Cost-Driven Data Caching in Edge-Based Content Delivery Networks

Yang Wang, Hao Dai, Xinxin Han, Pengfei Wang, Yong Zhang, and Chengzhong Xu, *Fellow, IEEE*

Abstract—In this paper we study a data caching problem in edge-based content delivery network (CDN) where a data item is shared between service requests. Instead of improving the hit ratio with respect to limited capacity as in traditional case, we study the problem based on a semi-homogeneous (semi-homo) cost model in the edge-based CDN with monetary cost reduction as a goal. The cost model is semi-homo in the sense that all pairs of caching nodes have the same transfer cost, but each has its own caching cost rate. In particular, given a stream of requests $\mathcal{R}$ to a shared data item in the edge network, we present a shortest-path based optimal off-line caching algorithm that can minimize the total transfer and caching costs within $O(mn)$ time (m : the number of network nodes, n : the length of request stream) in a proactive fashion. While for the online case, by extending the anticipatory caching idea, we also propose a 2-competitive online reactive caching algorithm and show its tightness by giving a lower bound of the competitive ratio as $2 - o(1)$ for any deterministic online algorithm. Finally, to combine the advantages of both algorithms and evaluate our findings, we also design a hybrid algorithm. Our trace-based empirical studies show that the proposed algorithms not only improve the previous results in both time complexity and competitive ratio, but also relax the cost model to semi-homogeneity, rendering the algorithms more practical in reality. We provably achieve these results with our deep insights into the problem and careful analysis of the solutions, together with a prototype framework.

Index Terms—Data caching, edge-based CDN, shortest-path algorithm, anticipatory caching, competitive analysis

1 Introduction

As THE volume of data across the globe is constantly growing up, especially with mobile devices (e.g., smartphone and tablet PC) gaining popularity to access, data service as a mainstream application has been inspiring great interest to people [2], [3]. As such, it is more important than ever for the service providers, given the dynamic nature of the Internet, to guarantee the quality of service (QoS) and improve the experience of users as well. Content Delivery Network (CDN) [4], [5] is a commonly used technique to deliver copies of content to end users in a cost-effective way. Typically, it achieves this by shipping the content from an origin server to a set of geographically distributed cache servers, and then redirecting the client requests to the most optimal cache server with an attempt to cut down bandwidth costs while ensuring reliable content delivery.

Currently, most CDN servers are deployed at points of presence (PoPs) in the internet exchange (IXP) or at distributed data centres with an aim to improve the reachability, which is more of bringing contents to more areas in the world, not necessarily with faster processes. However, with the prevalence of data services in different domains, this architecture is in facing of growing challenges.

First, the distribution of the servers is too centralised, far away to satisfy the requirements of some time-critical services [6], [7], for example, the users in gaming always expect real-time responses, and even a few milliseconds of delay in download speeds could significantly compromise his/her quality of experience (QoE). Second, as more smart applications are deployed [8], [9], the CDN architecture is often required to design with more compute resources integrated with the storage cache for its functional augmentation, more than just simply bringing data closer, for example, an image CDN for websites can be enabled to calculate how its cached images will appear in different user's gadgets [3].

To fulfill these requirements, the traditional CDN architectures need to be expanded to not only enforce the reachability but also boost the efficiency with respect to the content delivery as its latency may be a determinant for the overall quality of service.

As edge computing is gaining its momentum, this expansion can be naturally attained by integrating with edge network as it is not only distributed much closer to users but also more concerned with bringing processes to the devices it will serve. With this expansion, one can leverage the combined expertise of both CDN and edge computing to provide fast and adaptive contents to its users across the globe as in image CDNs. As a result, both reachability and efficiency can be achieved for content delivery. Given these benefits, a lot of companies are currently taking edge computing as a viable mean to improve CDN, which results in so-called *Edge-based Content Delivery Network* (edge-based CDN) [6], [10], [11], [12], [13], [14], [15].

- *Yang Wang, Hao Dai, Xinxin Han, Pengfei Wang, and Yong Zhang are with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China. E-mail: {yang.wang1, hao.dai, xx.han, pf.wang, zhangyong}@siat.ac.cn.*
- *Chengzhong Xu is with the State Key Lab of IoTSc, Department of Computer Science, University of Macau, Macau SAR 999078, China. E-mail: czxu@um.edu.mo.*

Manuscript received 12 Mar. 2021; revised 23 Aug. 2021; accepted 24 Aug. 2021. Date of publication 30 Aug. 2021; date of current version 3 Feb. 2023. (Corresponding author: Yong Zhang.) Digital Object Identifier no. 10.1109/TMC.2021.3108150

However, to exploit these benefits, some new challenges have to be addressed before the edge-based CDNs can be widely deployed. Traditionally, operators are locked out of the commercial model for CDN services as they are often hosted in data centres or IXPs, not within the mobile edge network. However, with the edge-based CDN introduced, telcos could play a more significant role in the CDN ecosystem and more revenue from their networks and services [16], [17]. As a result, the TTL-based content caching scheme in current CDN is highly desired to optimize for cost-effective uses of the edge network—not only accelerating the transfer of data items with the reduced number of requests to the central origin but also minimizing the total service cost. As such, how to design an optimized caching schedule for sharing of data items between the requests received by each server with minimum monetary cost in content delivery edge is a highly desirable problem, which is also the goal of this paper [18], [19].

To this end, we model the optimization problem as a *caching problem* where a fully-connected network is assumed to share a set of data items via caching, transferring, and replicating operations among a group of cache servers (i.e., the network nodes) to minimize the overall service cost for a sequence of incoming requests. In particular, the cost model considered in our case is *semi-homogeneous* (semi-homo model for short), which means the transfer cost between any pair of cache servers in CDN is identical while the caching cost per time unit at each individual server is different. The semi-homo cost model in our study is practical as the billing policy for data storage and transfer in the edge is often borrowed from its cloud counterpart as in AliCloud [20], whose prices are typically fixed on per unit size of the data within a region, regardless of the exact locations on the same region [21].[1]

Based on this model, we first propose an optimal *proactive caching* (pro-caching) algorithm for edge-based CDN, which is able to schedule a shared data item ahead of its expected requests across a group of caches with minimum cost in a long term. This algorithm is by nature off-line, and effective for the edge-based CDN with *push network* as its outsourcing strategy. We achieve the pro-caching algorithm by developing a new idea to reduce the caching problem to a simple *shortest path problem* in a directed weighted graph, which is derived from standard *instance graph* by carefully adjusting the edge weights in its defined *shadow regions*. Although the algorithm is effective in both time and space complexities and optimal in cost reduction, it relies heavily on the availability of pre-defined sequence of requests, which is not always feasible in practice unless an accurate predictor for the long-term requests can be achieved [22], [23].

To address this issue, we then investigate this problem in its online form and obtain an online *reactive caching* (re-caching) algorithm with 2-competitive ratio by extending the concept of *anticipatory caching* [24] to the semi-homo cost model. The algorithm is amenable to the edge-based CDN with *pull network* as its outsourcing strategy (see next section). To show the tightness of the competitive ratio, we also prove the lower bound of $2-o(1)$ for any deterministic online algorithm.

Although each respective algorithm is effective to its own case, an algorithm that combines the advantages of both proactive and reactive strategies could be more beneficial when it comes to fully exploit the potentials of edge-based CDNs while reducing their service costs. To this end, we also design a hybrid algorithm that integrate both of the algorithms. Our trace-based empirical studies show that the proposed algorithms, both proactive and reactive, not only ameliorates the experience of users by improving the QoS of content access, but also reduces the costs for service providers. As such, it is essential to the success of edge-based CDNs.

In summary, based on a semi-homo cost model, we formulate the data sharing in content delivery edge as an optimization problem of collaborative caching in a fully connected edge network, and make the following contributions to address this problem in this paper.

1) By following the shortest-path idea, we design a fast optimal off-line pro-caching algorithm, which can cache and transfer the shared data item in a m-node network to serve a n-length request sequence within $O(mn)$ time complexity.
2) With the concept of anticipatory caching, we further present an efficient 2-competitive re-caching algorithm for the online case and show its tightness by giving a lower bound of $2-o(1)$ for the competitive ratio of any deterministic online algorithm.
3) To combine the advantages of both pro-caching and re-caching algorithms, we also develop a hybrid caching algorithm by integrating them as a whole, and investigate it with respect to a simulated edge network.
4) We conduct an empirical study by comparing our algorithms with some existing ones to evaluate our findings. The results not only validate the theoretical aspects of our algorithms, but also reveal our algorithms are feasible and practical in reality.

The organization of the paper is as follows: we introduce some background knowledge regarding the edge-based CDNs in Section 2, and present the caching notation and analyze the problem in Section 3. After that, we propose our optimal pro-caching algorithm in Section 4 and competitive re-caching algorithm in Section 5, which are followed by the combination of both algorithms in Section 6 and the empirical studies to validate the findings in Section 7. Finally, we survey on some related work in Section 8 and conclude the paper in the last section.

2 Background Knowledge

In this section, we introduce some background knowledge regarding the edge-based CDN and rephrase the problem in a more formal way that motivates this research.

A CDN is typically composed of an *origin sever* and a set of geographically distributed cache servers (aka surrogate servers) to deliver contents to end users at minimum cost. The origin server is a powerful storage system that is

1. Alibaba has been deploying its edge services around its AliCloud, which adopts a flexible charging model that allows the cloud and its edge to have the same charge rate for coordinating the computation between the cloud and its edge [20].

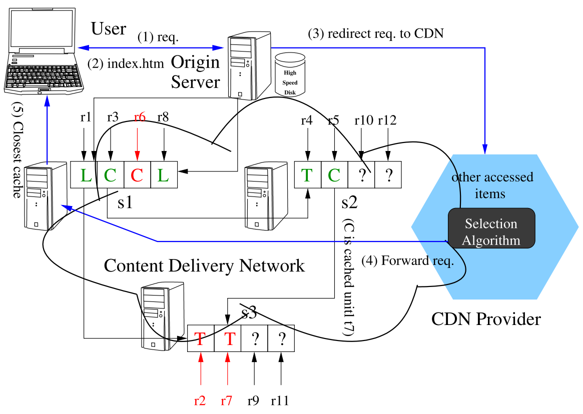


Fig. 1. Architecture and request routing in a CDN environment where three caching servers (squares) are fully connected by an edge network with a shared data item initially located at the origin server, which can then be cached at s_1 via a pull operation to load ("L"), thereafter, transferred ("T"), replicated, cached ("C"), or deleted to satisfy request stream $\mathcal{R} = < r_1, \ldots, r_{12} >$ in time order with minimum cost.

usually managed by content provider to host all the data items and/or their metadata for delivery while the cache servers are surrogates that replicate these data items from the origin server and cooperate with each other to improve the delivery efficiency. A typical architecture of CDN is shown in Fig. 1 where the components of CDN as well as the request routing mechanism are illustrated:

1) user makes a request r_1 by specifying its URL in his/her web browser, which is directed to the origin server;
2) upon receiving the request, the origin server only returns the basic content (e.g., index page) that it can serve;
3) while for other frequently accessed items, the origin server redirects the request to the CDN provider for further processing;
4) as soon as the request is received, the CDN provider designates the 'closest' cache server s_1 according to its *selection algorithm* to provide those requested item;
5) the designated cache server s_1 serves request r_1 directly if the requested data item is cached, otherwise, it gets the item from the origin server and caches it locally with a time-to-live (TTL) value for future re-uses.

As shown in Fig. 1, it is CDN that handles user's traffic, not the origin server, since the cache servers are always populated with and pass around the data items from the origin server. Basically, for the CDN, there are two distinct *content outsourcing* practices inherited from traditional CDN: push-network and pull-network. *Push-network* proactively pushes content from the origin server to a group of cache servers for efficient delivery at minimum cost while *pull-network*, whether or not there is collaboration between peering surrogates for the data discovery in cache miss, is instead reactive by nature wherein a data item is cached only when it is requested on demand.

The study in this paper devotes to the data delivery for both outsourcing networks with minimizing the total cost as an ultimate goal. Since updated data items are always outsourced to delivery network from the origin server, the outsourcing costs in both networks can be viewed as fixed constants with equal value. Consequently, we can only concentrate on the optimization of the caching schedule in the delivery network. To this end, the edge-based architecture as shown in Fig. 1 should be extended to including the functionality that allows the shared data item to be cached in a group of cache servers, according to an optimized schedule for serving request sequences made proactively or reactively with minimum total cost.

By following the example in Fig. 1, we further demonstrate how the edge-based CDN works in our case. Suppose another request r_2 is made to the same data item after r_1, it is first directed to the closest cache server s_3. Unfortunately, the item is not presented in that server. Thus, in a traditional case, the cache server would forward the request to the site's origin server. However, in our case, the item is transferred from s_1, which is represented by "T" in the figure, reducing the traffic to the origin server. After that, r_3 is coming, it is still served by the copy cached in s_1, denoted by "C". By the same argument, the following requests, $r_4, \ldots, r_{12}$ are satisfied either by transfer or by caching. Note that the red color indicates that the corresponding cached data item is deleted after being accessed. As such, the next request at the same server should be served by a transfer (e.g., $r_7@s_3$). The goal is to satisfy the request stream with minimum cost when the semi-homogeneous cost model is assumed.

