On Cost-Driven Collaborative Data Caching: A New Model Approach

Yang Wang, Shuibing He, Xiaopeng Fan, Chengzhong Xu, *Fellow, IEEE*, and Xian-He Sun, *Fellow, IEEE*

Abstract—In this paper we consider a new caching model that enables data sharing for network services in a cost-effective way. The proposed caching algorithms are characterized by using monetary cost and access information to control the cache replacements, instead of exploiting capacity-oriented strategies as in traditional approaches. In particular, given a stream of requests to a shared data item with respect to a homogeneous cost model, we first propose a fast off-line algorithm using dynamic programming techniques, which can generate an optimal schedule within $O(mn)$ time-space complexity by using cache, migration as well as replication to serve a n-length request sequence in a m-node network, substantially improving the previous results. Furthermore, we also study the online form of this problem, and present an 3-competitive online algorithm by leveraging an idea of anticipatory caching. The algorithm can serve an online request in constant time and is space efficient in $O(m)$ as well, rendering it more practical in reality. We evaluate our algorithms, together with some variants, by conducting extensive simulation studies. Our results show that the optimal cost of the off-line algorithm is changed in a parabolic form as the ratio of caching cost to transfer cost is increased, and the online algorithm is less than 2 times worse in most cases than its optimal off-line counterpart.

Index Terms—Data caching and migration, collaborative caching, anticipatory caching, dynamic programming, competitive analysis

1 Introduction

As one of the often-used techniques to facilitate data accesses in network services, data caching has been widely studied since decades ago [1], [2], [3]. By leveraging the data caching technique, one can substantially reduce the access latency to frequently used data items, which in turn also minimizes the network traffics. Traditionally, the capacity of network cache is limited and its potentials are maximized by exploiting the spatial-and-temporal localities exhibited in programs to optimize the caching replacements. Nowadays, with the explosive growth of data services in diverse areas, the applications of data caching are becoming much more challenging than ever before [4], [5], [6]. Apart from the capacity constraints, there would be a number of other factors that may affect the effectiveness of the cache [6], [7]. Specifically, in contrast to traditional case, there are two main challenges to the caching replacement policies. First, the cache replacement policy is usually cost driven in current data services, instead of being capacity-oriented as in classic network caching [8], [9], [10], [11]. This is because the current data services are usually deployed in the cloud platforms, where the cache resources could be virtually infinite provided that the users can afford them based on the billing model. Second, as apposed to traditional network caching, whose design, as stated, is to exploit the spatial and temporal localities in access sequences, the data caching in current data services is usually required to facilitate the mobile accesses that often exhibit *spatial-temporal trajectory* patterns [12], [13]. This implies that the data items need to be carefully cached along the path of request sites at different points of time for efficient access.

The above challenges profoundly affect the design of cache policies on how to utilize the resources in a cost-effective way. The conventional capacity-oriented replacements like *LRU* or *LFU* are typically designed to maximize the cache *hit ratio* by selecting victims wisely to evict from the cache so that newly accessed items could be brought in. While for the caching in current data services, a new model is demanded that not only facilitates the spatial-and-temporal trajectory of accesses, but also supports the goal that minimizes the overall cost of the accesses, instead of sensible use of the cache capacity, since the capacity is in general not an issue anymore, it can be neither fixed nor bounded as a cloud service.

In this paper, we study the new caching model by efficiently moving around a shared data item, with possible multiple copies, in a fully connected network so that a sequence of requests to the item could be satisfied with minimum service cost. In the study, the request sequences could be online or off-line, signifying different application scenarios in reality. For example, let us imagine an off-line case where a set of users of a multimedia network (say Netflix)

- Y. Wang, X. Fan, and C. Xu are with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Xueyuan Blvd. 1068, Shenzhen 518055, China. E-mail: {yang.wang1, xp.fan, cz.xu}@siat.ac.cn.
- S. He is with the School of Computer Science, Wuhan University, Luojiashan, Wuhan, Hubei 430072, China. E-mail: heshuibing@whu.edu.cn.
- X.-H. Sun is with the Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616. E-mail: sun@iit.edu.

Manuscript received 1 June 2018; revised 22 Aug. 2018; accepted 1 Sept. 2018. Date of publication 5 Sept. 2018; date of current version 13 Feb. 2019. (Corresponding author: Xiaopeng Fan.) Recommended for acceptance by J. Wang. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2018.2868642

1045-9219 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

TABLE 1
Classic Model versus Cost-Driven Model

	Classic Caching [9]	Cost-Driven Caching
Network	Fully Connected	Fully Connected
Cost Model	Transfer Cost	Caching&Transfer Costs
Operation	Page Fault	Cache, Transfer& Replicate
Cache Size	Fixed Number k	Not Fixed Number
Opt. Goal	Total Fault Cost	Total Service Cost
Locality	Spatial-Temporal	Spatial-Temporal Trajectory
Opt. Off-line	Belady's Alg. [15]	**O(mn) Fast Opt. Alg.**
Comp. Online	k-competitive [16]	**3-competitive**

may open different movies at different time from different movie servers, and an ad clip is randomly inserted into the stream of each movie video for advertisements. In this case, the play time instances of the ad clip in each video stream can be determined in advance, and the network service can exploit this knowledge to schedule the ad clip to the request sites at the desired time instances by caching or migrating between the servers for cost saving, instead of retrieving it from the central ad repository. As in this example, the off-line sequence in general is embedded with the spatial-temporal trajectory information of the requests, which could be secured in advance by mining the data historical logs or exploiting some spatial-temporal trajectory model [13]. While for online sequence, we assume nothing known about it in order to make competitive analysis, which could give a theoretical bound to the worst case of the algorithm.

As with the classic caching problem, which is usually assumed to have homogeneous miss penalty cost, we also assume that the cost model in the caching model is homogeneous, which means the transfer cost between any pair of servers in the network is identical and the caching cost per time unit at each individual server is also identical. The model is practical in the sense that the provisioned infrastructure for a particular data service is always organized as a subset of homogeneous resources, which in turn results in homogeneous computations and communications [14]. To make our research goal more clear, a detailed comparison between classic network caching and the new data caching is listed in Table 1 where the fast optimal off-line and competitive online algorithms for the data caching problem are the focus of this paper, and the main results of this paper are also highlighted in the table.

Specifically, we make the following contributions in this paper:

1) *An Optimal $O(mn)$ Off-line Algorithm:* We first consider the off-line algorithms with respect to a given request sequence characterized by its spatial-temporal trajectory information. To this end, we use dynamic programming technique to design a fast optimal algorithm that can cache, migrate and replicate the shared data item in a fully connected m-node network to serve an n-request sequence within $O(mn)$ time and space complexity. The result is $O(m\log m)$ times faster than the previous algorithms [14], [17].
2) *A 3-Competitive Online Algorithm:* We conduct competitive analysis on the online version of this problem. By leveraging a concept of *anticipatory caching* idea, we propose an 3-competitive online algorithm, together with some variants, which not only guarantees the performance of this algorithm in the worst case, but is also efficient in both time and space. To the best of our knowledge, this is the first competitive analysis to show this new caching paradigm has a *constant* competitive ratio, a surprising result when comparing it with those in the *classic caching* problem.
3) *Efficient Algorithm Implementation and Evaluation:* We develop efficient data structures to implement the algorithms, together with some variants, and conduct simulation-based studies to investigate their performance behaviors in reality. The results show that the proposed algorithms are not only practical to the data caching problem, but also appear to be of theoretical significance to a natural new paradigm in the realm of online algorithms

The reminder of this paper is organized as follows: we survey some related work and compare them with ours in Section 2, and introduce the problem notation in Section 3. After that, we present the off-line algorithm, together with its critical analysis, in Section 4, and conduct competitive analysis for the online problem in Section 5. Then, we validate our findings in Section 6, and conclude the paper in the last section.

2 Related Work

The caching problem, in its different forms, has been extensively studied in many traditional network-based applications to improve the efficiency of data accesses [1], [2], [4], [18], [19], [20]. Most of the existing researches, each with different knowledge and goals, are capacity oriented, which implies the replacement strategies are determined given limited size of the cache [8], [10], [11].

The caching problem can be considered in both online and off-line settings, depending on the application scenarios. In the online setting, the incoming requests are unknown, and the replacements are determined at run-time to improve the optimization goals (e.g., *hit ratio*, *load balancing*). Karger et al. [2] introduce a cooperative caching strategy based on *consistent hashing* [21] to address the load balancing and fault tolerance in the web caches. The consistent caching approach does away with all inter-cache communication and allows the involved caches to behave together in one coherent system. In contrast, Chockler et al. [4] study the caching problem in the cloud service, and propose *BLAZE*, a simple multi-tenant caching scheme to guarantee minimum QoS for each tenant. However, these schemes are still oriented to the cache resources with limited capacity, which is not necessary in the cloud services. With *BLAZE* as a basis, Chockler et al. [5] present a new cloud-based caching service called *Simple Cache for Cloud* (SC2), which can optimize the global use of cache resources while simultaneously guaranteeing minimum service quality for all users according to their stated requirements. Although SC2 is distributed in network for shared uses among tenants as in our case, it, as in *BLAZE*, still takes the maximization of overall *hit ratio* as a goal.

As with Karger et al. [2], Cao and Irani also investigate the web caches, but from competitive analysis perspective. They present a cost-aware caching algorithm that incorporates locality with cost and size concerns in a simple and non-parameterized fashion for high performance [1]. Their algorithm is k-competitive, where k is the ratio of the size of the cache to the size of the smallest data item.

