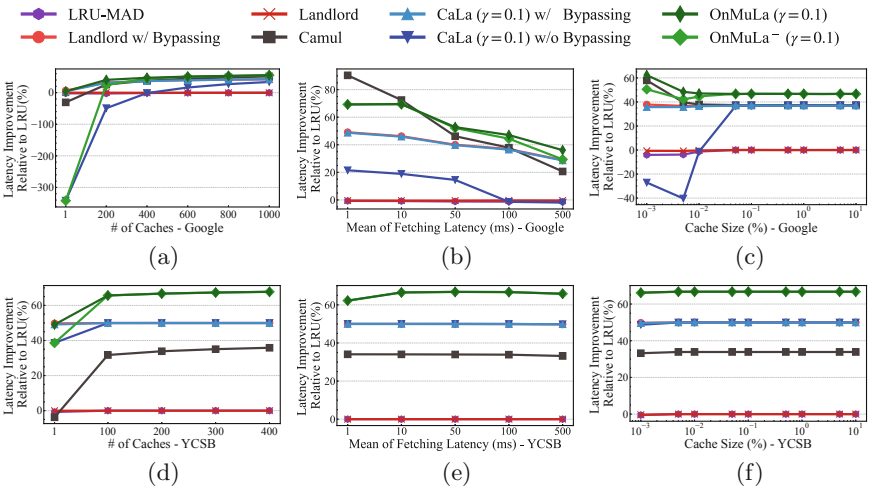


Fig. 4. Impact of number of caches, fetching latency and cache size.

Impact of Fetching Latency. We show the result of the impact of fetching latency in Fig. 4(b) and Fig. 4(e). The fluctuation of the curves reflects the different sensitivity of various algorithms to the fetching latency in different traces. In Google's trace, the performance of `OnMuLa` and `OnMuLa`$^-$ gradually outperforms Camul when the fetching latency increases, since their awareness of delayed hits. In YCSB, the performance of other algorithms besides `OnMuLa` and `OnMuLa`$^-$ is more likely the case without delayed hits. The reason is that the request locality of YCSB is too low, and there are few requests with delayed hits, especially when the requests are distributed to multiple caches.

Impact of Cache Size. Figure 4(c) and Fig. 4(f) show the impact of cache size for Google's trace and YCSB, respectively. In Google's trace, when the cache size is small, the performance of `OnMuLa` and Camul are far beyond other algorithms. This is because bypassing can avoid evicting some frequently requested files from the cache. As the cache size gradually increases, the performance of `OnMuLa`$^-$, CaLa gradually catches up with `OnMuLa`. Due to the discreteness of files' size in the trace and the non-consecutiveness between different caches, 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.

Impact of γ. For Google's trace, as shown in Fig. 5(a), the best performance is achieved when $\gamma = 0.15$, which shows that it is better to use a value of γ closer to the aggregate delay for burst requests. For YCSB, as shown in Fig. 5(b), the performance of `OnMuLa`$^-$and `OnMuLa` remains stable as γ changes from 0 to 0.2.

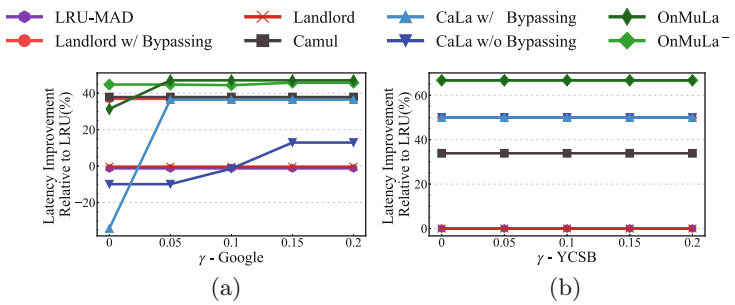


Fig. 5. Impact of γ.