3 Caching Schedule Problem

With the understanding of the edge-based CDN, we can define an instance of the caching schedule problem as a 4-tuple $\mathcal{I} = (\mathcal{P}, \mathcal{R}, \Theta, \Omega)$, where

1) $\mathcal{P} = \{s^1, \ldots, s^m\}$ is a set of servers embedded in a fully connected network supplying a data item subject to requests.
2) $\mathcal{R} = < r_1, \ldots, r_n >$ is a request vector. Each request is made at a specified time and to a specified server, thus $r_i = (s_i, t_i)$ and the vector is ordered so that $t_i \leq t_{i+1}$. Note the use of subscripts for references versus superscripts for labels, e.g., $s_i = s^j \in \mathcal{P}$.
3) $\Theta = \{\lambda_{ij} : 1 \leq i \neq j \leq m\}$ is the set of non-negative transfer costs from s^i to s^j.
4) $\Omega = \{\mu_i : 1 \leq i \leq m\}$ is the non-negative cost per unit time to store the item on the indicated server. These rates do not vary with time.

To simplify boundary conditions, we extend the request vector to assume the data item is initially (at time t_0) cached at s^1 and the initial (dummy) request is $r_0 = (s^1, 0)$. Thus, the cost of serving the initial request is 0.

We define a data item *transfer* $Tr(s_i, s_j, x)$ from server s_i to s_j at time x, and indicate a copy of the data item is cached, or *held in cache* on server s from time x to y using the notation of $H(s, x, y)$. Note that in this model, we are not concerned with the problem of obtaining the pre-defined request stream as it is realistically solvable by using off-line profiling and/or historical logging. Also, we do not consider the characteristics of the requests such as their sizes, which may have impact on the QoS of data accesses, but not relevant to our goal—minimizing the total costs for service providers to serve pre-defined request stream $\mathcal{R}$ via caching and transfer as exemplified in Fig. 1.

Definition 1 (Schedule). *We say that a* schedule $\mathcal{S}$ *is any* minimal *set of caches and transfers satisfying*

1) *At least one server is caching the data item at any time $t, t_0 \leq t \leq t_n$.*
2) *The data item is available for r_j on s_j at time $t_j, 1 \leq j \leq n$. We assume transfer time is negligible, thus we can satisfy this by a transfer at time t_j. This assumption can be validated by tweaking the instance graph (discuss later) as shown in [25], and is thus often adopted in previous studies [25], [26], [27].*

For a given schedule $\mathcal{S}$, the *cost* of the schedule, $\mathcal{C}(\mathcal{S})$, is the sum of the cache and transfer costs in the schedule, which can be recursively define its cost as follows,

Definition 2 (Schedule Cost). *Suppose there are n requests in $\mathcal{S}$, and $\mathcal{S}_i$ represents the schedule of requests from r_1 to r_i, without loss of generality, we can assume $\mathcal{C}(\mathcal{S}_1) = 0$, then we have*

$$\mathcal{C}(\mathcal{S}_i) = \mathcal{C}(\mathcal{S}_{i-1}) + \mathcal{C}(\mathcal{S}_{i-1}, \mathcal{S}_i),$$

here, $\mathcal{C}(\mathcal{S}_{i-1}, \mathcal{S}_i)$ is the cost to serve request r_i from schedule $\mathcal{S}_{i-1}$ at time t_{i-1} to serve r_{i-1} via caching and/or transfer. Clearly, $\mathcal{C}(\mathcal{S}) = \mathcal{C}(\mathcal{S}_n)$.

We use Γ to represent the space of the schedules for $\mathcal{R}$, and then define optimal schedule, which is our goal to find,

Definition 3 (Optimal Schedule). *An optimal schedule $\mathcal{S}^*$ is the schedule $\mathcal{S}$ that satisfies*

$$\mathcal{C}(\mathcal{S}^*) = \min_{\mathcal{S} \in \Gamma}\{\mathcal{C}(\mathcal{S})\}.$$

Note that the requirement that a schedule is minimal does not imply that it is optimal.

The data caching problem with heterogeneous cost model is a variant of the *Rectilinear Steiner Arborescence* problem [28]. As such, it is believed to be NP-complete [27]. However, its formal proof still remains open. Fortunately, in some restricted settings, we can expect optimal solution to this problem. The following is a restricted cost model regarding the cache and transfer for which we can have fast optimal algorithms.

Definition 4 (Semi-Homogeneous Cost Model). *We assume the cache cost is heterogeneous, denoted by μ_i whose value is different from server to server, but transfer cost $\lambda_{ij} = \lambda$ is constant for all pairs of servers. For convenience we also assume that all requests occur at distinct times, so that $\forall i, t_i < t_{i+1}$.*

For the remainder of this paper we mainly consider the semi-homo model as this model well fits in with the actual situation where transfer cost of a unit of data between any pair of network nodes is fixed. As such, with this billing policy, the transfer cost of the shared data item in our problem is identical between any pair of network nodes. For quick reference, we summarize the frequently used symbols in Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TMC.2021.3108150.

4 A Proactive Off-Line Algorithm

Based on the problem definition, in this section we propose an efficient off-line algorithm, called *Proactive Caching (pro-caching)* algorithm, with $O(mn)$ time and space complexities (m: network size, n: sequence length). To this end, we first define two major concepts —*standard schedule* and *instance graph*, then make several key observations on this problem, and finally reduce the caching problem to a shortest path-based problem.

4.1 Standard Schedule and Instance Graph

We can cast a schedule as a directed graph where the edges are caching intervals and transfers, and the vertices are requests and end points of transfers. Note that since a schedule is a minimal set, it implies that this graph is a tree. If there is more than one path from r_0 to r_i then at the last juncture of paths, at least one of the entries must be a transfer which can be deleted without loss of service and thus such a graph cannot be minimal. Also, a schedule will contain no dead-end caches, that is no cache on a server beyond the last request or transfer time from that server.

In general transfers could occur at any time point, so to make the search discrete we use the following lemma.

Definition 5 (Standard Schedule). *We say that a schedule is a* standard schedule *if every transfer occurs at a request time t_i with either its output end or its input end on server s_i; that is, it starts or ends at a request.*

With this definition, it is not difficult to deduce that given a schedule $\mathcal{S}$, there exists at least one standard schedule $\mathcal{S}^{std}$ with cost $\mathcal{C}(\mathcal{S}^{std}) \leq \mathcal{C}(\mathcal{S})$.

Also noted previously that every schedule is a tree (e.g., Fig. 2), we can immediate have the followings.

Observation 1. *In any optimal standard schedule, each request r_i will be served by either the cache (i.e., the cached copy) on s_i or by a single transfer ending at r_i.*

Given this observation, we are now viewing a schedule from a space-time diagram of view, called *Instance Graph* as follows, where the edges are caching intervals or transfers, and the vertices are requests or end points of transfers.

Definition 6 (Standard Instance Graph). *We define a* standard instance graph *as a weighted directed graph $G = (V, E, W)$ where $V = \{v_{ij} : 0 \leq i \leq n, 1 \leq j \leq m\}$. Vertex v_{ij} corresponds to time t_i on server s^j. The edge set E consists of two subsets: the set of cache edges $C = \{(v_{ij}, v_{i+1,j}) : 0 \leq i < n, 1 \leq j \leq m\}$ and a set of transfer edges T. The transfer edges are bi-directional $T = \{(v_{ij}, v_{ik}), (v_{ik}, v_{ij}) : j \neq k, \text{and } (s_i, t_k) \in \mathcal{R}\}$. The edge weights W are defined as $W(e) = \lambda$ for edges $e \in T$ and $W(e) = \mu_j(t_{i+1} - t_i)$ for edges $(v_{ij}, v_{i+1,j}) \in C$.*

An example of a standard schedule in the instance for a stream of requests is shown in Fig. 2 where the caching cost and the transfer cost are $1.4\mu_1 + 0.2\mu_2 + 3.2\mu_3 = 8.4$ and $4\lambda = 20.0$, respectively, here $\mu_1 = 1, \mu_2 = 3, \mu_3 = 2$ and $\lambda = 5$.

Notice that a request r_i in the instance will correspond to vertex v_{i,s_i} in the instance graph. For convenience we will often refer to request vertices r_i. All other vertices we call

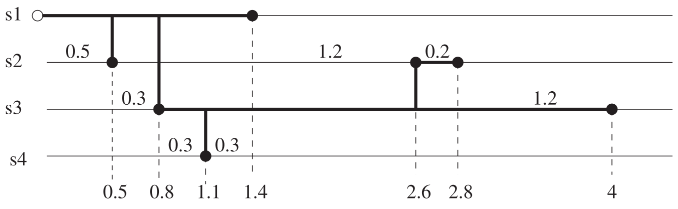


Fig. 2. An example of a standard schedule (shown in bold lines) for a predefined request stream (solid dots along time-line). Vertical lines represent transfers that end on requests. The optimal cost is $1.4\mu_1 + 0.2\mu_2 + 3.2\mu_3$ (caching cost) $+4\lambda$ (transfer cost) $= 1.4 + 0.6 + 6.4 + 20.0 = 28.4$ ($\mu_1 = 1, \mu_2 = 3, \mu_3 = 2$ and $\lambda = 5$).

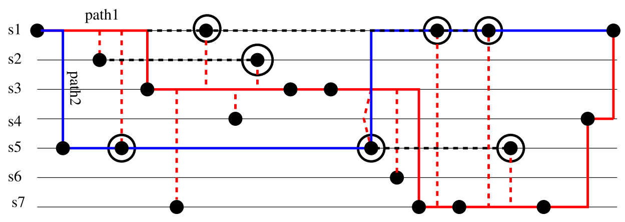


Fig. 3. An example to show different paths from $r_0 \rightarrow r_n$ could miss different sets of request vertices. Each of the missed requests in cycles can be served either by a cache or by a transfer.

intermediate vertices. The set of vertices v_{i*} induce a subgraph that is a bi-connected star centred on the request vertex r_i (e.g., see how server s^3 at t_4 connects to other servers in Fig. 4).

4.2 Proactive Caching Algorithm

We now show how to solve the caching problem by reducing it to a simple shortest path problem. To motivate this approach, we observe that any standard schedule $\mathcal{S}$ can be represented as a sub-graph $G' = G[\mathcal{S}] \subseteq G$ of the standard graph, with cost $\mathcal{C}_G(G') = \mathcal{C}(\mathcal{S})$ where $\mathcal{C}_G(G')$ is the sum of the edge costs in G'.

As noted previously, any schedule is a tree there will be a unique directed path $r_0 \rightarrow r_n$ in the schedule. Alternatively, for any $r_0 \rightarrow r_n$ path in G, we can create a schedule by adding an edge of cost at most λ for each r_i not in the path. Since different paths would cover different sets of request vertices, the cost added would be different for different paths as shown in Fig. 3, where *path1* and *path2* cover different sets of request vertices.

Suppose that we modify G by subtracting λ from the weight of each edge ending on a request vertex. Now given any $r_0 \rightarrow r_n$ path P we can create a schedule of cost $n\lambda$ plus the cost of P in this modified graph. Note that λn is independent of the path P. Unfortunately, this schedule may not be optimal over the set of schedules containing P as some of the requests not on the path could be more cheaply served by caching than by transfer. This is evidenced by those request vertices in cycles in Fig. 3. Thus, this schedule construction is not sufficient.

To address this issue, we have to analyze the reduction of the edge weights in the standard graph. To this end, we define two useful concepts *server caching cost* and *forward cost* for vertex v_{ij}, corresponding to a request or a non-request, in the graph.

Before defining the concepts, we first define some notations. For v_{ij}, we denote the *previous request index* to be

$$p(v_{ij}) = \begin{cases} \max\{k: & k < i, v_{kj} \text{ is a request vertex}\} \\ \Lambda & \text{if no request on } s^j \text{ prior to } t_i \end{cases}.$$

Similarly, we denote the *next request index* to be

$$\eta(v_{ij}) = \begin{cases} \min\{k: & k \geq i, v_{kj} \text{ is a request vertex}\} \\ \Lambda & \text{if no following request on } s^j \end{cases}.$$

Note that if a vertex is a request vertex r_i then $\eta(r_i) = i$.