Charikar et al. [22] propose a *dynamic servers* (DS) problem, which can be viewed as a generalization of the data caching problem in our case in terms of its request pattern and metric space. For the online DS problem in arbitrary metric spaces, they present an $O(\min\{\log n, \log \phi\})$-competitive algorithm where n is the number of requests and ϕ, the (normalized) diameter of the metric space. This is an improvement to the previous result of Halperin et al. in [23] where the DS problem has an $O(\log n)$ competitive algorithm in a special case of paths. As a complement to these studies, our results further show that the DS problem has constant competitive ratio with respect to homogeneous cost model when the sequence of requests is not batched (i.e., batch size is one).

More recently, Mansouri et al. [24] studied a similar problem regarding the dynamic replication and migration of data in cloud data centers. They propose two online algorithms that could make a trade-off between residential and migration costs and dynamically select storage classes across CSPs. Our problem is different from theirs in terms of system model, cost model, and problem definition. However, they are highly related. We achieve an online algorithm with a constant competitive ratio for our problem while the competitive ratios of their algorithms depend on the configurations of their system model, which could be much worse in certain cases.

In the off-line setting, one of the pioneer works on optimal caching is conducted by Belady [15]. Although there have been some follow-up studies on this problem since then [25], [26], [27], none of them is applicable to our caching model. A highly related one is Veeravalli's work [17], which deals with the network caching problem with respect to sharing a data item in a set of fully networked servers. With a homogeneous cost model, they obtain an optimal algorithm using dynamic programming technique within $O(nm^2 \log m)$ time, which can automatically generate multiple copies to minimize the total service cost of request sequence. In contrast to this algorithm, our off-line algorithm can reach the same goal in $O(mn)$ time and space complexity. Veeravalli's work is later extended by Wang et al. [14] to the context of clouds with some practical constraints so that a balance between the caching costs and the transfer costs of multiple shared data items can be optimally struck.

Unlike previous studies, Scouarnec et al. [6] investigate cache policies for cloud-based caching from a different angle that views cloud resources to be potentially infinite and only paid when used. To deal with this new context, they design and evaluate a new caching policy that minimizes the cost of a cloud-based system in online fashion. We adopt this point of view to design the optimal off-line and competitive online data caching algorithms in the cloud. Particularly, the off-line sequence is assumed to be available in advance when considering the trajectory information inherent in access patterns.

In summary, compared to existing methods, our model is new in the sense that it not only uses the cost metric, but also considers the spatial-and-temporal trajectory pattern of the mobile accesses, whereby a fast optimal off-line and $O(1)$-competitive online algorithms are introduced, together with their deep analysis.

TABLE 2
Frequently Used Notation

Symbol	Meaning
$\mathcal{S}$	server set $\mathcal{S} = \{s^1, \dots, s^m\}$
s^i	server index
s_i	server reference label
$\mathcal{R}$	request vector $\mathcal{R} = \langle r_1, \dots, r_n \rangle$
r_i	request $r_i = (s_i, t_i), s_i \in \mathcal{S}$
r_0	$r_0 = (s^1, 0)$
r_{-j}	$r_{-j} = (s^j, -\infty), 1 \leq j \leq n$
$\delta t_{i,j}$	$\delta t_{i,j} = t_j - t_i$, time diff. btw r_i and r_j
$p(i)$	the previous req. before t_i (same server)
$p'(i)$	the most recent event before t_i (same server)
σ_i	$\sigma_i = t_i - t_{p(i)}$
$Tr(s_i, s_j, x)$	data transfer from s_i to s_j at t_x
$H(s, x, y)$	data is held in cache on server s from t_x to t_y
μ	uniform caching cost per time unit
λ	uniform transfer cost between servers
ω_j^i	the ith anticipatory caching cost on s^j
Ω_j	the set of SC costs on s^j
$\Gamma(i)$	space of the feasible schedules for up to r_i
$\Psi^*(n)$	optimal schedule for up to r_n
$Cost(\Psi(i))$	the cost of schedule $\Psi(i)$
$\Psi^{(-1)}(i)$	sub-schedule of $\Psi(i)$ for up to r_{i-1}
$\Delta_\Psi(i)$	marginal cost $\Delta_\Psi(i) = \Pi(\Psi(i)) - \Pi(\Psi^{(-1)}(i))$
b_i	$b_i = \min\{\lambda, \mu\sigma_i\}, 1 \leq i \leq n.$
B_i	$B_i = \sum_{j=1}^{i} b_j$
$\Psi'(i)$	conditional opt. sched. with $H(s_i, t_{p(i)}, t_i)$ as the final cache H
κ	pivot index

3 Caching Model and Its Notation

In this section, we describe the caching model in details under a homogeneous cost model. We first define some useful concepts and then give the standard form of the solution to the problem. Some used symbols are listed in Table 2 for easy reference.

Suppose in a network environment, the server set is $\mathcal{S} = \{s^1, \dots, s^m\}$ that are fully connected by a network, and a shared data item is initially located at a certain server, say s^1. The request for the data item is generated online at each server, which could be made by users outside the network. The request vector is denoted as $\mathcal{R} = < r_1, \dots, r_n >$, where each $r_i = (s_i, t_i)$, with $t_i < t_{i+1}$ and $s_i \in \mathcal{S}$, represents that r_i is made from s_i at t_i. Note that the use of superscripts for the server indexes (e.g., s^i) should be distinguished from the reference labels (e.g., s_i). For example, the ith request r_i's server s_i could be s^j. To satisfy the request sequence, the shared item should be moved around between the servers, replicated at or cached and then deleted at certain servers to satisfy each request on time only if the total cost is minimum (cost model is discussed later).

To simplify boundary conditions, we define $r_0 = (s^1, 0)$ and $r_{-j} = (s^j, -\infty), 1 \leq j \leq n$.[1] Note that the requests at

1. As special cases, we use server indexes for r_0 and r_{-j}, instead of the server labels as in $r_i = (s_i, t_i)$.

$-\infty$ will never be included in a solution, and are only defined to make notation on the intervals on a server easier. We assume at least one request for each server (i.e., we ignore those servers without requests).

For $i < j$, we define $\delta t_{i,j} = t_j - t_i$, which is the time difference between requests r_i and r_j. For $r_i = (s_i, t_i)$, $1 \leq i < n$, we define the previous request on the same server by $p(i) = \arg\max_{i' < i}\{s_{i'} = s_i\}$. Then, we can define the *server interval on request* r_i as $\sigma_i = t_i - t_{p(i)}$. Moreover, we can define a data item *transfer* $Tr(s_i, s_j, x)$ from server s_i to s_j at time x, and say the data item is cached, or *held in cache* on server s from time x to y using the notation of $H(s, x, y)$.

The cost model for this process is homogeneous in the sense that the cost of one unit of time caching on each server is the same across all the servers, denoted by μ, and the transfer cost from any server to any other server, denoted by λ, is also identical no matter which servers are being used. As a result, the transfer cost between servers s^j and s^k is λ, $\forall j \neq k$ at any time x, and the caching cost from time x to y for any server is $\mu(y - x)$. This model is corresponding to the traditional caching model where the penalty of cache miss is always assumed a constant. On the other hand, the homogeneous cost model is also available in reality [14].

The problem is to find the set of cache intervals and transfers for the data item so that

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, $1 \leq j \leq n$ (discuss later).
3) The total transfer and caching costs are minimized.

To this end, multiple copies of the data item could be automatically generated during the service process. Except the first one each of the other copies is caused by a transfer, and eventually deleted when it is no longer used. Therefore, without loss of accuracy, we assume that the replication cost and the deletion cost are free since these costs are always constants and can be included in the transfer cost.

Fig. 1 shows an example to illustrate the problem notation where three servers are fully connected by a network. A shared data item is initially located at s^1, it is migrated, replicated or cached among the servers to satisfy the request sequence in time order with minimized total cost as a goal. 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., $r_7@s_3$).

Definition 1 (Schedule). *We say that a* schedule $\Psi(i)$ *is any* minimal *set of caches and transfers satisfying 1) and 2) for online requests* $r_0, \ldots, r_i$. *A schedule is* optimal *if it also satisfies 3). However, it is not achievable for online algorithms.*

There could be many feasible (off-line) schedules for $r_0, \ldots, r_i$, and we use $\Gamma(i)$ to represent the space of the feasible schedules for up to r_i, each $\Psi(i)$ having a cost, denoted by $Cost(\Psi(i))$. The goal is to find an optimal schedule $\Psi^*(n)$

$$\Psi^*(n) = \underset{\Psi(n)\in\Gamma(n)}{\arg\min}\{Cost(\Psi(i))\}.$$

We can view a schedule in a *space-time diagram* as shown in [19], where the edges are caching intervals or transfers, and the vertices are requests or end points of transfers. More formally, we have

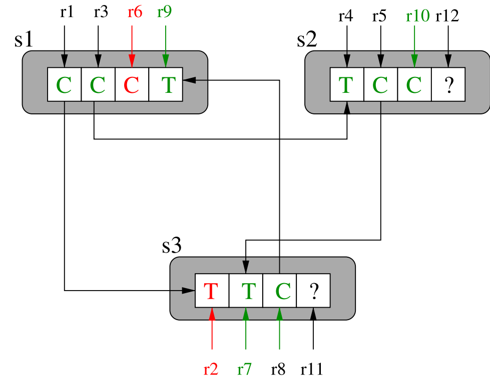


Fig. 1. Three servers are fully connected by a network (only caches are depicted as squares), and a shared data item is initially located at s^1, which could be migrated, replicated, or cached to satisfy request sequence $\mathcal{R} = \langle r_1, \ldots, r_{12}\rangle$ in time order. The characters in each square specify how the data item is created in the cache, say "C" is by caching, and "T" is by transferring, each being charged by its own rate. After the requests are served, the corresponding cached items could be either kept in the cache for future demands (in green color) or deleted for cost saving (in red color). The goal is to minimize the costs to serve the request sequence.