6 Conclusion

In this paper, we study the online file caching problem on multiple caches with relaying and bypassing in latency-sensitive systems. The objective is to minimize the total latency to serve all requests. We first propose *file weight* to capture the potential impact of the fetching and relaying process. Then we propose an online algorithm `OnMuLa` to support delayed hits, relaying and bypassing, and

its version without bypassing, `OnMuLa`$^-$. We evaluate `OnMuLa` and `OnMuLa`$^-$ on Google's trace and YCSB. The experiment results show that compare with Camul, `OnMuLa` can reduce the latency by 14.77% in Google's trace and 49.69% in YCSB.

Acknowledgements. The work is partially supported by NSFC under Grant 62132009, and the Fundamental Research Funds for the Central Universities at China.

References

1. Abdi, M., et al.: A community cache with complete information. In: USENIX FAST 2021, pp. 323–340 (2021)
2. Atre, N., Sherry, J., Wang, W., Berger, D.S.: Caching with delayed hits. In: ACM SIGCOMM (2020)
3. Beckmann, N., Chen, H., Cidon, A.: LHD: Improving cache hit rate by maximizing hit density. In: USENIX NSDI (2018)
4. Breslau, L., Cao, P., Fan, L., Phillips, G., Shenker, S.: Web caching and zipf-like distributions: evidence and implications. In: IEEE INFOCOM'99
5. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: ACM SoCC (2010)
6. Dilley, J., Maggs, B., Parikh, J., Prokop, H., Sitaraman, R., Weihl, B.: Globally distributed content delivery. IEEE Internet Comput. **6**(5), 50–58 (2002)
7. Epstein, L., Imreh, C., Levin, A., Nagy-György, J.: Online file caching with rejection penalties. Algorithmica **71**(2), 279–306 (2015). https://doi.org/10.1007/s00453-013-9793-0
8. Fuerst, A., Sharma, P.: Faascache: keeping serverless computing alive with greedy-dual caching. In: ACM ASPLOS 2021, pp. 386–400 (2021)
9. Karlsson, M.: Cache memory design trade-offs for current and emerging workloads. Ph.D. thesis, Citeseer (2003)
10. Liang, Y., et al.: Cachesifter: sifting cache files for boosted mobile performance and lifetime. In: USENIX FAST 2022, pp. 445–459 (2022)
11. Megiddo, N., Modha, D.S.: Arc: A self-tuning, low overhead replacement cache. In: FAST 2003 (2003)
12. Pan, L., Wang, L., Chen, S., Liu, F.: Retention-aware container caching for serverless edge computing. In: IEEE INFOCOM 2022 (2022)
13. Ramanujam, M., Madhyastha, H.V., Netravali, R.: Marauder: synergized caching and prefetching for low-risk mobile app acceleration. In: MobiSys (2021)
14. Reiss, C., Wilkes, J., Hellerstein, J.: Google cluster-usage trace. In: Technical Report (2011)
15. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Commun. ACM **28**(2), 202–208 (1985)
16. Tan, H., Jiang, S.H.C., Han, Z., Liu, L., Han, K., Zhao, Q.: Camul: online caching on multiple caches with relaying and bypassing. In: IEEE INFOCOM (2019)
17. Vietri, G., et al.: Driving cache replacement with ML-based LeCaR. In: HotStorage 2018 (2018)
18. Wang, J., Hu, Y.: Wolf-a novel reordering write buffer to boost the performance of log-structured file system. In: FAST 2002 (2002)
19. Wendell, P., Freedman, M.J.: Going viral: flash crowds in an open CDN. In: ACM/USENIX IMC (2011)

20. Yan, G., Li, J.: Towards latency awareness for content delivery network caching. In: USENIX ATC 2022, pp. 789–804 (2022)
21. Young, N.E.: On-line file caching. Algorithmica **33**(3), 371–383 (2002). https://doi.org/10.1007/s00453-001-0124-5
22. Yuan, M., Zhang, L., He, F., Tong, X., Li, X.Y.: Infi: end-to-end learnable input filter for resource-efficient mobile-centric inference. In: MobiCom (2022)
23. Zhang, C., Tan, H., Li, G., Han, Z., Jiang, S.H.C., Li, X.Y.: Online file caching in latency-sensitive systems with delayed hits and bypassing. In: IEEE INFOCOM 2022, pp. 1059–1068. IEEE (2022)