Then with these notations, we can define the server caching cost and the forward cost for v_{ij} (again, v_{ij} could be a non-request vertex) as follows,

Definition 7. *The server caching cost for v_{ij} is*

$$\sigma(v_{ij}) = \begin{cases} +\infty, & p(v_{ij}) = \Lambda \\ \mu_j(t_i - t_{p(v_{ij})}), & \text{otherwise} \end{cases},$$

and similarly, the forward cost for v_{ij} to serve the next request $\eta(v_{ij})$ on the same server is

$$\delta t_{v_{ij}} = \begin{cases} +\infty, & \eta(v_{ij}) = \Lambda \\ \mu_j(t_{\eta(v_{ij})} - t_i), & \text{otherwise} \end{cases}.$$

As such, for request vertices, $\delta t_{r_i} = 0$.

Accordingly, we can immediately have the following definition:

Definition 8 (Marginal Request Cost). *The marginal cost of request r_i is $b_i = \min\{\lambda, \sigma(r_i)\}, 1 \leq i \leq n$. We define the total cost $B = \sum_{i=1}^{n} b_i$, which is the lower bound of the cost to satisfy the request vector $\mathcal{R}$.*

With these concepts, we develop our new idea that, instead of subtracting λ from the weight of each edge ending on a request vertex v_{ij}, subtracts b_h or $b_h - \delta t_{v_{ij}}$, depending on whether v_{ij} is a request vertex or not, where $h = \eta(v_{ij})$, meaning the next request after v_{ij} on server s^j is r_h. However, this subtraction is not necessary to apply to the edges ending on every non-request vertex. Only should those in a specific region (i.e., shadow region defined later) that is covered by b_h from r_h (i.e., $\delta t_{v_{ij}} < b_h$) is considered since the weight of each edge ending on such a vertex is affected by subtracting b_h from the weight of edges ending on r_h.

To be more clearly, we define a directed weighted graph $G_r = (V, E_r, W_r)$ derived from the standard graph $G = (V, E, W)$ that will support the desired reduction. We use the same vertex set V as well as $E_r = E$.

Notably, it is sufficient to set $E_r = E$, although some efficiency gain might be possible by reducing the number of edges in E_r as indicated in the following lemma,

Lemma 1. *A transfer edge (v_{ik}, v_{ij}) can never be part of any optimal schedule if $\sigma(v_{ij}) < \lambda$.*

Proof. If a schedule contains such a transfer edge, it can be replaced by a cache from the request prior to v_{ij} on the same server at reduced cost. □

Based on Lemma 1 we can define the edge set of G_r by $E_r = E \setminus \{(v_{ik}, v_{ij}) : \sigma(v_{ij}) \leq \lambda\}$. Note we allow equality in

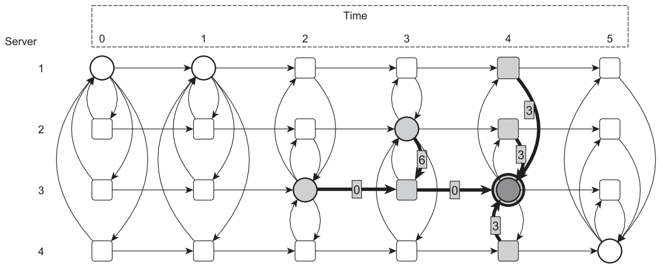


Fig. 4. An illustration of the shadow for a request at time t_4. Requests are shown as circles, and vertices of interest are highlighted. For this example we use $\lambda = 10$, $\mu_3 = 1$ and assume that $t_2 = 4$, $t_3 = 8$ and $t_4 = 11$. Thus, $b_4 = 11 - 4 = 7$ since 7 is less than 10. The shadow edges have labels showing their resultant costs.

this definition since in that case we can also use a cache instead of a transfer. Given G_r, we can formally define *shadow region* for request r_i as follows,

Definition 9 (Shadows). *For request r_i on server s^j, we define the transfer shadow set of edges as $S^T(r_i) = \{(v_{ik}, v_{ij}) \in T_r : r_i = \eta(v_{ij}), \delta t_{v_{ij}} < b_{\eta(v_{ij})}, k \neq j, k \leq m\}$. Similarly, we define the caching shadow set of edges as $S^C(r_i) = \{(v_{i-1,j}, v_{ij}) \in C_r : r_i = \eta(v_{ij}), \delta t_{v_{ij}} < b_{\eta(v_{ij})}\}$. Finally, $S(r_i) = S^C(r_i) \cup S^T(r_i)$.*

Based on this definition, we can observe that 1) in both cases, v_{ij} is a non-request vertex if $v_{ij} \in (r_{p(i)}, r_i)$ on s^j, and 2) the shadow sets are non-intersecting, i.e., $S(r_i) \cap S(r_j) = \varnothing, i \neq j$.

As discussed above, the idea is that for each request vertex r_i in G_r we reduce the weight of edges in a *shadow* region, where the shadow consists of edges in paths leading to r_i. The vertices falling in the shadow region of r_i (including r_i) could cache a copy to serve r_i via caching or transfer. Informally, the shadow region for a request is characterized by a region in the instance graph and the request could be served by the cached copies in that region. As such, according to our discussion, the weights of the incoming edges to that request have to be adjusted accordingly.

We show an example in Fig. 4 where four requests are made by four servers at different time. In this example, we use $\lambda = 10$, $\mu_3 = 1$ and assume that $t_2 = 4$, $t_3 = 8$ and $t_4 = 11$. Thus, $b_4 = 11 - 4 = 7$ of v_{43}, which is greater than $\delta t_{v_{33}} = 11 - 8 = 3$. As a result, the vertex v_{33} falls in the shadow of v_{43} and the weights of its incoming edges as well as those of v_{43} have to be adjusted according to (1) and (2). In the figure, the shadow edges have labels showing their resultant costs.

We can now define the weight function W_r for the edges of G_r. We will consider edges ending on a vertex v_{ij}. All edges leading towards r_h $(h = \eta(v_{ij}))$ within distance b_h (i.e., $\delta t_{v_{ij}} < b_h$) will have their costs reduced as follows.

First, we consider the transfer edges that have the form $e_r = (v_{ik}, v_{ij}) \in T_r$

$$W_r(e_r) = \begin{cases} \lambda, & h = \Lambda \\ \lambda, & \delta t_{v_{ij}} \geq b_h \\ \lambda - (b_h - \delta t_{v_{ij}}), & \delta t_{v_{ij}} < b_h \end{cases}. \qquad (1)$$

Observe that if $b_h = \lambda$ $(\delta t_{r_h} = 0)$ then the cost will be reduced to 0 on transfer edges ending on request r_h. Moreover, each distinct $\delta t_{v_{ij}}$ is added to the weight of its corresponding transfer edges as shown in Fig. 5. The edge weights between v_{ij} and r_h should be 0.

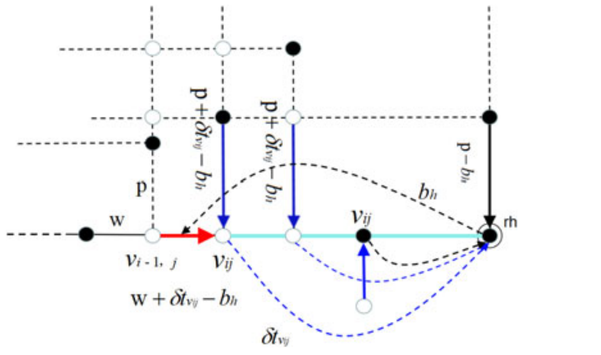


Fig. 5. Adjustments of the edges weights in G_r for request r_h. The black dots represent request vertices while the white dots are non-request vertices.

Let us first examine the transfer edges toward r_4 (request at v_{43}) in Fig. 4. In this case, $b_4 = 7$ and $\delta t_{v_{43}} = 0$, then according to (1), $W_r(e_4) = \lambda - (b_4 - \delta t_{v_{43}}) = 10 - (7 - 0) = 3$. Then we look at the case of v_{33}, a non-request vertex. As $\delta t_{v_{33}} = 3$, which is less than $b_4 = 7$, we have $W_r(e_3) = \lambda - (b_4 - \delta t_{v_{33}}) = 10 - (7 - 3) = 6$.

Then, for each vertex $v_{ij}, i > 0$ there will be one cache edge $e'_r = (v_{i-1,j}, v_{ij}) \in C_r$ ending on v_{ij}. Let $\hat{\mu} = \mu_j(t_i - t_{i-1})$, which is the weight of the corresponding edge in G

$$W_r(e'_r) = \begin{cases} \hat{\mu}, & h = \Lambda \\ \hat{\mu}, & \delta t_{v_{ij}} \geq b_h \\ \hat{\mu} - (b_h - \delta t_{v_{ij}}), & \delta t_{v_{ij}} < b_h < \delta t_{v_{i-1,j}} \\ 0, & \delta t_{v_{i-1,j}} \leq b_h \end{cases}. \qquad (2)$$

Note that no edge in G_r has negative weight, and all edges in $S^C(r_i)$ except the first (the red interval in Fig. 5) have cost zero (the cyan intervals in Fig. 5). This is because, as shown in Fig. 5, the weights of $e_r = (v_{i-1,j}, v_{ij})$ within b_h, i.e., $\delta t_{v_{ij}}$s, have been added to either the corresponding transfer edges $(v_{ik}, v_{ij}) \in T_r$ or the caching edge $(v_{i-1,j}, v_{ij}) \in C_r$. In the example, for both v_{33} and r_4, their forward costs (7 and 3, respectively) are not greater than $b_4 = 7$. Therefore, $W_r(e'_3) = W_r(e'_4) = 0$.

From Fig. 5, one can see that the weight of each edge in G_r is uniquely defined by (1) and (2) by which the path length can be correctly computed no matter how the path reaches r_h if a single entry into its shadow is performed.

Definition 10 (Reduced Graph Costs). *Given a sub-graph $G' \subseteq G_r$ we define the cost $\mathcal{C}_r(G')$ to be the sum of the edge costs in G'.*

Definition 11 (Multiple Entries). *Let $G' \subseteq G_r$ induced by the edge set E' and let $H = E' \cap S(r_i)$ for some request $r_i = (t_i, s_i)$ where $s_i = s^j$. We say that G' has multiple entries to the shadow $S(r_i)$ if there is an edge $e_r \in H$ with $v_{kj} \in e_r$ and an edge $(v_{hj'}, v_{hj}) \in S^T(r_i) \cap E'$ with $h > k$.*

We will need to take special care with multiple entry subgraphs which are paths that enter, leave and then re-enter a shadow, and schedules which have similar multiple entries into a shadow as shown in Fig. 6.

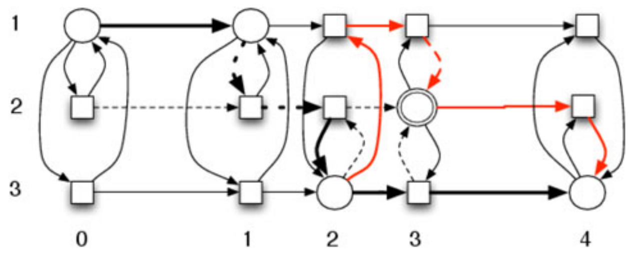


Fig. 6. Multiple entries: the numbers along x-axis and y-axis are time instance indices and server indices, respectively. The orange arrowed lines depict a path with multiple entries to the shadow of r_h. The dashed orange arrowed line indicates that the path leaves and re-enter the shadow.

Lemma 2. *Let P be a $r_0 \rightarrow r_n$ path in G_r with no multiple entries into any shadow. Then there exists a schedule $\mathcal{S}$ such that $\mathcal{C}(\mathcal{S}) = \mathcal{C}_r(P) + B$.*

Proof. We will add each $r_i = (s_i, t_i)$ inductively and its contribution b_i to B. The base starts with r_0 which is always in the path. For $i > 0$, there are two cases.

Case(i) no Intersection With Shadow. If $P \cap S(r_i) = \varnothing$, we note that $i < n$, since the path cannot reach a request unless it intersects its shadow. There are two sub-cases. If $b_i = \lambda$ then since $t_i < t_n$ there must exist a vertex v_{ik} on the path with $k \neq s_i$. We can thus add this request to the schedule for cost b_i using a transfer corresponding to the edge v_{ik} to v_{i,s_i}. If $b_i < \lambda$, then since by induction we have already scheduled $r_{p(i)}$ we can add r_i by adding a cache from the previous request with cost b_i.

Case(ii) Single Entry to Shadow. When the path intersects a shadow in a contiguous subset, b_i covers the reduction of costs in $P \cap S(r_i)$ due to definitions (1) and (2) (i.e., each edge in $S(r_i)$ has been reduced by b_i), or the cost of any edge required to reach r_i from this path. Note that this holds whether $r_i \in P$ or not. □

Lemma 3. *Let $\mathcal{S}$ be a standard schedule whose image in G_r contains no multiple entries into any shadow. Let P be the $r_0 \dots r_n$ path induced by $\mathcal{S}$. Then $\mathcal{C}(\mathcal{S}) \geq \mathcal{C}_r(P) + B$.*

Proof. If for request r_i there are no edges in $P \cap S(r_i)$ then the cost of connecting r_i to P in the schedule must be at least b_i (It may be greater in $\mathcal{S}$ if $b_i < \lambda$ and a transfer is used instead of a cache from the previous request on the same server.).