Definition 2 (Space-Time Graph). *We define a* space-time graph *as a weighted directed graph* $G = (V, E, W)$ *where* $V = \{v_{ji} : 0 \leq j \leq m, 0 \leq i \leq n\}$. *Vertex* v_{ji} *corresponds to time* t_i *on server* s^j *when* $1 \leq j \leq m$, *and* $v_{0,i}$ *represents the external storage at* t_i. *The edge set* E *consists of two subsets:*

1) a set of cache edge $C = \{(v_{j,i-1}, v_{ji}) : 0 \leq i \leq n, 1 \leq j \leq m\}$,
2) a set of transfer edge $T = \{(v_{ki}, v_{ji}), (v_{ji}, v_{ki}) : j \neq k$, and $(s_k, t_i) \in \mathcal{R}\}$.

Then, the weight of edge $e \in E$ *is defined as*

$$W(e) = \begin{cases} \lambda & e \in T \\ \mu(t_{i-1} - t_i) & e \in C. \end{cases}$$

Note that a request r_i in the instance will correspond to vertex $v_{s_i,i}$ in the graph. For convenience we will often refer to request vertex r_i. We call all the other vertices *intermediate vertices*. The set of vertices v_{*i} induce a subgraph that is a biconnected star centred on the request vertex r_i. According to the graph, the transfer time is negligible, we thus can satisfy r_i by a transfer at time t_i. This assumption can be validated by tweaking the graph as shown in [14], and is thus often adopted in previous studies [14], [17], [22].

As a schedule is minimal, it implies that it is tree-like. If there is more than one path from s^1 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 since such a path cannot be minimal. Also, a schedule will contain no dead-end caches, that is cache on a server beyond the last request or transfer time from that server.

The data staging problem in its general form is a variant of the Rectilinear Steiner Arborescence problem [28]. As such, it is believed to be NP-complete [17]. However, its formal proof still remains open. Fortunately, in some realistic settings as in our case, when the cost model is homogeneous, we can expect optimal solution to this problem. The

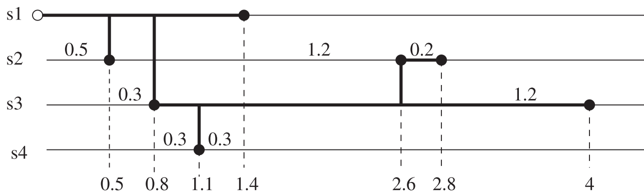


Fig. 2. An example of a standard schedule (shown in bold lines) for an off-line request sequence (solid dots along timeline). The vertical (horizontal) lines represent transfers (caching) that end on requests. The optimal cost is $1.4\mu + 0.2\mu + 1.6\mu$ (caching cost) $+4\lambda$ (transfer cost) = $1.4 + 0.2 + 1.6 + 4.0 = 7.2$ ($\mu = 1$ and $\lambda = 1$).

following observation indicates that we only need to consider the schedules where the transfers end on requests.

Observation 1 (Standard Form). For any instance, there exists at least one optimal schedule in which every transfer occurs at a request time t_i with its output ends on server s_i.

This observation can be directly obtained from Theorem 1 in B. Veeravalli [17]. Fig. 2 shows an optimal schedule in the standard form in a space-time diagram that all transfers end up with the requests at different servers. In the figure, the caching cost and the transfer cost are $1.4\mu + 0.2\mu + 1.6\mu = 3.2$ and $4\lambda = 4.0$, respectively, when $\mu = 1$ and $\lambda = 1$.

The following observation implies that the optimal schedule is a directed tree rooted at s^1 (see Fig. 2).

Observation 2. In any optimal schedule, each request r_i will be served: 1) by the cache (i.e., the cached copy) on s_i, or 2) by a single transfer ending at r_i (see Fig. 2 again).

Given the standard form of schedules, we can define sub-schedule as follows:

Definition 3 (Sub-schedule). *The* primary sub-schedule, *hereafter referred to as the* sub-schedule, *$\Psi^{(-1)}(i)$ of $\Psi(i)$ is a schedule for r_{i-1} that consists of the set of caching intervals and transfers from $\Psi(i)$ required to satisfy all requests $r_0, \ldots, r_{i-1}$.*

Note that even if $\Psi(i)$ is optimal, the sub-schedule $\Psi^{(-1)}(i)$ may not be an optimal schedule for $r_0 \ldots r_{i-1}$. Moreover, $\Psi^{(-1)}(i)$ may not be unique.

Since Observation 1 applies to every transfer in an optimal schedule $\Psi^*(i)$, it will also hold in the sub-schedule $\Psi^{(-1)}(i)$. This justifies that from now on we only need to consider those schedules in the standard form indicated by Observation 1, which dramatically simplifies the feasible schedule space as it is not necessary to consider the proactive data transfer to some vantage servers for the subsequent requests. Note that the last caching interval in $\Psi^*(i)$ may be truncated to the last transfer point or request prior to i on that server in $\Psi^{(-1)}(i)$ (e.g., $r_7@s_3$ in Fig. 2).

4 An Optimal Fast Off-Line Algorithm

Given the notation of the caching model, in this section, we conduct strict analysis on the problem and give our optimal off-line algorithm in time and space complexity of $O(mn)$. We also present our efficient implementation of the proposed algorithm, and show how it works via a running example.

4.1 Problem Analysis and Algorithm

The following lemma shows that if a time interval is greater than λ, we only consider a single caching location.

Lemma 1. *In any optimal schedule, if $\mu\delta t_{i,i+1} > \lambda$ then only one server will cache the data in the interval $[t_i, t_{i+1}]$.*

Proof. Suppose there are two or more caching intervals covering $[t_i, t_{i+1}]$ in some optimal solution. By Observation 2, r_{i+1} will be served by exactly one of the following choices,

1) the cached copy on server s_{i+1}
2) a transfer $Tr(s^j, s_{i+1}, t_{i+1})$ where s^j holds one of the overlapping caching intervals and $s^j \neq s_{i+1}$.

In either case, there will be another cached copy on another server s^k, $H(s_k, t_k, t_j)$ which spans the interval $[t_i, t_{i+1}]$. Since there are no requests between r_i and r_{i+1}, $t_k < t_i$ and $t_j > t_{i+1}$. If we transfer $Tr(s_{i+1}, s_k, t_{i+1})$ and eliminate the interval $[t_i, t_{i+1}]$ on server s_k, we will reduce the cost by $\mu\delta t_{i,i+1} - \lambda > 0$, contradicting the assumption that the solution was optimal.[2] □

On the other hand, if a server interval is short enough, then it is always an advantage to cache the data copy on the server for that interval, even if this means caching at two or more locations.

Lemma 2. *For any i where $\mu\sigma_i < \lambda$, $H(s_i, t_{p(i)}, t_i)$ is a part of every optimal schedule.*

Proof. Consider a request r_i where $\mu\sigma_i < \lambda$. Any solution which uses a transfer to supply request r_i can be improved by $\lambda - \mu\sigma_i$ by replacing the transfer with the cached copy on s_i for the interval $[t_{p(i)}, t_i]$. □

Fig. 2 illustrates the both cases given $\mu = 1$ and $\lambda = 1$. Let's see r_4 and r_5, their time difference is 1.2, which is greater than $\lambda = 1$. Thus, r_5 is satisfied by the cached copy on s_3 plus a transfer from s_3 to s_2. According to Lemma 1, it is unlikely to keep other copies on s_1 and $s2$ that span across $[t_4, t_5]$ in optimal schedule. The copy on s_3 is the unique copy in the interval $[t_4, t_5]$. In contrast, $\sigma_6 = 0.2$, which is less than λ. According to Lemma 2, it is more advantageous to use the copy held on s_2 to satisfy r_6.

Definition 4 (SSI and SR). *In view of Lemma 2, we define the set of requests on* short server intervals (SSI) *by $SR = \{r_i : \mu\sigma_i < \lambda, i > 0\}$.*

Definition 5 (Marginal Cost). *We define the* marginal cost *$\Delta_\Psi(i)$ of r_i in a schedule $\Psi(i)$ as the difference in the cost of $\Psi(i)$ minus the cost of the sub-schedule $\Psi^{(-1)}(i)$, that is*

$$\Delta_\Psi(i) = \Pi(\Psi(i)) - \Pi(\Psi^{(-1)}(i)).$$

Referring to Lemma 2 we obtain

Observation 3. For any optimal schedule $\Psi^*(i)$, if $r_i \in SR$ then the marginal cost of r_i in $\Psi^*(i)$ is $\Delta_\Psi(i) = \mu\sigma_i$.

On the other hand, all non-SSI requests (i.e., $r \notin SR$) require a transfer or a caching over an interval greater than λ, and thus we have

2. We can modify the improved solution without cost to match Observation 1 by storing on server s_{i+1} until the nearest future request on s_k before transferring.

Observation 4. For any optimal schedule $\Psi^*(i)$, if $r_i \notin SR$ then the marginal cost of r_i in $\Psi^*(i)$ is at least λ, i.e., $\Delta_\Psi(i) \geq \lambda$.