Otherwise, P intersects the shadow using only one entry. In this case only cache edges may be used to connect r_i to P (if needed) since it is only a single entry schedule, and so the cost of the edges in the shadow, including those in P, used in the schedule is at least b_i.

Subtracting these costs from the schedule $\mathcal{S}$ leaves us with the cost of at least P in G_r, then we have the conclusion. □

Theorem 1. *The optimal schedule has cost $\mathcal{C}(\mathcal{S}^*) = \min_{P \in G_r}\{\mathcal{C}_r(P) + B\}$, where each P is an $r_o \dots r_n$ path in G_r.*

Proof. Given a path P with multiple entries to a shadow $S(r_i)$, where i is minimum, we can obtain a path P' from P by two operations: 1) adding any necessary shadow cache edges from the end of the first entry to the last vertex in the intersection of the path with the shadow, and 2) deleting the edges previously used to make this connection (exemplified by the bold arrowed lines (v_{23}, v_{33}) and (v_{33}, v_{43}) in Fig. 6). Since all these additional shadow cache edges have cost zero by Definition (2), $\mathcal{C}_r(P') \leq \mathcal{C}_r(P)$. As the shadow sets are non-intersecting, any subsequent shadow in P' with multiple entries can be treated independently using this method, and so there exists a minimum cost path with no multiple entries to any shadow. From this result and Lemma 2, it follows that

$$\mathcal{C}(\mathcal{S}^*) \leq \min_{P \in G_r}\{\mathcal{C}_r(P) + B\}.$$

For any standard schedule $\mathcal{S}$ whose image in G_r has multiple entries to a shadow $S(r_i)$, where i is the minimum such occurrence in this schedule, there exists a schedule $\mathcal{S}'$ with cost $\mathcal{C}(\mathcal{S}') \leq \mathcal{C}(\mathcal{S})$ obtained by replacing the last transfer with required cache edges in the shadow from the end of the preceding entry. From the definition of the shadow, these cache edges will cost less than the transfer edge. This argument can be repeated until there is only one entry into the shadow $S(r_i)$. Given the shadow sets are non-intersecting, this can be applied to each subsequent shadow independently to obtain a schedule $\mathcal{S}''$ with no multiple entries, and $\mathcal{C}(\mathcal{S}) \geq \mathcal{C}(\mathcal{S}'')$. Let P be the $r_0 \dots r_n$ path induced by $\mathcal{S}''$. From this and Lmma 3 it follows that $\forall \mathcal{S}, \exists \text{path } P, \mathcal{C}(\mathcal{S}) \geq \mathcal{C}_r(P) + B$ and thus $\mathcal{C}(\mathcal{S}^*) \geq \min_{P \in G_r}\{\mathcal{C}_r(P) + B\}$. Together with the previous inequality, this completes the proof. □

Based on the description of this algorithm, we can easily obtain the best result in parallel for multiple data items in our caching context since we can apply the algorithm to each individual item independently without concerning with the cache capacity (again, the service cost reduction is our goal).

4.3 An Efficient Implementation

To obtain the cost of an optimal schedule, we need to only find the cost of a shortest $r_0 \dots r_n$ path in G_r and add B. G_r has $O(mn)$ vertices and edges and can be constructed in $O(mn)$ time. Since it is well known that shortest paths in directed acyclic graphs (DAGs) can be solved via topological sort in linear time, that is $O(mn)$ time in our case, we show this efficiency can also be attained in the semi-homo model.

Assume we have already found the cost of the shortest path from r_0 to each vertex at time t_{i-1}. At time t_i, the cost of reaching the request vertex $r_i = v_{i,s_i}$ will be the minimum of

1) the cost of reaching vertex v_{i-1,s_i} plus the cost of the cache edge (v_{i-1,s_i}, v_{i,s_i}).
2) the minimum over $j, s^j \neq s_i$ of the cost to reach $v_{i-1,j}$ + the cost of the cache edge $(v_{i-1,j}, v_{ij})$ plus the transfer cost λ to reach r_i.

The cost is dominated by 2), and is $O(m)$ at each i.

For each non-request vertex v_{ij} where $s^j \neq s_i$, the cost to reach v_{ij} is the minimum of the cost to reach $v_{i-1,j}$ plus the cache edge $(v_{i-1,j}, v_{ij})$ and the cost to reach $r_i + \lambda$. Once the cost of r_{i-1} has been computed these can be computed in $O(m)$ time also for each i.

One final consideration needs to be met when we want not just the cost of the optimal schedule but an actual schedule is required. In the first part of the proof of Theorem 1 we note that it is possible for the multiple entry path P to have the same cost as the final single entry path. This can occur because the edges that are removed from the path P outside of the shadow of r_{i-1} may also have cost zero if they are part of the shadow of other requests. To construct a schedule, we must ensure that the shortest path we construct has no multiple entries. This can easily be done by tie breaking when paths are of equal length based on the number of edges. The grid-like structure of the graph ensures that a path that leaves and re-enters a shadow will have more edges than one that stays in the shadow using only cache edges.

To complete the schedule construction, at each request we only need to find the closest connection to the path or the preceding request which can be done in time proportional to the number of edges. Thus, an optimal schedule can be obtained in $O(mn)$ time.

5 A Reactive Online Algorithm

In this section, we first describe an 2-competitive reactive caching algorithm for the online version of this problem and then show its tightness by giving a lower bound of the competitive ratio, which is at least 2, for any deterministic online algorithm for this problem. We finally analyze the complexity of this algorithm and describe briefly its implementation.

5.1 Reactive Caching Algorithm

The algorithm is built on a concept of *anticipatory caching* that, depending on the number of alive copies, allows the copy migrated to a sever (say s^j) to speculatively keep alive for another one or two period of $\Delta t_j = \lambda/\mu_j, 1 \le j \le m$, after it serves the most recent request at time t. Specifically, if there are multiple alive copies and the next request on s^j is arriving no later than $t + \Delta t_j$, it should be served by caching as the caching cost is no more than λ. Otherwise, when a single copy is left, its alive period is extended to $t + 2\Delta t_j$ to either serve incoming requests at most cost of 2λ or be transferred to the cheapest server if no request is found in $[t, t + 2\Delta t_j]$. After the alive period, the copy is not worthwhile to keep alive, and the request is served by a transfer from the cheapest server, instead.

By this way, we can enable the online algorithm to mimic the optimal off-line algorithm as close as possible. Without loss of generality, we assume $\mu_1 \le \mu_2 \le \cdots \le \mu_m$, and design an event-driven online algorithm, called *Reactive Caching (re-caching)* algorithm, which operates as shown in Algorithm 5.1 with an assumption that a data item is initially located at s^1 (the cheapest one).

In the algorithm, we use variables c, initialized by 1, to record the number of alive copies in the network, a counter array of $E[m]$, initialized by zero, to maintain the copy expiration information of each server in the network, e.g., $E[j] \leftarrow t$ indicates the copy on s^j will expire at t, and $t_{p'(j)}$ to specify the most recent time when a request is made on $s^j, 1 \le j \le m$ (`Line 2`).

Algorithm 1. ReCaching Algorithm

```
1:  /* a data item is initially located at $s^1$ */
2:  **Initialize:** $c \leftarrow 1; E[m] \leftarrow 0, 1 \le j \le m$;
3:  **if** *(request $r_i$ arrives $s^j$ at $t_i$)* **then**
4:      **if** $(E[j] = 0)$ **then**
5:          serve $r_i$ by a transfer from any $s^k, k \neq j$, who has an
            alive copy;
6:          $E[j] \leftarrow t_i + \Delta t_j$;
7:          $c \leftarrow c + 1$;
8:      **end**
9:      **else**
10:         **if** $(E[j] \neq 0)$ **then**
11:             serve $r_i$ by the copy on $s^j$;
12:             $E[j] \leftarrow t_i + \Delta t_j$;
13:         **end**
14:     **end**
15: **end**
16: **if** *(a copy expires on $s^j$ at $t$)* **then**
17:     **if** $(j = 1)$ **then**
18:         **if** $(c = 1)$ **then**
19:             $E[j] \leftarrow t + \Delta t_j$;
20:         **end**
21:         **else**
22:             drop the copy at $t$;
23:             $E[j] \leftarrow 0$;
24:             $c \leftarrow c - 1$;
25:         **end**
26:     **end**
27:     **else**
28:         **if** $(c = 1)$ **then**
29:             **if** $(E[j] - \Delta t_j = t_{p'(j)})$ **then**
30:                 $E[j] \leftarrow t + \Delta t_j$;
31:                 **if** $(E[j] - 2\Delta t_j = t_{p'(j)})$ **then**
32:                     $s^j$ performs a transfer to $s^1$;
33:                     drops the local copy;
34:                     $E[j] \leftarrow 0$;
35:                 **end**
36:             **end**
37:         **end**
38:         **else**
39:             **if** $(E[j] - \Delta t_j = t_{p'(j)})$ **then**
40:                 drop the local copy;
41:                 $E[j] \leftarrow 0$;
42:                 $c \leftarrow c - 1$;
43:             **end**
44:         **end**
45:     **end**
46: **end**
```

The algorithm is driven by two events—request arrival and copy expiration. When a new request r_i arrives s^j at t_i (create copies) (`Line 3`), we first check $E[j]$. If $E[j] = 0$, we then serve r_i by a transfer from any $s^k, k \neq j$, who has an alive copy, and then update $E[j]$ and c (`Line 4-8`). Otherwise, if $E[j] \neq 0$, we serve r_i by the copy on s^j, and then only update $E[j]$ (`Line 10-13`);

When a copy expires on s^j at t (drop copies), we check whether it happens on the cheapest server $(j = 1)$ or not $(j > 1)$ (`Line 16`). In the former case $(j = 1)$, we further look at if only one copy is left in the network. If $c = 1$, we extend the copy expiration time on s^j to another period of

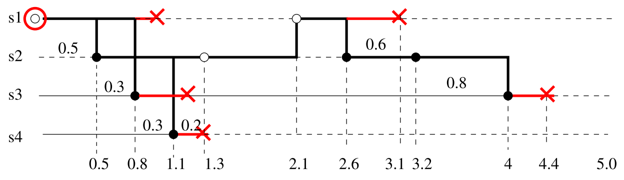


Fig. 7. An example of the online *re-caching* algorithm to illustrate a schedule for a sequence of 6 requests (black dots) where the red line indicates copy extension till to red cross when the copy is dropped. Note that there is a transfer from s^2 to s^1 at time $t = 2.1$ after $2\Delta t_2 = 1.6$ from $t = 0.5$.

Δt_j. Otherwise, we drop the copy at t and update $E[j]$ and c `(Line 17-26)`.[2]

While in the later case ($j > 1$), if only one copy is left ($c = 1$), we then check the local copy on s^j. If it has been alive for a period of Δt_j since it served the last request, $E[j]$ is updated, indicating the copy will be alive for another period of Δt_j `(Line 28-37)`. However, if the copy has been alive for $2\Delta t_j$, then s^j will perform a transfer to s^1, drop the local copy, and reset $E[j]$. Otherwise, if there are more than one copies alive in the network ($c > 1$), we examine $E[j]$ to see if the copy has been alive for a period of Δt_j since it served the last request. If so, we drop the local copy, reset $E[j]$, and update c `(Line 39-43)`.

An illustrative example of this algorithm is shown in Fig. 7 where the copy on s^j survives another period of time at most $\Delta t_j = \lambda/\mu_j$ for incoming requests, given $\Delta t_1 = 1.0, \Delta t_2 = 0.8, \Delta t_3 = 0.4$, and $\Delta t_4 = 0.2$.

5.2 Competitive Analysis

Based on the design of the algorithm, we can make the following observation to conduct its competitive analysis.

Observation 2. *During its execution, the online* re-caching *algorithm satisfies the following properties:*

1) *when multiple copies exists, no copy on $s^j, 1 \le j \le m$ can be kept alive for more than Δt_j;*
2) *where there is a single copy, except for the copy on s^1, no other copy on $s^j, 1 < j \le m$, can survive a period of time longer than $2\Delta t_j$ after it serves a request;*
3) *at any point of time, there is always a copy to serve the incoming request, either by caching or by transferring.*

With the above results, we now have the main theorem for the online case:

Theorem 2. *The online re-caching algorithm is 2-competitive.*

Proof. Let $C_A(r_i)$ define the cost of request r_i obtained by the re-caching algorithm and $C^*(r_i)$ be the optimal cost of request r_i gained by the optimal off-line algorithm. First, we analyze the cost ratio of request r_i by considering the following cases:

Case 1. If r_i is the first request on server s^j for $1 \le j \le m$, without loss of generality, we assume the first request r_1 is made on s^1, we have $C_A(r_1) = \mu_1\Delta t_1$, $C^*(r_1) = \mu_1\Delta t_1$, Thus $C_A(r_1)/C^*(r_1) = 1 < 2$.