We still use Fig. 2 to illustrate these concepts. In the figure, $SR = \{r_6, r_8\}$ as σ_6 and σ_8 (again $\mu = 1$) are less than $\lambda = 1$. Then the marginal cost of r_6 and r_8 in optimal schedules $\Psi^*(6)$ and $\Psi^*(8)$ are $\sigma_6 = 0.2$ and $\sigma_8 = 0.7$, respectively. Clearly, for those non-SSI requests $\{r_1, r_2, r_3, r_4, r_7\}$, they can be satisfied either by the copies in the same servers that are kept for a period with cost greater than σ_i ($\{r_4, r_7\}$) or by the copies in different servers ($\{r_1, r_2, r_3, r_5\}$). In both cases, the marginal cost is at least λ.

To this end, we first obtain a lower bound on the marginal costs to satisfy each individual request by combining above observations, which is defined by

Definition 6 (Marginal Cost Bound). *The marginal cost bound of request r_i is $b_i = \min\{\lambda, \mu\sigma_i\}, 1 \leq i \leq n$.*

Based on the marginal cost bound, we can further have a lower bound on the total costs to satisfy a request sequence, which is defined by

Definition 7 (Running Bound). *The running bound of the marginal costs up to request i is $B_i = \sum_{j=1}^{i} b_j$.*

The following table shows the marginal cost bounds and running bounds for each request in the example in Fig. 2.

i	0	1	2	3	4	5	6	7	8
b_i	0	1	1	1	1	1	0.2	1	0.4
B_i	0	1	2	3	4	5	5.2	6.2	6.6

Definition 8 (Optimal Cost $C(i)$). *We define $C(i), 1 \leq i \leq n$ to be the cost of the optimal schedule $\Psi^*(i)$. Recall that r_0 is a boundary variable we created, with $C(0) = 0$, so $C(i)$ is defined for $0 \leq i \leq n$.*

Clearly, given the definitions of B_i and $C(i)$ for $1 \leq i \leq n$, we can have the following observation as B_i is only a lower bound of the optimal cost, that is $B_i \leq C(i), 1 \leq i \leq n$.

Our goal is to create a recurrence for $C(i)$ that we can solve dynamically. To this end, we first prove the following lemma,

Lemma 3. *If $\Psi^*(i)$ is an optimal schedule in which the last operation is a transfer $Tr(s_j, s_i, t_i)$ then $\Psi^{(-1)}(i)$ is an optimal schedule up to request r_{i-1} (i.e., $\Psi^{(-1)}(i) = \Psi^*(i-1)$).*

Proof. If the optimal $\Psi^*(i)$ ends in a transfer, it must cache the unique data copy on the interval $[t_{i-1}, t_i]$ on some server other than s_i. All transfers are of equal cost, so one optimal extension to the sub-schedule is to cache $H(s_{i-1}, t_{i-1}, t_i)$ and then transfer $Tr(s_{i-1}, s_i, t_i)$. If $\Psi^{(-1)}(i)$ were not optimal this would lead to a contradiction. □

Given Observation 2, we only need to consider two cases that r_i is served, either by the cache on s_i or by the immediate transfer from r_{i-1}. The next lemma covers the easy case of our recurrence.

Lemma 4. *If the conditions of Lemma 3 hold, then*

$$C(i) = C(i-1) + \mu\delta t_{i-1,i} + \lambda.$$

Proof. This is just the sum of the optimal cost up to r_{i-1} plus the cost of caching and transfer. □

Now, we consider the non-trivial case that the optimal schedule $\Psi^*(i)$ uses the cached data copy on server s_i to satisfy r_i. Unlike the transfer case where the last data transfer $Tr(s_j, s_i, t_i)$ does not impact the optimality of $\Psi^{(-1)}(i)$, in this case, the last $H(s_i, t_{p(i)}, t_i)$ may impact all the requests made in $[t_{p(i)}, t_{i-1}]$ since a cache is extended from $t_{p(i)}$ to t_i which allows the requests to re-adjust the sources of the data item (e.g., a cache may be changed to transfer for cost reduction). As a consequence, no request $r_j, 0 < j < i$ is guaranteed to be optimal anymore for the sub-schedule of $\Psi^*(i)$ with respect to the interval $[t_1, t_{i-1}]$. To deal with this, we define an auxiliary recurrence that helps compute $C(i)$ in this case.

Definition 9 (Semi-Optimal Cost $D(i)$). *We define $D(i)$ to be the semi-optimal cost of a schedule $\Psi(i)$ in standard form (see Observation 1) under the condition that r_i is served by cache on server s_i. Clearly, $C(i) \leq D(i)$.*

To see the efficacy of this definition we note the following.

Observation 5. In a $\Psi(i)$, if s_i has a cached copy at t_i, then the cache extends from time $t_{p(i)}$, that is, the cache is $H(s_i, t_{p(i)}, t_i)$.

Observation 5 follows from the standard form requirement that no transfer ends at point that is not a request.

We can now complete the recurrence for $C(i)$ in terms of the not yet completed $D(i)$ since the optimal will either use cache or transfer (Observation 2)

$$C(i) = \begin{cases} 0 & i = 0 \\ \min\{D(i),\ C(i-1) + \mu\delta t_{i-1,i} + \lambda\} & \\ & 1 \leq i \leq n. \end{cases} \quad (1)$$

Recall that we added boundary points to our problem definition, and we extend here by defining base cases $D(i) = +\infty, i < 1$. These together with the infinite negative starting values of these intervals prevent us from using $D(i)$ as the cost of the first request on any server. That is, the first request on any server except s^1 will have to be served by a transfer. Recall that the first request on s^1 is r_0 with cost 0.

The basic idea of auxiliary recurrence is to establish the relationships between $D(i)$ and certain $C(\kappa)$ that has been available whereby the most recent $C(i)$ can be computed. To this end, we define the following concepts.

Definition 10 (Reduced Schedule). *Let $\Psi'(i)$ be a conditional optimal schedule with $H(s_i, t_{p(i)}, t_i)$ as the final cache H. Given its cost of $D(i)$, we define the* reduced schedule *$\Psi^{(-H)}(i)$ to be $\Psi'(i)$ with the cache H removed.*

Note that a reduced schedule may not be a proper schedule for requests in the interval $[t_{p(i)}, t_i]$, since transfers from the server which held the removed cache will no longer be valid. We now look for the last cache in $\Psi^{(-H)}(i)$ that covers $r_{p(i)}$ (again, such a cache has potentials to satisfy $r_{p(i)}$ by a transfer), defined as the *pivot index*, and extend it to get our recurrence for $D(i)$. To this end, we first define a concept of *cover index set* for $\Psi^{(-H)}(i)$, and then the *pivot index*.

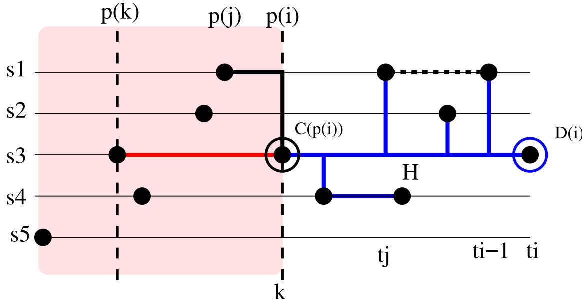


Fig. 3. An example of the trivial case when $\kappa \leq p(i)$. $H(s_i, t_{p(i)}, t_i)$ as the final cache H impacts how the requests in $[t_{p(i)}, t_{i-1}]$ are served (shown in bold blue line).

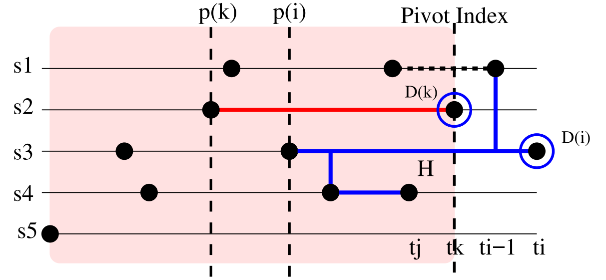


Fig. 4. An example of the non-trivial case when there are some caches in $\Psi^{(-H)}(i)$ span across $t_{p(i)}$ ($\pi(i) \neq \varnothing$). $H(s_i, t_{p(i)}, t_i)$ as the final cache H impacts how the requests in $[t_{p(i)}, t_{i-1}]$ are served (shown in bold blue line). Additionally, the figure also shows the concept of pivot index κ.

Definition 11 (Cover Index Set). *We define* cover index set *$\pi(i)$ with respect to $\Psi^{(-H)}(i)$ as follows:*

$$\pi(i) = \{k | H(s_k, t_{p(k)}, t_k) \subset \Psi^{(-H)}(i), p(k) < p(i) \leq k < i\}.$$

Definition 12 (Pivot Index κ). *The* pivot index *κ with respect to $\Psi^{(-H)}(i)$ is defined by either 0 or the maximum in $\pi(i)$, depending on whether or not $\pi(i) = \varnothing$, i.e.,*

$$\kappa = \begin{cases} 0 & \pi(\mathrm{i}) = \varnothing \\ \max\{\pi(i)\} & \text{Otherwise.} \end{cases}$$

The definition of $\kappa \neq 0$ is important as it signifies the last request in $[t_{p(i)}, t_{i-1}]$ that is served by the cache $H(s_k, t_{p(k)}, t_k)$ other than the transfer from $H(s_i, t_{p(i)}, t_i)$ in $\Psi'(i)$, which forms the basis for the $D(i)$ recurrence. We distinguish two cases: 1) $\kappa \leq p(i)$, and 2) $\kappa > p(i)$. The first is the boundary case, which is trivial.