Case 2. When a new request r_i arrives at s^j at time t_i, its cost is considered from four perspectives.

2. When multiple copies expire at the same time, only the one on s^1 is kept while others are dropped.

a) If $t_i \in [t_{p(i)}, t_{p(i)} + \Delta t_j]$ and $E[j] \ne 0$, then $C_A(r_i) = C^*(r_i) = \mu_j\Delta t_j \le \lambda$, we thus have $C_A(r_i)/C^*(r_i) = 1 < 2$.

b) If $t_i \in [t_{p(i)} + \Delta t_j, t_{p(i)} + 2\Delta t_j]$ and $E[j] = 0$, then $C_A(r_i) = \mu_j\Delta t_j + \lambda = 2\lambda$ and $C^*(r_i) = \lambda$, we have $C_A(r_i)/C^*(r_i) = 2$.

c) If $t_i \in [t_{p(i)} + \Delta t_j, t_{p(i)} + 2\Delta t_j]$ and $E[j] \ne 0, c = 1$, that is there is no new request r_i coming and no copy on any server other than s^j who has held the caching Δt_j, according to the algorithm, the data should be cached on s^j for one more Δt_j, as such $\lambda < C_A(r_i) \le 2\mu_j\Delta t_j \le 2\lambda, \lambda < C^*(r_i) \le 2\lambda$, then we have $C_A(r_i)/C^*(r_i) < 2$.

d) If $t_i > t_{p(i)} + 2\Delta t_j$ and $E[j] \ne 0, c = 1$, according to the algorithm, the data item should be transferred to s^1 after being held for $2\Delta t_j$. We thus count this cost 2λ into the service cost of r_i. Consequently, $C_A(r_i) = \mu_j \cdot 2\Delta t_j + 2\lambda + \mu_1\Delta t = 2\lambda + 2\lambda + \mu_1\Delta t$, $C^*(r_i) = \min\{2\lambda + \mu_j\Delta t, \lambda + \lambda + \mu_1(2\Delta t_j + \Delta t)\}$, then we have $C_A(r_i)/C^*(r_i) < 2$.

Suppose $\mathcal{C}(RC)$ is the total cost of the first n requests obtained by the re-caching algorithm and $\mathcal{C}(OPT)$ is the corresponding optimal off-line cost. We then have

$$\mathcal{C}(RC)/\mathcal{C}(OPT) = \left(\sum_{1\le i\le n} C_A(r_i) + m\lambda\right) / \sum_{1\le i\le n} C^*(r_i).$$

Therefore, we conclude

$$\lim_{n\to+\infty} \mathcal{C}(RC)/\mathcal{C}(OPT) = 2. \tag{3}$$

□

Next, we show that the competitive ratio of any deterministic online algorithm is at least 2 given the semi-homo caching model.

Theorem 3. *The competitive ratio of the semi-homo caching problem is at least $2 - o(1)$.*

Proof. We construct an instance to obtain a particular lower bound as follows: suppose there are two servers, and $\mu_1 = \delta$, $\mu_2 = 1$, and $\lambda = 1$, where $\delta < 1$ is a very small positive constant. For any deterministic caching algorithm, we assume an adaptive adversary who can produce a sequence of requests, depending on all the actions of the online algorithm up to the current time.

Without loss of generality, for any deterministic algorithm $\mathcal{A}$, it is reasonable to assume its caching time is l_i after the ith request is satisfied, and then the data item is transferred to s_1 for minimum caching cost in the cheapest server.

At the beginning, the data item is assumed to cache in server s^1. Then, request $r_1 = (s^2, 0)$ is coming to server s^2 at $t = 0$, which can be immediately satisfied with a transfer. then there are two cases for l_1:

Case 1.1. If $l_1 \ge 1$, the adversary has no other request to come. Thus, the competitive ratio is

$$\frac{\mathcal{C}(\mathcal{A})}{\mathcal{C}(OPT)} \ge \frac{1 + l_1 + O(\delta)}{1} \ge 2. \tag{4}$$

Case 1.2. Otherwise, if $l_1 < 1$, the adversary makes a follow-up request $r_2 = (s^2, l_1 + \tau)$, where τ is a very small interval. The caching time is l_2 for r_2. Similarly, there are also two cases:

Case 2.1. If $l_2 \geq 1$, there is no other coming requests, we then have the competitive ratio as

$$\frac{\mathcal{C}(\mathcal{A})}{\mathcal{C}(OPT)} \geq \frac{2 + l_1 + l_2 + O(\delta)}{1 + l_1} \geq \frac{3 + l_1}{1 + l_1} > 2. \quad (5)$$

Case 2.2. Otherwise, if $l_2 < 1$, the adversary makes the next request $r_3 = (s^2, l_1 + l_2 + 2\tau)$. Suppose the caching time is l_3 for r_3, we can make the same arguments as above with two cases until kth request arrives

$\vdots$

After the kth request is satisfied, there are still two cases:

Case k.1. If $l_k \geq 1$, there is no other incoming requests. Then the competitive ratio is

$$\begin{aligned}\frac{\mathcal{C}(\mathcal{A})}{\mathcal{C}(OPT)} &\geq \frac{k + l_1 + l_2 + \cdots + l_k + O(\delta)}{1 + l_1 + l_2 + \cdots + l_{k-1}} \\ &\geq \frac{k + 1 + \sum_{i=1}^{k-1} l_i + O(\delta)}{1 + \sum_{i=1}^{k-1} l_i} > \frac{2k}{k} = 2. \end{aligned} \quad (6)$$

Case k-2. Otherwise, if $l_k < 1$, the next request $r_{k+1} = (s^2, \sum l_i + k\tau)$ is made. Then the competitive ratio is

$$\begin{aligned}\frac{\mathcal{C}(\mathcal{A})}{\mathcal{C}(OPT)} &\geq \frac{k + 1 + \sum_{i=1}^{k} l_i + O(\delta)}{1 + \sum_{i=1}^{k} l_i} > \frac{2k + 1 + O(\delta)}{k + 1} \\ &> 2 - 1/(k + 1). \end{aligned} \quad (7)$$

Note that if we assume that the data item is always cached on the cheapest server s^1, the "$=$" relationship is held, otherwise, the "$>$" is obtained. By now, we have established a lower bound of $2 - o(1)$ for the competitive ratio of this problem, which implies there is no deterministic algorithm that can do better than this ratio. □

This theorem demonstrates that the competitive ratio of the proposed algorithm is fairly tight, compared to its lower bound. As with the off-line pro-caching case, the designed online re-active algorithm can be also applied to the multiple data items, each being scheduled independently.

5.3 An Efficient Implementation

The implementation of the algorithm is straightforward by following its description in Section 5.1 where a counter array $E[m]$ is maintained to record the copy expiration information for each server in the network. Specifically, for each incoming request, all its serving copy information in $E[m]$ can be manipulated within time $O(1)$. As such, if the index of one of the alive copies is maintained in a pointer for efficient checking, then, the time complexity of the algorithm incurred by one request is also a constant, which is highly efficient.

However, this complexity analysis is somewhat idealized as we lack the notion of the amortized overhead of dropping time-out copies for each request. Fortunately, the copy-timeout event processing can be implemented as a background daemon running in parallel with the request event processing. As such, it does not hurt the time efficiency of request services.

6 Hybrid Caching Algorithm

Although both of the algorithms presented in previous sections are efficient, they have demerits of their own. In particular, the pro-caching algorithm is optimal in terms of cost reduction, but it relies on pre-defined request sequence, which is not always feasible in practice. In reverse, the re-caching algorithm, though not requiring the availability of pre-defined sequence, lacks a global view of the requests, rendering it sub-optimal for cost reduction. As such, to have the complementary advantages of both algorithms, we deliberately refactor them to combine as a single hybrid caching algorithm, called *Hybrid Caching (hy-caching)* algorithm as shown in Algorithm 2.

Algorithm 2. Hybrid Algorithm (HyCaching)

```
1: run the pro-caching algorithm on $\mathcal{R}$ and record the pro-caching schedule $\mathcal{S}^*$;
2: Initialize: $E[j] \leftarrow 0, 1 \leq j \leq m$;
3: for (each new request $r_i$ arrives $s^j$ at $t_i$) do
4:   if ($r_i \in \mathcal{R}$) then
5:     serve $r_i$ either by the copy created by $\mathcal{S}^*$ or by a caching from $s^j$, whose copy is maintained by re-caching in extension $\Delta t_j$, whichever is cheaper;
6:     update $E[j]$ by the caching schedule on $s^j$ based on $\mathcal{S}^*$;
7:   end
8:   if ($r_i \in \mathcal{R}' \setminus \mathcal{R}$) then
9:     if ($r_i$ falls on a caching interval of $\mathcal{S}^*$) then
10:      serve $r_i$ immediately by caching at cost 0;
11:    end
12:    else
13:      if ($E[j] = 0$) then
14:        serve $r_i$ by a transfer from $\mathcal{S}^*$, who has an alive copy;
15:        $E[j] \leftarrow t_i + \Delta t_j$;
16:      end
17:      else
18:        serve $r_i$ by the copy on $s^j$ or by a transfer from $\mathcal{S}^*$, whichever is cheaper;
19:        $E[j] \leftarrow t_i + \Delta t_j$;
20:      end
21:    end
22:  end
23: end
24: if (a copy expires on $s^j$ at $t$) then
25:   drop the local copy;
26:   $E[j] \leftarrow 0$;
27: end
```

Suppose the data item is located at s^1 (the cheapest one), for a predicted sequence $\mathcal{R}$ and a true online sequence $\mathcal{R}'$, the algorithm is roughly composed of two phases. In the first phase we conduct a pro-caching schedule in which the pro-caching algorithm is run on $\mathcal{R}$, then we record the pro-caching schedule $\mathcal{S}^*$.

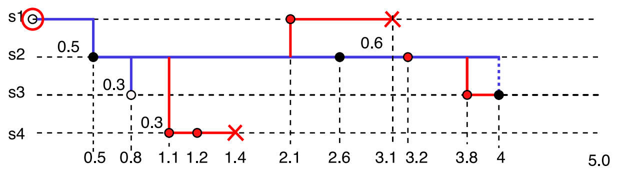


Fig. 8. An example of the *hy-caching* algorithm to illustrate a caching schedule for a sequence of 9 requests, where the black dots are correctly predicted requests, the white dots are not correctly predicted, and the red dots are the requests missed in the prediction. The blue lines show the *pro-caching* schedule for severing the predicted requests and the red lines indicate how the missing predicted requests are severed by the online *re-caching* schedule. Note that in this example, $\Delta t_1 = 1.0, \Delta t_2 = 0.8, \Delta t_3 = 0.4$ and $\Delta t_4 = 0.2$.

In the second phase, we run the re-caching algorithm in which as before we use a counter array of $E[m]$, initialized by zero, to maintain the copy expiration information of each server, $1 \leq j \leq m$ `(Line 2)`.

When a new request r_i arrives s^j at t_i (create copies), we first check if r_i has been predicted, if so, we serve r_i either by the copy pre-determined by pro-caching $\mathcal{S}^*$ or by a caching from s^j, whose copy is maintained by re-caching in extension Δt_j, whichever is cheaper, then update $E[j]$ based on $\mathcal{S}^*$ `(Line 4-7)`. Otherwise, if it is a true request, but not predicted (`Line 8`), then we see if r_i can be served by $\mathcal{S}^*$, if so, r_i can immediately be served by caching for free `(Line 9-11)`. Otherwise, if there is no copy cached on s^j, then r_i will be served by a transfer from $\mathcal{S}^*$, who has an alive copy, and update $E[j]$ `(Line 13-16)`, or else if there is a copy, then r_i is served by the copy on s^j or by a transfer from $\mathcal{S}^*$, whichever is cheaper, and then $E[j]$ is updated `(Line 17-20)`.

When a copy expires on s^j at t (drop copies), we can simply drop the local copy and reset $E[j]$ `(Line 24-27)`.

Remarks. Essentially, the algorithm is designed by improving the idea of the re-caching algorithm with the aids of pro-caching schedules on predicted sequence. As such, it allows the combined algorithms to mutually optimize each other in an online fashion with results being highly relied on the accuracy of the prediction. Specifically, when the prediction is 100% accurate, the incoming requests are severed at most $\mathcal{C}(\mathcal{S}^*)$. Otherwise, extra costs for the requests in $\mathcal{R}' \setminus \mathcal{R}$ need to be added while the counted costs for the requests in $\mathcal{R} \setminus \mathcal{R}'$ are wasted. Given this complication, the competitive ratio of the hy-caching algorithm is hard to analyze in general. We will study this issue experimentally.

An illustrative example of the hy-caching algorithm is shown in Fig. 8 where a predicted sequence of requests is $\mathcal{R} = \{v_{0.5}, v_{0.8}, v_{2.6}, v_4\}$ and the real sequence is $\mathcal{R}' = \{v_{0.5}, v_{1.1}, v_{1.2}, v_{2.1}, v_{2.6}, v_{3.2}, v_{3.8}, v_4\}$. Then, according to the algorithm, the requests in $\mathcal{R}$, both correctly (block dots) and incorrectly predicted (white dots), are satisfied by the optimal pro-caching schedule (blue lines) while the real requests in $\mathcal{R}' \setminus \mathcal{R} = \{v_{1.1}, v_{1.2}, v_{2.1}, v_{3.2}, v_{3.8}\}$ (red dots) are satisfied by the re-caching schedule (red lines) as they are missing in the prediction. Also, both algorithms are beneficial to each other, say, it is not necessary for the re-caching algorithm to maintain an alive copy in the course of service as it can be provided by the pro-caching algorithm as shown for the transfer from s_2 to $v_{3.8}$. Reversely, if a re-caching schedule of a missed request (i.e., $v_{3.8}$ with time extension $\Delta t_3 = 0.4$) is cheaper than an optimal transfer schedule (i.e., the transfer from s_2 to v_4), then it can be replaced with a new re-caching schedule (i.e., the caching from $v_{3.8}$ to v_4 at s_3) to further minimize the cost.

7 Performance Evaluation

In this section, we conduct a trace-based simulation study to evaluate the proposed algorithms and show how they behave in practice with different configurations. As in [25], we did not take into account some properties and features of the network platform in the simulation as these properties can be modeled into the service cost, which is the focus of this research.

7.1 Experimental Setups

Our trace-based study is based on an open dataset to compare the cost-efficiency between our algorithms and different baselines with respect to different values of λ and μ_i in a simulated edge-based CDN, which is designed by following the model described in Fig. 1.

Dataset. The trace data we used is the largest open international mobile network dataset collected using the MONROE platform spanning across 6 countries, 27 mobile network operators, and 120 measurement nodes [29]. Reasonably, we selected the connection requests in the dataset to mimic the accesses to a shared data item in edge-based CDN, which includes "request node id", "request time", and other relevant information whereby a request vector $\mathcal{R}$ and a set of servers $\mathcal{P}$ can be extracted.

Baselines. The proposed algorithms for both online (*ReCaching*) and off-line (*ProCaching*) cases as well as their combination *HyCaching* are efficiently implemented and compared with some often-used baseline algorithms to show their respective cost-efficiencies.

1) *Greedy:* An offline greedy algorithm, which is to start with the last request on each server and finds the one with the lowest service cost as the solution to the next request [30].
2) *Min-Cost Always Online (MCAO):* An intuitive online algorithm, which is to keep an active cached copy all the time on the minimal-cost server to serve all the requests on other servers by transferring.
3) *AC3:* An extension of 3-competitive online algorithm in [24] that can work under the semi-homo model.
4) *Online Greedy (OGreedy):* An online greedy algorithm, which is designed to keep only one cached copy across all the servers. The locally cached copy is deleted immediately after it is transferred to a remote server where the next request is made.

Parameters. The simulator is configured by several parameters, including the size of the network (the number of servers), the caching and transfer cost (λ and $\mu_j, 1 \leq j \leq m$), and other available resources. It also accepts as an input a sequence of request demands that are made for a shared data item. In our studies, we assume the sequence is either presumably known in advance or generated in an online fashion, and each request is characterized by 2-element tuple $< s, t >$ as described in Section 3.

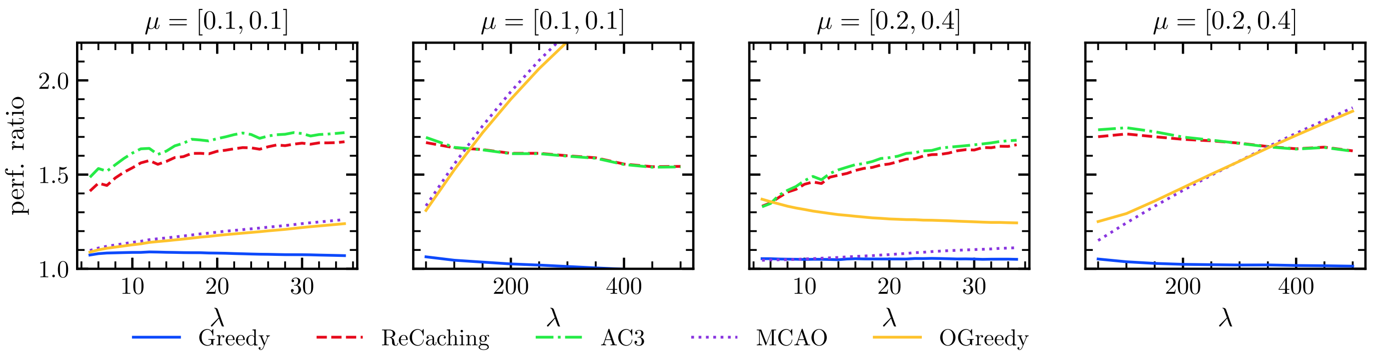


Fig. 9. How the costs of baselines are changed with respect to different λ where x-axis and y-axis are λ and performance ratio, respectively. The first and the fourth sub-figures are results for homo cost model, and others are for semi-homo cost model. The number of requests is 1000.

We took the billing policy of *Google Cloud Platform* (GCP)[3] as a reference to model the cache cost of servers. For instance, "n1-standard-8" costs 61¢ per hour (i.e., 0.16¢ per second) [31]. Therefore, we made uniform distribution as the value of per second cost μ, ranging from $U[0.1, 0.1]$ (homo cost model) to $U[0.4, 0.8]$ (semi-homo cost model). Based on *Google Cloud CDN* pricing ($5 - 20$ ¢ per GB), we further set the value of λ in $[5, 35]$ as this interval is expected to cover a range of practical cases with respect to the given μs. Meanwhile, to fully study the cost-efficiency of the algorithms, a thousand of requests and tens of their corresponding servers are randomly selected from the dataset as the configuration for the experiments. Moreover, to reach our goal, we defined a *performance ratio* (the lower, the better)

$$\rho = \frac{\mathcal{C}(\mathcal{A})}{\mathcal{C}(ProCaching)}, \tag{8}$$

to measure the proprieties of the algorithms in terms of *relative* cost-efficiency and scalability with respect to request workloads, here, $\mathcal{C}(X)$ denotes the sum of the cache and transfer costs in the algorithm solution, $\mathcal{A}$ denotes all the compared algorithms, and *ProCaching* represents the offline optimal algorithm.

Notably, all experiments are conducted under Linux Ubuntu 20.04 running on Intel(R) Core(TM) i9-10900K CPU @ 3.70GHz with 64GB Memory and 20MB L2 Cache.

7.2 Evaluation Results

In this section, we present our numerical results of the experiments. We first study the pro-caching and re-caching algorithms with respect to different model parameters, and then investigate the performance of their combination.

7.2.1 ProCaching and ReCaching

Impact of $\lambda\&\mu$. First, we studied the impact of $\lambda\&\mu$ on the costs of both proposed algorithms. We set up 5 different groups of experiments, including 2 groups with homogeneous μ and the other 3 groups with heterogeneous μ, with each group investigating how the costs of serving 1000 requests are changed for each uniform μ with the increase of λ from 5 to 35.

From Fig. 9, we can observe as λ grows up, the performance ratios of the compared online algorithms are increasing accordingly while the off-line *Greedy* exhibits the best performance in a relative stable fashion as in this case, the produced off-line schedule remains close to the results of *ProCaching*.

To measure their relative performance, we also compared *ReCaching* and *AC3* as shown in the figure where *ReCaching* is consistently better than *AC3* for both homo and semi-homo cost models. These results demonstrate the advantages of *ReCaching*, it can not only exploit the heterogeneity of the caching cost model to reduce the overall service cost but also overcome the inefficiency of *AC3* in homogeneous cases as the competitive analysis illustrates. Although the worst performances are guaranteed given their online nature, the actual performance of both algorithms *ReCaching* and *AC3* are not that impressive in our experiments, compared to *MCAO* and *OGreedy*.

This is because both MCAO and OGreedy maintain only one copy in the network. As such, when λ is relatively small, a large number of transfers would result in better performance than ReCaching, which heavily relies on caching to serve more requests than both MCAO or OGreedy. Thus, ReCaching is inferred to have more benefits when the transfer cost is increased.

We validate this inference by increasing λ in the last experiment. As shown in Fig. 9, the performance of ReCaching becomes stable within our bound as λ increases, which is different from OGreedy and MCAO, both exhibit unboundedly lousy performance. This is because high transfer cost will suppress data movements in the network, which in turn results in many caching operations in favor of the ReCaching algorithm. On the other hand, due to the single-copy nature, both OGreedy and MCAO have to perform a transfer for each incoming request if it is not made at the copy-cached server.

To further validate our explanation, we fully investigate how the cost is composed. To this end, we broke down the total cost for each compared algorithm in terms of caching and transfer when λ is increased from 100 to 500 with respect to μs in different uniform intervals. As shown in Fig. 10, the caching cost is barely changed for both MCAO and OGreedy while increasing in ReCaching and AC3. This observation confirms our conclusion that both ReCaching and AC3 can effectively take advantage of the caching operations to reduce the overall costs. Therefore, the ReCaching algorithm is a stable and theoretically bounded algorithm.

3. https://cloud.google.com/

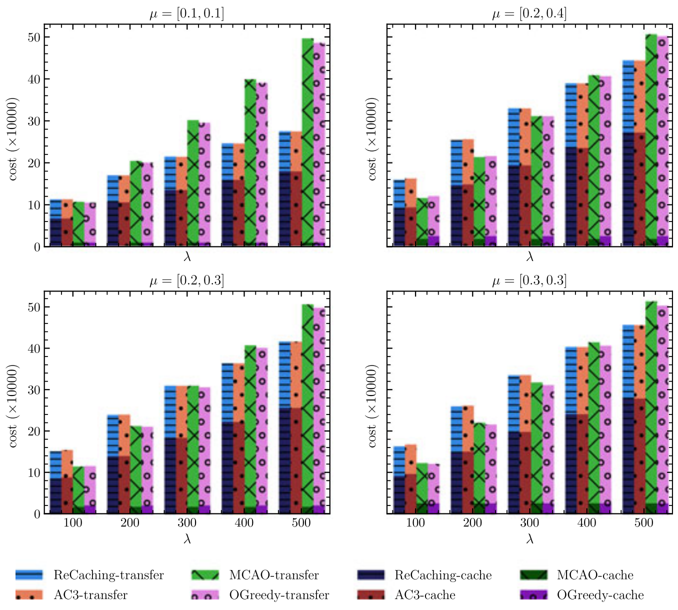


Fig. 10. The comparisons of cost breakdowns (caching and tranfer) between different compared algorithms when λ is increased from 50 to 500.

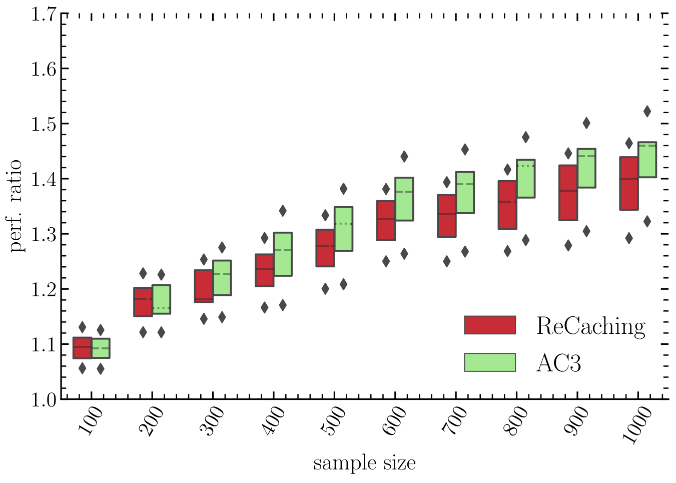


Fig. 11. How the costs of baselines are changed with respect to different sample sizes where x-axis and y-axis represent sample size and performance ratio, respectively ($\mu = [0.4, 0.5]$).

Scalability. To study the scalability of *ReCaching* and *AC3* with respect to request workloads, we gradually increase the number of requests from 100 to 1000 and observe how their performance ratios are changed with the number of involved servers being increased from 20 to 40. To this end, we conducted 5 groups of experiments, each setting the values of λ as $\{10, 15, 20, 25, 30\}$.