Lemma 5. *For the pivot index κ as defined in Definition 12, if $\kappa \leq p(i)$ then the optimal restricted cost $D(i) = C(p(i)) + \mu\sigma_i + B_{i-1} - B_{p(i)}$.*

Proof. In $\Psi^{(-H)}(i)$ all requests up to $t_{p(i)}$ must still be satisfied. $C(p(i))$ is a lower bound on any schedule satisfying this. The cost of the cache H is $\mu\sigma_i$, and with this cache we can satisfy all requests $r_j, p(i) < j < i$ using transfers and short cache intervals by a cost of $B_{i-1} - B_{p(i)}$.[3] Since this difference is a lower bound on serving these requests, the total is optimal under the stated conditions of the lemma. □

An illustrative example of the trivial case is shown in Fig. 3 where $\kappa \leq p(i)$. According to Observation 2, $r_{p(i)}$ could be served by a single transfer ending at $r_{p(i)}$ (in bold black line), say the cache on s_1 in the example. In this case, $B_{i-1} - B_{p(i)}$ might be overestimated as $\mu\sigma_j$ in $b_j = \min\{\lambda, \mu\sigma_j\}$ has been reduced by an amount of $\mu\delta t_{p(j),p(i)}$ on s^1. However, such reduction does not compromise the correctness of the algorithm since the examination of $H(s^1, t_{p(j)}, t_j)$ in Recurrence (2) will get rid of the overestimate.

Now let's examine the non-trivial case that $\kappa > p(i)$. In this case, both $H(s_\kappa, t_{p(\kappa)}, t_\kappa)$ and $H(s_i, t_{p(i)}, t_i)$ are in the final schedule $\Psi'(i)$ as shown in Fig. 4 as an example, then we have

Lemma 6. *For κ as defined in Definition 12, if $\kappa \neq 0$ then the optimal restricted cost $D(i) = D(\kappa) + \mu\sigma_i + B_{i-1} - B_\kappa$*

Proof. We can construct a schedule up to r_κ equal in cost to $\Psi^{(-H)}(i)$ by modifying $\Psi^{(-H)}(i)$. In $\Psi^{(-H)}(i)$, all requests up to $t_{p(i)}$ are satisfied. All requests $r_h, p(i) < h \leq \kappa$ which were satisfied with a transfer from s_i in $\Psi(i)$ will now use a transfer from s_κ. $D(\kappa)$ is a lower bound on the cost of this schedule; that is, $D(\kappa) \leq Cost(\Psi^{(-H)}(i))$. Since $B_{i-1} - B_\kappa$ is a lower bound on adding the requests $r_h, \kappa < h < i$, and we must add $\mu\sigma_i$ to cover the interval $[t_{p(i)}, t_i]$. Then, we see that $D(\kappa) + \mu\sigma_i + B_{i-1} - B_\kappa \leq Cost(\Psi'(i)) = D(i)$.

If we start with a restricted optimal schedule to r_κ with cost $D(\kappa)$, then we can similarly construct a restricted schedule to r_i with cost $D(\kappa) + \mu\sigma_i + B_{i-1} - B_\kappa$, and thus the lemma follows. □

By combining these lemmas, we enumerate all the request indexes on the interval $[t_{p(i)}, t_{i-1}]$ to derive $D(i)$ recurrence as follows:

$$D(i) = \begin{cases} +\infty & -m \leq i \leq 0 \\ \min\begin{cases} C(p(i)) + \mu\sigma_i + B_{i-1} - B_{p(i)} \\ \min_{j\in\pi'(i)}\{D(j) + \mu\sigma_i + B_{i-1} - B_j\}, \end{cases} \end{cases} \tag{2}$$

where $\pi'(i) = \{k | H(s_k, t_{p(k)}, t_k) \text{ that } p(k) < p(i) \leq k < i\}$ for $1 \leq i \leq n$.

Theorem 1. *With a homogeneous cost model, Recurrences (1) and (2) correctly compute the minimum cost of the data caching problem within $O(mn)$ time and space complexity.*

Proof. The optimality of the algorithm can be directly derived from Lemmas 4, 5, and 6 by ruling out the overestimate in Lemma 5. As for the time and space complexity, we can sweep the off-line sequence first so that for each request r_i, its index set $\pi'(i)$ can be computed in $O(m)$ time and space complexity ($|\pi'(i)| = m$). Since there are n requests in total, we then have $O(mn)$ time and space in pre-processing stage. Given $\pi'(i)$ for each r_i, Recurrence (2) can also be computed in $O(mn)$ time and space.

Suppose $r_{p(i)}$ is served by a single transfer from the cache at $t_{j'}$ on s_j (Observation 2), and the most recent request on s_j after the cache is r_j, that is $j' = p(j)$, we consider several cases (reference Fig. 3).

When $\lambda \geq \mu\sigma_j$, then $b_j = \mu\sigma_j$, which will incur the overestimate of $B_{i-1} - B_{p(i)}$ in Lemma 5 by an amount of $\mu\delta t_{p(j),p(i)}$. However, such overestimated $D(i)$ can be

3. This value might overestimate the cost when $\kappa = 0$. However, the overall optimal cost can be corrected by Recurrence (2) when those $D(j)$s that satisfy $p(j) < p(i) < j < i$ are enumerated.

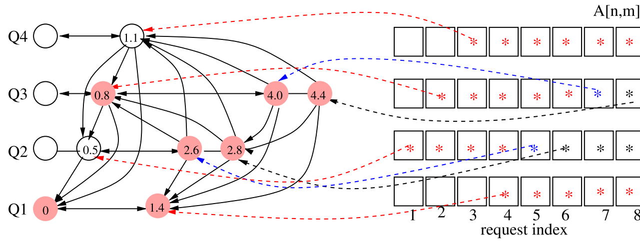
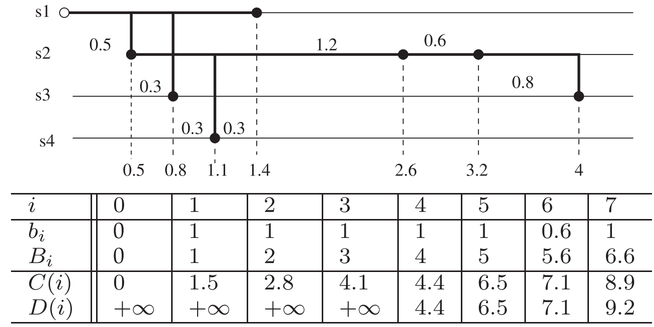


Fig. 5. An example to show how the efficient implementation of the proposed algorithm addresses the data staging problem in Fig. 2 where the computed cache intervals on each server are also marked, and each pointer, represented by "*" in different colors, is kept up to t_8 (updated when a new request made on that server is processed).



i	0	1	2	3	4	5	6	7
b_i	0	1	1	1	1	1	0.6	1
B_i	0	1	2	3	4	5	5.6	6.6
$C(i)$	0	1.5	2.8	4.1	4.4	6.5	7.1	8.9
$D(i)$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	4.4	6.5	7.1	9.2

Fig. 6. An optimal schedule for an off-line request sequence (solid dots) is shown in bold lines, and the marginal cost bounds (b_i), running bounds (B_i), as well as costs $C(i)$ and $D(i)$ are also presented in the table at bottom.

ruled out by considering j as the pivot index (i.e., $H(s_j, t_{p(j)}, t_j)$ covers $rp(i)$) in Lemma 6. The same arguments can also be applied to the case when $\mu(\sigma_j - \delta t_{p(j),p(i)}) \leq \lambda \leq \mu\sigma_j$.

Finally, when $\lambda < \mu(\sigma_j - \delta t_{p(j),p(i)})$, then $b_j = \lambda$, and thus, there is no overestimate in $B_{i-1} - B_{p(i)}$. Overall, Equation (2) correctly computes $D(i)$. By combining Equations (1) and (2), we conclude the theorem. □

4.2 Efficient Implementation

4.2.1 An $O(mn)$ Time&Space Implementation

We assume here that the requests are available ordered by time, and we use a uniform cost random access model of computation.

Recurrences (1) and (2) define a recurrence system that allows us to compute the optimal cost. Using a sweep algorithm, we can compute this value by incrementally indexing through the requests from 1 to n, storing $C(i)$ and $D(i)$ for each request. A straightforward implementation should run in $O(n^2)$ time, which is dominated by the need to check at most $O(n)$ previous values in the computation of $D(i)$ as indicated in Recurrence (2).

However, a closer look at Recurrence (2) indicates that at each i we need check at most one interval on each server (since $|\pi(i)| \leq m$, we do not need to compute $\pi(i)$ for $1 \leq i \leq n$ in our algorithm), provided we can efficiently find the interval on each server containing time $t_{p(i)}$ (reference Fig. 4). To do this we create the following structures in a pre-scan of the requests. For each server, $s^j, 1 \leq j \leq m$, we create a doubly linked list Q_j which is initialized by the dummy boundary request, and a matrix $A[n, m]$ of pointers. As r_i is considered, $1 \leq i \leq n$, it is added to the list Q_{s_i}, and $A_{i,j}$ is assigned to the current last element of Q_j for $1 \leq j \leq m$. Then, for each request node $r_i = (s_i, t_i)$ in Q_j, a pointer is set up for each other server s_k that points to its most recent request node $r_k = (s_k, t_k)$, which could be obtained from $A[i, k], k \neq j$. Given that each node in $Q_j, 1 \leq j \leq m$, has $O(m)$ space, the total data space in pre-scan thus takes $O(mn)$ time and space.

During the next pass over the requests to compute the recurrences, these pointers can be used to precisely identify each of the intervals required by Recurrence (2) in $O(m)$ time per request. Thus this pass also takes $O(mn)$ time.

Fig. 5 is an example to how the data structures are organized in efficient implementation of the proposed algorithm for the data staging problem in Fig. 2. During the computation of the recurrences, the algorithm follows the pointer of recent request r_i in $A[i, j]$ (e.g., $A[7, 3]$) to find the current last element of Q_j (e.g., request node 4.0) and then go back along the backward link to get its previous request node which records $t_{p(i)}$ (e.g., 0.8). Then, by following the $m = 1$ pointers, each for one server, the required interval on each server by Recurrence (2) can be identified in $O(1)$ (e.g., $\{[0, 1.4], [0.5, 2.6], \varnothing, \varnothing\}$ in our example).

4.2.2 A Running Example

For illustration purpose, we present a running example of the algorithm for an off-line demand sequence shown in the space-time diagram Fig. 6 where $m = 4$, $n = 8$ and each time instance for the requests is also marked. The data item is initially located at s^1 given $\lambda = 1$ and $\mu = 1$.

To facilitate the computation, we can pre-scan the sequence and compute the marginal cost bound b_i, the running bound B_i as well as $\pi'(i)$, for each individual request r_i, $1 \leq i \leq n$, in advance as we illustrated before. With the information of B_i, we can further compute the $C(i)$ and $D(i)$.

At $t_0 = 0$, $C(0)$ and $D(0)$ are initialized by 0 and $+\infty$, respectively. Since the first request on any server except s^1 will have to be served by a transfer, $D(1) - D(3)$ are set by $+\infty$, while $C(1) = \min\{D(1), C(0) + 1 + 0.5\} = 1.5$, $C(2) = \min\{D(2), C(1) + 0.3 + 1\} = 2.8$, $C(3) = \min\{D(3), C(2) + 0.3 + 1\} = 4.1$. In order to compute $C(4)$, we have to compute $D(4)$ first. $D(4) = C(0) + 1.4 + 3 - 0 = 4.4$ and $C(4) = \min\{D(4), C(3) + 0.3 + 1\} = 4.4$. Now we consider to compute the final optimal value $C(7)$. To this end, according to Recurrence (2), we have $D(7) = 9.2$ and $C(7) = 8.9$ because

$$D(7) = \begin{cases} C(2) + 3.2 + 5.6 - 2 = 9.6 \\ \min \begin{cases} 4.4 + 3.2 + 5.6 - 4 = 9.2, \\ 6.5 + 3.2 + 5.6 - 5 = 10.03, \\ 7.1 + 3.2 + 5.6 - 5.6 = 10.03, \end{cases} \end{cases}$$

and $C(7) = \min\{D(7), C(6) + 0.8 + 1\}$. The full vectors of C and D are listed in the table of Fig. 6.

The optimal schedule $\Psi^*(7)$ can be reconstructed by recursively backtracking the vectors of C and D up to the initial configuration at $t = 0$. As such, we can phase by phase steer to the final optimal results as shown in Fig. 6. Since the transfer cost is a constant, we can only store the cache schedule by marking the responding request nodes in $Q_j, 1 \leq j \leq m$ (Fig. 6).

Based on the description of this algorithm, we can easily obtain the best result 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).

5 An 3-Competitive Online Algorithm

Although the off-line algorithm for the problem can efficiently minimize the cost in an optimal way, it is only useful in certain scenarios when the whole sequence of requests is pre-defined and fully available in advance, which is not always feasible in reality. As a result, online algorithm is necessary. In this section, we first present an 3-competitive online algorithm for this problem, and then make a competitive analysis on it. We close this section with a description of some variants in the implementation of this algorithm.

5.1 The Algorithm

The algorithm is built on a concept of *anticipatory caching* that allows the copy migrated to a sever to anticipatorily keep active for another period of $\Delta t = \lambda/\mu$ after it serves the most recent request at time t. The rationale behind this idea is that if the next request is coming no later than $t + \Delta t$, it should be served by caching as the caching cost is not more than λ; otherwise, the copy is not worthwhile to be kept, and the request is served by a transfer from other server, instead. In this way, we can enable the online algorithm to mimic the optimal off-line algorithm as close as possible. The algorithm operates on a per-epoch basis along the timeline, and each *epoch* is composed of n transfers. We call this online algorithm *Anticipatory Caching (AC)* algorithm, which operates as follows in each epoch.

1) use variables c and r to record the number of active copies and the number of transfers in current epoch, respectively. Initially, $c \leftarrow 1$ and $r \leftarrow 0$, and the data is located at s^1;
2) use a counter array of $C[m]$, initialized by zero, to maintain the copy expiration information of each server in current epoch, e.g., $C[i] \leftarrow t_i$ indicates the copy on s^i will expire at $t_i, 1 \leq i \leq m$.
3) when a new request r_i on s^j is coming at t_i:
   - for s^j, if $t_i \in [t_{p'(i)}, t_{p'(i)} + \Delta t)$ and $C[j] \neq 0$, then serve r_i by the copy on s^j, and then update $C[j] \leftarrow t_i + \Delta t$. Otherwise, serve r_i by a transfer from $s^k, k \neq j$ where r_{i-1} is made, and update $C[j] \leftarrow t_i + \Delta t$ and $r \leftarrow r + 1$;
   - for $s^k, k \neq j$, if $s^k(C[k] \neq 0)$ performs a transfer at t_i, then update $C[k] \leftarrow t_i + \Delta t$.
   - if $r = n$ then the current epoch is completed, and the next epoch is started with $c \leftarrow 1$ and $r \leftarrow 0$, $C[m] \leftarrow 0, 1 \leq i \leq m$, and the data located at s^j.
4) when events of copy expiration happen at t_i:[4]
   - $c \leftarrow$ the number of active copies,
   - if there are two events on s^j and s^k at the same time, and $c > 2$, then $c \leftarrow c - 2$ and delete the copies on s^j and s^k (i.e., $C[j] \leftarrow 0, C[k] \leftarrow 0$).
   - if there are two events on s^j and s^k, but $c = 2$ (the last two copies), then delete the copy in source server, say s^j, to break the tie, keep the copy in target server s^k (i.e., $C[j] \leftarrow 0, C[k] \leftarrow t_i + \Delta t$), and finally set $c \leftarrow 1$,
   - in other cases, if there is a single event on s^j and $c > 1$, just delete the copy on s^j (i.e., $C[j] \leftarrow 0$) and set $c \leftarrow c - 1$. Otherwise if there is a single event on s^j but $c = 1$, then extend the copy expiration time on s^j to $t_i + \Delta t$ (i.e., $C[j] \leftarrow t_i + \Delta t$).

4. According to AC, there are at most two expiration events resulted from a transfer that could occur at the same time.

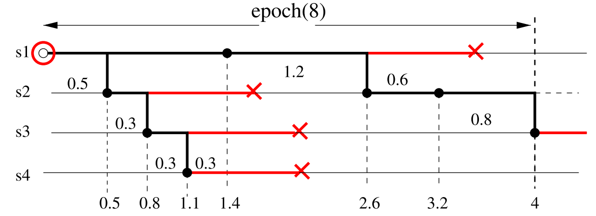


Fig. 7. An example of the online *AC* algorithm where the schedule for an epoch with size of 5 is illustrated.

An illustrative example of this algorithm for a single epoch with 5 transfers is shown in Fig. 7 where each copy survives another anticipatory period of time at most $\Delta t = \lambda/\mu$ for incoming requests. Based on the algorithm and this example, we can easily make the following observation:

Observation 6. For each request r_i at t_i on s^j, the online *AC* algorithm satisfies the following properties:

1) when $\mu\delta t_{p'(i),i} < \lambda$, r_i is always served by caching on s^j;
2) when $\mu\delta t_{p'(i),i} \geq \lambda$,
   - if $t_{p'(i)} < t_{i-1}$, r_i is served by the copy created at t_{i-1} on $s^k, k \neq j$ via a transfer where $t_{p'(i)} < t_{i-1}$.
   - otherwise, $t_{p(i)} = t_{i-1}$, r_i is served by the copy created at t_{i-1} on s^j.

here, $t_{p'(i)}$ is the most recent time instance that a request or a transfer (to other server) happen before r_i at t_i on the same server, say s^j in this observation (e.g., $p'(6) = 3$ and $t_{p'(6)} = 0.8$ on s^2 in Fig. 7). Clearly, $p(i) \leq p'(i) \leq i - 1$. 2) is correct since according to the algorithm, the latest copy created at t_{i-1} is always available to r_i by continuously expanding its active periods.

5.2 Competitive Analysis

In analyzing online algorithms, the *competitive ratio* (CR) is always adopted to measure the quality of solution, which is defined below for online algorithm A (if we ignore the additive constant term)

$$\gamma_A = \sup_{\mathcal{R}} \frac{Cost(A, \mathcal{R})}{Cost(OPT, \mathcal{R})}. \tag{3}$$

γ_A is essentially the approximation ratio of algorithm A.