The numerical results on comparison of different groups are described in Fig. 11 where each box represents the statistics across the five λ-values for a fixed-sized sequence of requests. From the figure, in general the performance ratios of both algorithms initially increase and gradually become stable, and in particular *ReCaching* is consistently better than *AC3* with a trend of enlarging the performance gaps as the size of request samples increases, which is aligned with our previous results. We can attribute these phenomenon to the proprieties of the compared algorithms. When there is only one copy left, *ReCaching* has one more λ caching than

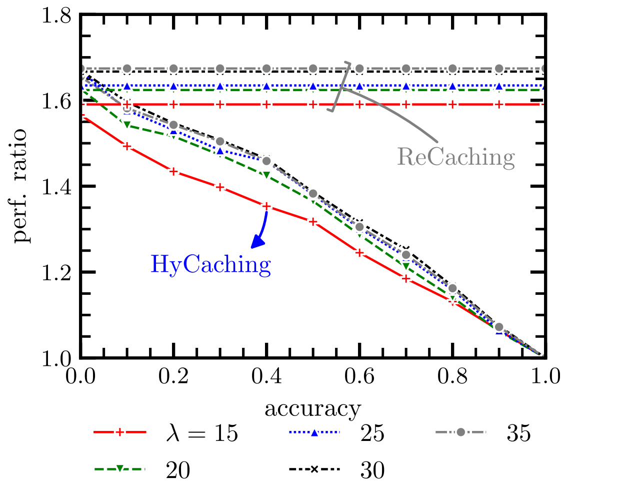


Fig. 12. How the performance ratios of *HyCaching* are relatively changed with respect to different prediction accuracies where x-axis and y-axis are accuracy ratio and performance ratio, respectively. The number of requests is 1000 and μ = [0.3,0.4].

AC3, some requests can be served by leveraging the extra caching in *ReCaching*, while in *AC3* they need to be served via transferring. Therefore, when requests becomes intensive, more of them are likely served by the extra caching. As such, *ReCaching* is expected to perform better than *AC3*, which means as the request workload increases, *ReCaching* is more scalable than *AC3*.

7.2.2 *HyCaching*

In this section, we evaluated the performance of *HyCaching* that combines both *ProCaching* and *ReCaching*. We first compared *HyCaching* with *ReCaching* to see how effectively the combination improves over *ReCaching*, and then disclosed how its component algorithms—*ProCaching* and *ReCaching*—behave under the hood. To this end, we deliberately constructed a set of *predicted* request sequences, each with different accuracies, by randomly replacing a number of requests in an *actual* sequence.

Performance Comparison. We compared the perf. ratio (the lower, the better) between *HyCaching* and *ReCaching* with respect to different λs when transfer cost λ is varied from 5 to 35. As shown in Fig. 12, the perf. ratio of *HyCaching* is increased with the growth of λ, which is aligned with that of *ReCaching*. This observation is not difficult to understand as the overall costs for both algorithms are effected by the transfer cost in proportion when μs are fixed. Another interesting observation, which is beyond our expectation, is that *HyCaching* is consistently better than *ReCaching*, even the accuracy is low. We can attribute this phenomenon to the mutual optimization between the two component algorithms as we remarked in Section 6.

On the other hand, with the increase of prediction accuracy, the perf. ratio of *ReCaching* is maintained as a constant for each λ since it works online and has nothing to do with the prediction accuracy while for *HyCaching*, its perf. ratio decreases in approximately a linear fashion to approach the off-line optimal solution when the accuracy is varied from 0 to 1, progressively enlarging the performance gap to *ReCaching*. This is not difficult to understand as when the

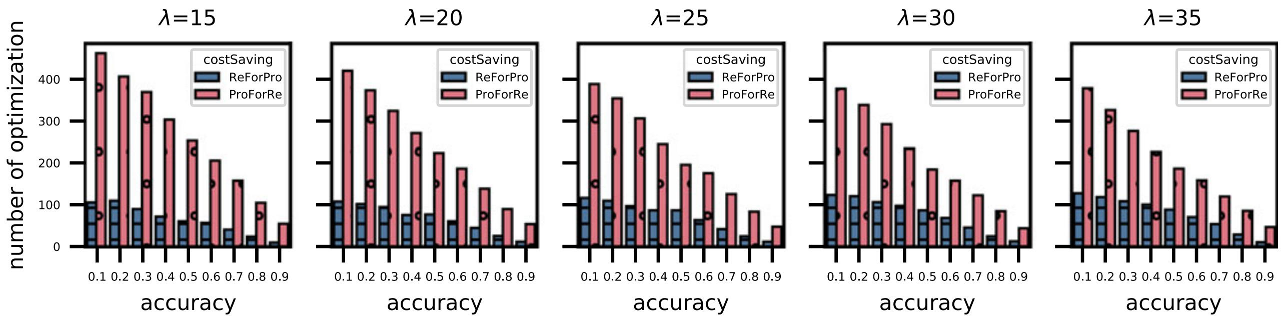


Fig. 13. Mutual optimization between *ProCaching* and *ReCacing* in *HyCaching*.The number of requests is 1000 and μ = [0.3,0.4].

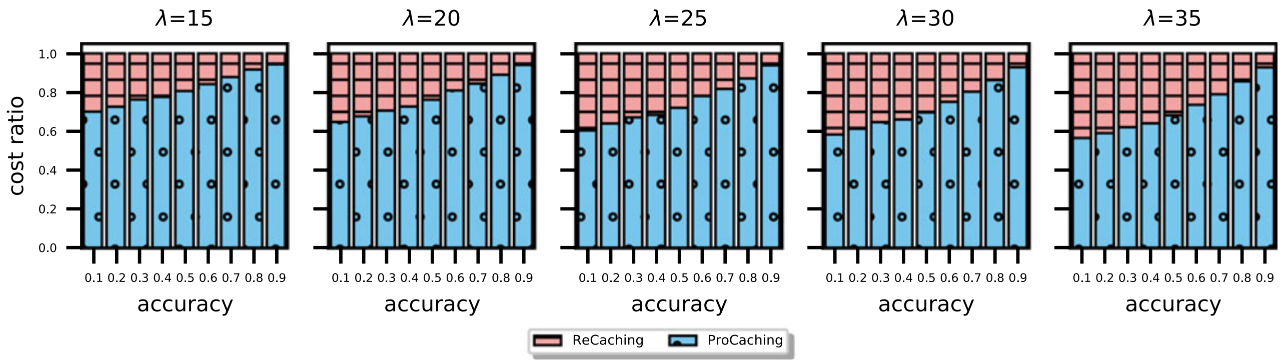


Fig. 14. Performance breakdowns of *HyCaching* under different λs. The number of requests is 1000 and $\mu = [0.3, 0.4]$.

accuracy increases, most of the requests are correctly predicted and thus satisfied in optimal ways.

Mutual Optimization. ProCaching and *ReCaching* are not orthogonally combined in *Hybird*. Actually they can benefit each other as shown in Fig. 8. In this experiment, we evaluated these mutual benefits. To this end, we denoted the number of times that *ReCaching* is exploited to reduce the cost of *ProCaching* as *ReForPro*, and as *ProForRe* in reverse. We compared them in Fig. 13.

As shown in the figure, the mutual optimizations for both *ReForPro* and *ProForRe* are inversely proportional to the prediction accuracy. This is not difficult to understand as with the increase of accuracy, the number of missed predicted request is reduced, so are the opportunities for mutual optimization. From the figure, another interesting observation is that *ProForRe* is much larger than *ReForPro* across all the examined cases, especially when the accuracy is lower. These results can be expected since the caching schedule created by *ProCahing* can cover most of the requests in $\mathcal{R}' \setminus \mathcal{R}$, especially when the accuracy is relatively low. However, in contrast, the opportunities of *ReCaching* to benefit *ProCaching* is relatively small, with only a period of Δt copy extension for each request served by *ReCaching*.

Cost Breakdown. To deeply understand how *ReCaching* and *ProCaching* fight against each other inside *HyCaching*, we broke down the cost of *HyCaching* based on these two algorithms and investigated their mutual effects as shown in Fig. 14. From the figure, one can see the cost ratios of *ProCaching* are becoming larger and larger as the prediction accuracy increases, which means *HyCaching* becomes more “optimal” with reduced *ReCaching* effects. In contrast, by following the same arguments in the last paragraph, one can observe that the cost ratio of *ReCaching* is also increasing accordingly with the increase of λ as more transfers are performed by *ReCaching* than by *ProCaching*.

8 Related Work

Given its inherent merits, edge-based CDN has been being widely deployed to realize fast, always-on access to data services from any device across the globe. As such, the studies on cost-effective edge CDNs, especially for the data caching problem in its edge network, are arousing great interest in both academia and industry [3], [6], [12], [13], [15], [32], [33].

Ding *et al.* [6] proposed an edge content delivery and update (ECDU) framework whereby edge content delivery and edge content update algorithms were developed with an aim to reduce the pressure on the core network while saving bandwidth resources to make up for the shortcomings of existing CDN-based works. Although the aim is consistent with ours, the algorithms they proposed are not competitive and the worst cases could not be bounded. Xia *et al.* [32], [33] conducted a further study on this problem in the edge from an app vendor’s perspective, and came up with both online and off-line algorithms for cost-effective data distribution to overcome the above shortcomings. However, these algorithms only take into account the transfer cost in the edge data distribution, missing the consideration on caching cost, which is the key in our study.

The similar yet different problems for data caching were also recently conducted in the context of CDN. Tan *et al.* [34] proposed a pair of $O(\log k)$-competitive randomized and deterministic algorithms to redirect caching requests for minimizing the overall cost via relay and bypass operations, where k is the total number of slots in all caches. Clearly, these operations are orthogonal to our case.

In addition to the edge and CDN, data caching problem in its abstract form has also been intensively studied in other contexts [27], [35], [36]. Arguably, the problem in the current form was first proposed and studied by B. Veeravalli in context of network services [27] as early as 2003, where a homogeneous cost model is assumed to obtain an $O(m^2 n \log m)$ time algorithm for optimum caching schedule. Wang *et al.* [25] re-investigated this problem in the context of cloud and developed an improved solution with time complexity of $O(m^2 n)$ and also with some abilities to handle certain variants of this problem such as those having multiple data items. However, due to the intrinsic hardness of this problem, only approximation algorithms for its off-line form are studied, sporadically with some considerations on its online cases [24]. A variant of this problem is also investigated by Mansouri *et al.* [37] in the context of cloud, who exploited the storage classes with different prices across CSPs to serve time-varying workloads. To this end, they proposed both optimal off-line and competitive online algorithms for data replication and migration in cloud data centers.

As opposed to the above-mentioned studies in the context of cloud, which focus more or less on the off-line situations, Gharaibeh *et al.* [38] proposed an $O(1)$-competitive online caching algorithm for this problem in the context of *Content Centric Networking* (CNN). With this result as a basis, they further delved into the situation when online collaborative caching can be exploited to bring the content item closer to its users with minimum rental cost in a multicell-coordinated system [39]. To this end, they presented an $O(\log m)$-competitive online algorithm (m: network size. Similarly, Bikenkowski *et al.* [40] conducted a competitive analysis to design a randomized and a deterministic online algorithms, but in the context of *virtual network*, that also achieve a competitive ratio of $O(\log m)$ for finding a good trade-off between the benefit and cost of a migratable data service.

The major difference between our work and these studies is that the existing work is either short of the notion of the function to automatically maintain the number of item copies for optimal costs or restricted to the cost model different from ours. However, this function plays an essential role in our cost-driven caching problem. Moreover, none of surveyed algorithms combine the merits of both online and off-line algorithms if they have them.

The problem proposed in this paper can be viewed as a follow-up research of the caching problem presented in [24] where an efficient and optimal off-line algorithm is obtained by leveraging dynamic programming techniques and an 3-competitive online is designed based on the concept of *anticipatory caching*. Our paper extends these results by relaxing the cost model to a semi-homo model whereby a fast and optimal off-line algorithm and an 2-competitive online algorithm are designed with a provable $2 - o(1)$ lower bound of the competitive ratio being established. On the other hand, to combine the merits of both algorithms, a hybrid caching algorithm is also put forward.

9 Conclusion

In this paper, we studied a data caching problem in edge-based CDNs to facilitate the content delivery to serve a sequence of requests, off-line and online, with minimum costs as a goal based on a semi-homo cost model. To this end, we first designed an $O(mn)$ time and space optimal proactive off-line algorithm, called *pro-caching*, by reducing the problem to a simple shortest path problem in a directed weighted network graph, and then extended the idea of anticipatory caching to develop an 2-competitive reactive online algorithm, called *re-caching*, for this problem and showed its tightness by proving that no deterministic online algorithm can do better than $2 - o(1)$ in its worst case. Finally, to combine the advantages of both algorithms, we also presented a hybrid algorithm, called *hy-caching*, to fully utilize the power and benefits of edge-based CDNs while reducing their service costs. Our results improve the previous results not only in the cost model being used but also in the time complexity, competitive ratio, and the quality of the solutions. We provably achieve these results with our deep insights into the problem and the careful analysis, together with an empirical evaluation.