For the sake of easy analysis, we transform the AC schedule in an epoch into an equivalent shadow schedule, called *Double Transfer (DT)* schedule, that has exactly the same cost with the AC schedule. To this end, we denote the set of AC costs on s^j as Ω_j, and have the following definition:

Definition 13 (Double Transfer Schedule). *The double transfer schedule can be obtained from the AC schedule by performing the following transformations for each AC cost $\omega_j^i \in \Omega_j$ on $s^j, 1 \leq j \leq m$ (note that $\omega_j^i \leq \lambda$):*

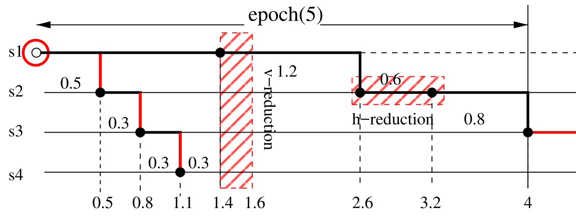


Fig. 8. An example of the DT schedule where the red circle and transfer lines represent the initial cost and the transfer cost of the data that are increased by corresponding $\omega^i_j \in \Omega_j, 1 \leq j \leq 4$ ($\omega^2_2 = 0$ and $\omega^2_3 = 0$ starting at t_4). Additionally, the two types of reductions are also showed in shaded rectangles.

1) *if $j = 1$ and $i = 1$, increase the initial cost on s^1 from 0 to ω^1_1;*
2) *otherwise, remove ω^i_j and add it to the weight of the most recent transfer edge to s^j, whose value is increased to $\lambda + \omega^i_j$ which is less than or equal to 2λ.*

The transformation is reasonable as each ω^i_j on s^j corresponds to an incoming transfer edge in the schedule. Therefore, for any online sequence $\mathcal{R}$, we can have $Cost(DT, \mathcal{R}) = Cost(AC, \mathcal{R})$ in $O(mn)$ time.

An example of the schedule produced by the *DT* algorithm after the transformation is shown in Fig. 8. Given ω^2_2 and ω^2_3 starting at t_4 are equal to zero in the example, one can verify they have the same scheduling strategies and the total costs.

With these results, we have the following lemma that shows if a time interval is greater than λ, we can only consider a single caching location for both *DT* and any optimal algorithm (*OPT*).

Lemma 7. *In the* DT *schedule, if $\mu\delta t_{i-1,i} > \lambda$ then only one server will cache the data in $[t_{i-1}, t_i]$.*

Proof. According to the *DT* schedule, if $\mu\delta t_{i-1,i} > \lambda$, then $t_{p'(i),i} \leq i - 1$, and $\delta t_{p'(i),i} > \lambda$. According to *DT*, r_i is served by the copy created at t_{i-1} on $s^k, k \neq j$ via a transfer. If there is another cached copy on another server that spans the interval $[t_{i-1}, t_i]$, it must contradict the algorithm by following the same arguments for *OPT* in Lemma 1. As a result, no more than one copies in *DT* will be active in parallel in $[t_{i-1}, t_i]$. □

Based on Lemma 7, we can make the following reduction on both schedules.

Definition 14 (V-Reduction). *For each interval $[t_{i-1}, t_i], i \in [1, n]$ that satisfies $\mu\delta t_{i-1,i} > \lambda$ in both schedules (*DT *and* OPT*), we can reduce its weight to $\mu\delta t_{i-1,i} = \lambda$ by setting $\mu\delta t_{i-1,i'} = 0$ where $t_{i'} < t_i$. We call it* V-Reduction.

As such, for any $r_i \in \mathcal{R}$ in an epoch, we have $\mu\delta t_{i-1,i} \leq \lambda$ in both *DT* and *OPT* after the reduction. An example of the *v-reduction* is shown in Figs. 8 and 9.

Moreover, we have the following lemma to show that each request in $SR = \{r_i : \mu\sigma_i < \lambda, i > 0\}$ is satisfied in the same way by both *DT* and any *OPT* schedules.

Lemma 8. *For any i where $\mu\sigma_i < \lambda$, $H(s_i, t_{p(i)}, t_i)$ is a part of the* DT *schedule.*

Proof. A direct conclusion from Observation 6. □

Based on Lemma 8, we can make the following reduction on both schedules.

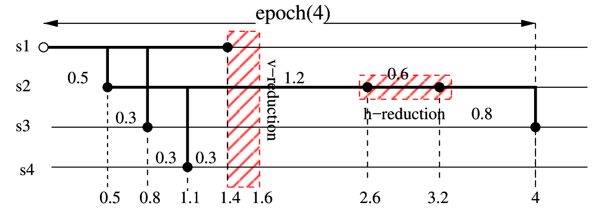


Fig. 9. The optimal schedule with 4 transfers after the two types of reductions.

Definition 15 (H-Reduction). *The caching cost of each request in SR can be removed by setting the cost to zero for both schedules. We call it* H-Reduction.

As a result, we have for any $r_i \in \mathcal{R}$ in an epoch, $\mu\sigma_i \geq \lambda$ in both *DT* and *OPT* after the reduction. We can observe it by comparing the *h-reductions* in Figs. 8 and 9.

With these two reductions, we can reduce both *DT* and *OPT* by first applying *h-reduction* to ensure for any $r_i \in \mathcal{R}$ in an epoch, $\mu\sigma_i \geq \lambda$, and then performing *v-reduction* for all intervals $\{[t_{i-1}, t_i] | \mu\delta t_{i-1,i} > \lambda, i \in [1, n]\}$. Let DT' and OPT' denote the *reduced DT* and *OPT* schedules, we can make the following observation, note that $\delta t'_{i-1,i}$ and σ'_i are defined with respect to $\mathcal{R}'$ in the sequel, where $\mathcal{R}' = \mathcal{R} \setminus SR$.[5]

Observation 7. In both DT' and OPT', for any $r_i \in \mathcal{R}$ in an epoch, we have $\mu\delta t'_{i-1,i} \leq \lambda$ and $\mu\sigma'_i \geq \lambda$.

With above results, we can have our main theorem:

Theorem 2. *The anticipatory caching (*AC*) algorithm is 3-competitive.*

To prove the theorem, we first have the following lemma to show an upper bound of $Cost(DT', \mathcal{R}')$.

Lemma 9. *For any online sequence $\mathcal{R}$ in an epoch, Cost $(DT', \mathcal{R}')$ is upper bounded by $3n'\lambda$ where $n' = |\mathcal{R}'|$.*

Proof. Since for DT', the schedule reduction results in $\mu\delta t_{i-1,i} \leq \lambda$ for any request $r_i \in \mathcal{R}'$, we then have

- if $\mu\delta t'_{p'(i),i} < \lambda$, r_i is served by caching at cost of less than λ.
- Otherwise, r_i is served by the copy created at t_{i-1} on $s^k, k \neq j$ via a transfer at most cost of 2λ. Since $\mu\delta t'_{i-1,i} \leq \lambda$, the cost to serve r_i would be at most 3λ.

Overall, the total cost to serve the whole sequence $\mathcal{R}'$ in an epoch is at most $3n'\lambda$. □

Now, we estimate the lower bound of $Cost(OPT', \mathcal{R}')$ and have the following lemma,

Lemma 10. *For online sequence $\mathcal{R}$, $Cost(OPT', \mathcal{R}')$ is lower bounded by $n'\lambda$ where $n' = |\mathcal{R}'|$.*

Proof. According to Definition 7, $Cost(OPT', \mathcal{R}') \geq B'$, where the running bound B' is defined as $B' = \sum_{i=1}^{n'} b'_i = \sum_{i=1}^{n'} \min\{\lambda, \mu\sigma'_i\}$ with respect to $\mathcal{R}'$ after the h-reduction from $\mathcal{R}$. Based on Observation 7, we have $\mu\sigma'_i \geq \lambda$. Then, $Cost(OPT', \mathcal{R}') \geq B' = \sum_{i=1}^{n'} b_i = \sum_{i=1}^{n'} \min\{\lambda, \mu\sigma'_i\} = n'\lambda$. □

5. We can equivalently view the reduced schedules as those working on $\mathcal{R}'$ after the *h-reduction*.

With the above results, we can prove Theorem 2 as follows since both DT and OPT schedules are reduced by the same amount of costs in serving any online sequence $\mathcal{R}$, we have

$$\begin{aligned}\frac{Cost(AC,\mathcal{R})}{Cost(OPT,\mathcal{R})} &= \frac{Cost(DT,\mathcal{R})}{Cost(OPT,\mathcal{R})}\\ &\leq \frac{Cost(DT',\mathcal{R}')}{Cost(OPT',\mathcal{R}')} \leq \frac{3n'\lambda}{n'\lambda} = 3.\end{aligned} \tag{4}$$

Finally, $Cost(AC,\mathcal{R}) \leq 3 \cdot Cost(OPT,\mathcal{R})$ in an epoch. Since it can be repeated on each epoch, the AC algorithm is at most 3-competitive.

As with the case of the off-line algorithm, the designed online algorithm can be also applied to the multiple data items, each being scheduled independently. As such, we can also achieve the same competitive ratio.

5.3 Implementation

One of the key points in the design that may have impact the overall performance of the algorithm is how to select a copy from the current live ones to serve the next incoming request. By default, we limited the algorithm to select the one that is alive in the last active server (serving a request). Of course, given the homogeneous cost model, it makes no difference in selecting other live copies in terms of the transfer cost. However, different selection strategies may have different performance in reality since the lifespan of the selected copies in corresponding servers are extended even though they all bear the same competitive ratio.

To fully study the online algorithm, in addition to the default strategy, we also implement two other simple strategies for cost optimization purposes.

- *Balance*: The algorithm keeps track of all the live copies, and deliberately selects the one with the shortest remaining life to serve the next incoming request;
- *Priority*: The algorithm not only keeps track of the live copies, but also prioritizes the workload intensities of the servers in descending order. The algorithm tries to select the copy from the *active* server with the highest priority.