Notably, in this research, we only focus on how to schedule the caching of a single data item without concerning the cache capacity as our caching policy is cost-driven, rather than capacity-driven for multiple data items as in the traditional case. As a result, we are concerned mainly with how to minimize the caching cost to serve the request sequence, instead of reducing the miss ratio. With this consideration, the proposed algorithms, online and off-line, are relatively easy to extend to the case for multiple data items by simply summing up the cost of each item in a linear fashion to get the overall cost for multiple data items in both cases. Of course, when considering capacity as a constrain, cost-driven data caching for multiple items is highly desired [41]. However, to the best of our knowledge, the problem is believed to be NP-hard for the off-line case, while for the online case, the design of an online algorithm with a good competitive ratio is still in our future work.

Acknowledgments

The authors would like to thank all reviewers for their insightful comments and suggestions. This work was supported by Key-Area Research and Development Program of Guangdong Province under Grant 2020B010164002. Portion of this paper [1] has been published in IEEE International Conference on Computer Communications 10-13 May 2021.

References

[1] Y. Wang *et al.*, "Cost-driven data caching in the cloud: An algorithmic approach," in *Proc. IEEE INFOCOM IEEE Conf. Comput. Commun.*, 2021, pp. 244–252.

[2] G. Huang *et al.*, "Software-defined infrastructure for decentralized data lifecycle governance: Principled design and open challenges," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 1674–1683.

[3] F. Wang, F. Wang, J. Liu, R. Shea, and L. Sun, "Intelligent video caching at network edge: A multi-agent deep reinforcement learning approach," in *Proc. IEEE INFOCOM*, 2020, pp. 2499–2508.

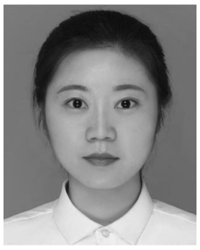
[4] F. Z. Jiang, K. Thilakarathna, S. Mrabet, M. A. Kaafar, and A. Seneviratne, "uStash: A novel mobile content delivery system for improving user QoE in public transport," *IEEE Trans. Mobile Comput.*, vol. 18, no. 6, pp. 1447–1460, Jun. 2019.

[5] T. Taleb, P. A. Frangoudis, I. Benkacem, and A. Ksentini, "CDN slicing over a multi-domain edge cloud," *IEEE Trans. Mobile Comput.*, vol. 19, no. 9, pp. 2010–2027, Sep. 2020.

[6] C. Ding, A. Zhou, J. Huang, Y. Liu, and S. Wang, "ECDU: An edge content delivery and update framework in mobile edge computing," *EURASIP J. Wireless Commun. Netw.*, vol. 2019, pp. 1–9, 2019.
[7] R. Karasik, O. Simeone, and S. Shamai Shitz , "How much can D2D communication reduce content delivery latency in fog networks with edge caching?," *IEEE Trans. Commun.*, vol. 68, no. 4, pp. 2308–2323, Apr. 2020.
[8] J. Xu, L. Chen, and P. Zhou, "Joint service caching and task offloading for mobile edge computing in dense networks," in *Proc. IEEE INFOCOM*, 2018, pp. 207–215.
[9] Z. Xu, L. Zhou, S. Chi-Kin Chau, W. Liang, Q. Xia, and P. Zhou, "Collaborate or separate? Distributed service caching in mobile edge clouds," in *Proc. IEEE INFOCOM*, 2020, pp. 2066–2075.
[10] T. Zhao, I. H. Hou, S. Wang, and K. Chan, "Red/LeD: An asymptotically optimal and scalable online algorithm for service caching at the edge," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 8, pp. 1857–1870, Aug. 2018.
[11] S. Zhang, P. He, K. Suto, P. Yang, L. Zhao, and X. Shen, "Cooperative edge caching in user-centric clustered mobile networks," *IEEE Trans. Mobile Comput.*, vol. 17, no. 8, pp. 1791–1805, Aug. 2018.
[12] Y. Jiang, Y. Hu, M. Bennis, F. C. Zheng, and X. You, "A mean field game-based distributed edge caching in fog radio access networks," *IEEE Trans. Commun.*, vol. 68, no. 3, pp. 1567–1580, Mar. 2020.
[13] Y. Qian, R. Wang, J. Wu, B. Tan, and H. Ren, "Reinforcement learning-based optimal computing and caching in mobile edge network," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 10, pp. 2343–2355, Oct. 2020.
[14] X. Ma, A. Zhou, S. Zhang, and S. Wang, "Cooperative service caching and workload scheduling in mobile edge computing," in *Proc. IEEE INFOCOM*, 2020, pp. 2076–2085.
[15] X. Wang, C. Wang, X. Li, V. C. Leung, and T. Taleb, "Federated deep reinforcement learning for internet of things with decentralized cooperative edge caching," *IEEE Internet Things J.*, vol. 7, no. 10, pp. 9441–9455, Oct. 2020.
[16] Y. Li, K.-H. Kim, C. Vlachou, and J. Xie, "Bridging the data charging gap in the cellular edge," in *Proc. ACM Special Interest Group Data Commun.*, 2019, pp. 15–28.
[17] D. T. Nguyen, L. B. Le, and V. Bhargava, "Price-based resource allocation for edge computing: A market equilibrium approach," *IEEE Transactions on Cloud Computing*, vol. 9, no. 1, pp. 302–317, Jan.–Mar. 2021.
[18] A. Sadeghi, F. Sheikholeslami, A. G. Marques, and G. B. Giannakis, "Reinforcement learning for adaptive caching with dynamic storage pricing," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 10, pp. 2267–2281, Oct. 2019.
[19] H. Wu *et al.*, "Delay-minimized edge caching in heterogeneous vehicular networks: A matching-based approach," *IEEE Trans. Wireless Commun.*, vol. 19, no. 10, pp. 6409–6424, Oct. 2020.
[20] AliCloud. Accessed: Jun. 13, 2021. [Online]. Available: https://www.alibabacloud.com/product/linkiotedge
[21] A. Mazrekaj, I. Shabani, and B. Sejdiu, "Pricing schemes in cloud computing: An overview," *Int. J. Adv. Comput. Sci. Appl.*, vol. 7, pp. 80–86, 2016.
[22] P. R. Lei, T. J. Shen, W. C. Peng, and I. J. Su, "Exploring spatial-temporal trajectory model for location prediction," in *Proc. IEEE 12th Int. Conf. Mobile Data Manage.*, Jun. 2011, pp. 58–67.
[23] C. P. Lau, A. Alabbasi, and B. Shihada, "An efficient content delivery system for 5G CRAN employing realistic human mobility," *IEEE Trans. Mobile Comput.*, vol. 18, no. 4, pp. 742–756, Apr. 2019.
[24] Y. Wang, S. He, X. Fan, C. Xu, J. Culberson, and J. Horton, "Data caching in next generation mobile cloud services, online vs. off-line," in *Proc. 46th Int. Conf. Parallel Process.*, Bristol, U.K., Aug. 2017, pp. 412–421.
[25] Y. Wang, B. Veeravalli, and C.-K. Tham, "On data staging algorithms for shared data accesses in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 4, pp. 825–838, Apr. 2013.
[26] M. Charikar, D. Halperin, and R. Motwani, "The dynamic servers problem," in *Proc. Ninth Annu. ACM-SIAM Symp. Discrete Algorithms*, Philadelphia, PA, USA, 1998, pp. 410–419.
[27] B. Veeravalli, "Network caching strategies for a shared data distribution for a predefined service demand sequence," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 6, pp. 1487–1497, Nov./Dec. 2003.
[28] W. Shi and C. Su, "The rectilinear Steiner arborescence problem is np-complete," in *Proc. 11th Annu. ACM-SIAM Symp. Discrete Algorithms*, Philadelphia, PA, USA, 2000, pp. 780–787.
[29] A. S. Khatouni *et al.*, "An open dataset of operational mobile networks," in *Proc. 18th ACM Symp. Mobility Manage. Wirel. Access*, Dec. 2019, pp. 83–90, 2020.
[30] Y. Wang, B. Veeravalli, and C. K. Tham, "On data staging algorithms for shared data accesses in clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 4, pp. 825–838, Apr. 2013.
[31] L. N. Hyseni and A. Ibrahimi, "Comparison of the cloud computing platforms provided by amazon and Google," in *Proc. Comput. Conf.*, 2017, pp. 236–243.
[32] X. Xia, F. Chen, Q. He, J. C. Grundy, M. Abdelrazek, and H. Jin, "Cost-effective app data distribution in edge computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 1, pp. 31–44, Jan. 2021.
[33] X. Xia, F. Chen, Q. He, J. Grundy, M. Abdelrazek, and H. Jin, "Online collaborative data caching in edge computing," *IEEE Trans. on Parallel and Distributed Syst.*, vol. 32, no. 2, pp. 281–294, Feb. 2021.
[34] H. Tan, S. H. Jiang, Z. Han, L. Liu, K. Han, and Q. Zhao, "CAMUL: Online caching on multiple caches with relaying and bypassing," in *Proc. IEEE INFOCOM*, 2019, pp. 244–252.
[35] S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, and W. Wang, "A survey on mobile edge networks: Convergence of computing, caching and communications," *IEEE Access*, vol. 5, pp. 6757–6779, Mar. 2017.
[36] C. Li, Y. Wang, H. Tang, Y. Zhang, Y. Xin, and Y. Luo, "Flexible replica placement for enhancing the availability in edge computing environment," *Comput. Commun.*, vol. 146, pp. 1–14, 2019.
[37] Y. Mansouri, A. N. Toosi, and R. Buyya, "Cost optimization for dynamic replication and migration of data in cloud data centers," *IEEE Transactions on Cloud Computing*, vol. 7, no. 3, pp. 705–718, Jul.–Sep. 2019.
[38] A. Gharaibeh, A. Khreishah, and I. Khalil, "An o(1)-competitive online caching algorithm for content centric networking," in *Proc. IEEE INFOCOM 35th Annu. IEEE Int. Conf. Comput. Commun.*, 2016, pp. 1–9.
[39] A. Gharaibeh, A. Khreishah, B. Ji, and M. Ayyash, "A provably efficient online collaborative caching algorithm for multicell-coordinated systems," *IEEE Trans. Mobile Comput.*, vol. 15, no. 8, pp. 1863–1876, Aug. 2016.
[40] M. Bienkowski, A. Feldmann, J. Grassler, G. Schaffrath, and S. Schmid, "The wide-area virtual service migration problem: A competitive analysis approach," *IEEE/ACM Trans. Netw.*, vol. 22, no. 1, pp. 165–178, Feb. 2014.
[41] D. Huang, X. Fan, Y. Wang, S. He, and C. Xu, "Dp_greedy: A two-phase caching algorithm for mobile cloud services," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2019, pp. 1–10.

Yang Wang received the BSc degree in applied mathematics from the Ocean University of China in 1989, the MSc degree in computer science from Carlton University in 2001, and the PhD degree in computer science from the University of Alberta, Canada, in 2008. He is currently with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, as a full professor and with Xiamen University, China, as an adjunct professor. From 2009 to 2011 and from 2014 to 2015, he was an Alberta Industry R&D Associate and a Canadian Fulbright Scholar. His research interests include service and cloud computing, programming language implementation, and software engineering.

Hao Dai received the MSc degree in communication and electronic technology from the Wuhan University of Technology in 2017. He is currently working toward the PhD degree with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. His research interests include cloud computing, big data processing, and mobile edge computing systems.



Xinxin Han received the bachelor's and master's degrees in mathematics and applied mathematics. She is currently working toward the PhD degree with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. Her research interests include scheduling and algorithm optimization and data caching and offloading problems in edge and cloud computing.

Pengfei Wang received the BSc degree in aircraft design and engineering from BeiHang University in 2018. He is currently working toward the MSc degree with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. His research interests include cloud computing, distributed system, and online dispatching algorithm.

Yong Zhang received the PhD with the Department of Computer Science and Engineering, Fudan University, in 2007. He is currently a professor with SIAT, CAS, honorary professor with the University of Hong Kong. Before joining SIAT, he was postdoctoral fellow and senior researcher with TU-Berlin and HKU. He has authored more than 100 papers in refereed journals and conferences. His research interests include design and analysis of algorithms, combinatorial optimization, and wireless networks.

Chengzhong Xu (Fellow, IEEE) received BSc and MSc degrees in computer science and engineering from Nanjing University in 1986 and 1989, respectively, and the PhD degree in computer science and engineering from the University of Hong Kong in 1993. He is currently a chair professor of computer science and the dean of the Faculty of Science and Technology, University of Macau, China. He has authored or coauthored more than 400 papers in journals and conferences. His research interests include cloud and distributed computing, systems support for AI, smart city, and autonomous driving. He is on a number of journal editorial boards and the chair of IEEE TCDP from 2015 to 2020. He is a fellow of the IEEE.