The rationale behind this implementation is that for the *balance* strategy, we intend to make a balance between the lifespans of the live copies in different servers for *balanced* case while for the *priority* strategy, we are in favor of the extensions to the lifespans of the copies caching in those servers with intense workloads for *unbalanced* case.

6 Simulation Studies

So far we have analyzed the performance of the proposed algorithms from theoretical perspective. However, the actual performance of these algorithms may exhibit diverse behaviors in reality. In this section, we conduct simulation-based study to show how the proposed algorithms, including both the online and off-line, behave in practice with respect to different incoming sequences of requests.

6.1 Experimental Setup

To reach our goal, we developed a *network caching simulator* in C++ to measure the average request cost as our major performance metric. The simulator efficiently implements the proposed algorithms for both the online and off-line cases as well as the models upon which the algorithms are built. As with the study in [14], we deliberately pulled out some properties and features of the network platform in the solver such as network traffic, bandwidth capacity, link latency and CPU power, and focus squarely on the factors closely related to our research goal since these properties and features, although important to model the reality, can be fully reflected in the monetary cost in our caching model.

The simulator is configured by several parameters including the size of network (the number of servers), the caching cost μ, the transfer cost λ, and other parameters regarding the model of incoming sequence of requests. We adopted the modeled requests since we can vary the generated sequence to fully evaluate the algorithms by simulating different situations in reality. 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 two-element tuple $< t, s >$, representing it is made at server s at time t. There exist studies on modeling the generation of various access sequences in different contexts and scenarios [29], [30]. In our particular case, we assume for each server the inter-arrival time of the requests follows the exponential distribution (determined by the *rate parameter* $\theta > 0$), which is often-used to model the web access patterns.

As for the experimental environment, we simulate a network of 50 to 200 fully connected nodes, and distinguish two cases, *balanced* and *unbalanced*, to manifest the situations in reality. In the former case, each node has a similar access intensity, which is determined by the value of $\theta > 0$, while in the later case, the access intensity is unevenly distributed among the nodes. The degree of load balancing is specified by *load-balance factor* β, which is defined as follows:

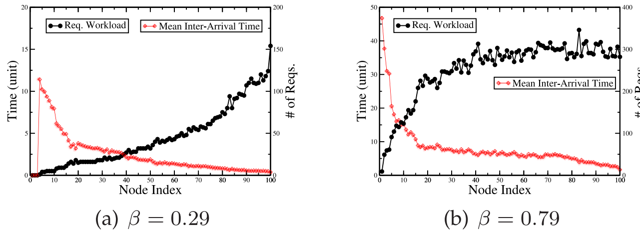
$$\beta = \frac{ave\{l_i : 0 \leq i \leq m\}}{max\{l_i : 0 \leq i \leq m\}}, \tag{5}$$

where $l_i : 0 \leq i < m$ is the number of requests made at $node_i$. Clearly, $\beta \leq 1$ and the larger, the more balanced.

To reflect the highly-skewed nature of the accesses in the unbalanced case, in our experiments we deliberately assume the values of θ among the nodes are also exponential distributed. The smaller the θ is, the more intensive the access requests. As such, a small number of nodes could be configured to experience highly intensive access workloads while others are light weighted. We will investigate the proposed algorithms, including both the online and off-line, for both cases in terms of performance and execution time. All experiments are conducted under Linux Ubuntu 14.04 (lucid) running on Intel Core2 CPU@3.16 GHZ with 4 GB Memory and 6 MB L2 Cache.

6.2 Simulation Results

In this section, we present our simulation results based on the experimental setup described above. We first characterize the workloads for our experiments, and then study the features of the off-line algorithm. We finally evaluate the online algorithm, together with its variants, by comparing it with its optimal off-line counterpart.



(a) $\beta = 0.29$ (b) $\beta = 0.79$

Fig. 10. The distributions of the number of requests as well as their mean inter-arrival times on per-node basis in an 100-node network for both the balanced ($\beta = 0.29$) and unbalanced ($\beta = 0.79$) cases.

6.2.1 Workload Characteristics

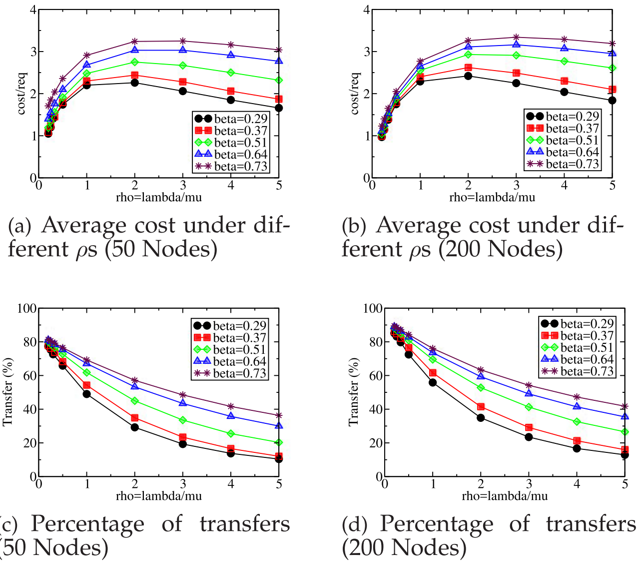
We present the characteristics of the workloads that are generated according to the described model for both the balanced and unbalanced cases. Fig. 10 illustrates how the number of requests, including their mean inter-arrival times, are generated before certain time thresholds (say $\tau = 2000$ time units), and distributed among the 100 nodes in an ascending order in the network to give different load balancing scenarios. Given the same requests, a similar observation can also be made for other sizes of the networks (not shown).

According to Fig. 10, the studied workloads can be truly characterized by the factor β to reflect the load balancing scenarios when they are served by the network service as we can observe that workloads are more evenly distributed in the $\beta = 0.79$ case than in the $\beta = 0.29$ case. Additionally, we also measure the mean inter-arrival times of the requests for each node as overlapped in Fig. 10, which reflects how intense a sequence of requests is served by each node, and determines the overall service cost.

6.2.2 The Off-Line Algorithm

We first investigate the impact of the ratio $\rho = \lambda/\mu$ on the behaviors of the off-line algorithm with respect to different βs. For fair comparisons, we deliberately enable $\lambda + \mu$ to have a fixed value, and change ρ to see how this ratio effects the cost on per-request basis for the network varied from 50 nodes to 200 nodes. Moreover, we also generate a set of request sequences by following the discussed methods, each representing a different workload distribution, characterized by β. Fig. 11 shows our experimental results on the changes of the average costs across different β values for different network scales when $\lambda + \mu = 6$ cost units and $\rho = \lambda/\mu$ is varied from 0.2 to 5, which cover a wide range of cloud storage pricing in reality [24].

From Figs. 11a and 11b, we can make three observations. First, as ρ grows up, the corresponding average cost of the requests for each β is changed in a *parabolic form* that it is quickly increased at the initial stage, and then gradually decreased afterwards for the networks with 50 and 200 nodes. This is not surprising as when the ratio ρ is low, the transfer is much cheaper than the caching, as thus in this situation, due to the nature of exponential distribution, the small number of live copies selected for the requests with large inter-arrival times will incur much caching cost, which in turn raises the average cost of the requests. As ρ continues to increase, the transfer will become more expensive, and thus, the number of transfers will be reduced and more copies are created, which are relatively cheap to serve the requests. As a result, the total cost, also the average cost for



(a) Average cost under different ρs (50 Nodes) (b) Average cost under different ρs (200 Nodes)

(c) Percentage of transfers (50 Nodes) (d) Percentage of transfers (200 Nodes)

Fig. 11. Impact of ρ on the performance behaviors of the off-line algorithm with respect to different load-balancing factor ρs for different network scales.

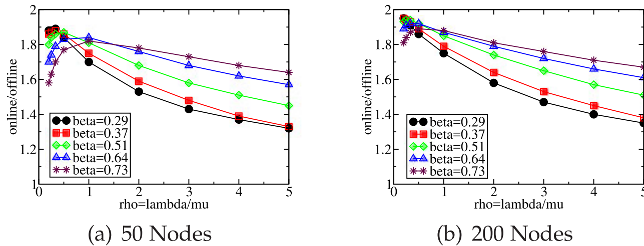
each request, is minimized. We can further evidence this observation by depicting how the percentages of the transfers are monotonically decreased (accordingly the percentages of the caching are monotonically increased) with the increase of ρ in Figs. 11c and 11d. From this experiment, we can derive that in reality to minimize the service cost, the values of λ and μ should not be defined close to each other as in this case it is hard for the algorithm to select the operations for data schedule optimization.

Second, as β increases from 0.29 to 0.73, the workloads (the number of requests) among the service nodes are becoming more balanced. As the workloads become more balanced, we found that the average cost for each request is gradually increased no matter what size of the network is. This is not an expected phenomenon since load balancing is always a goal to pursue for performance optimization. However, in our case we have to pay for it. The rationale behind this phenomenon could lie in the fact that the balanced case would result in more costs due to its even distribution of the requests among the network nodes, compared to its unbalanced counterpart, which would incur more transfer operations, especially when ρ becomes large. Thus, in reality we should strike a balance between the service cost and the performance benefits from the load balancing.

Finally, another interesting observation is the scalability of the algorithm. Fig. 11 shows how the performance of the algorithm with respect to different βs is consistent across the networks with 50 and 200 nodes, respectively. We also conducted experiments for the networks with other number of the nodes, say 100 and 150, and made a similar observation that the average cost of the requests are relatively stable and independent of the network size, which demonstrates the scalability of our online algorithm in terms of the average service cost.

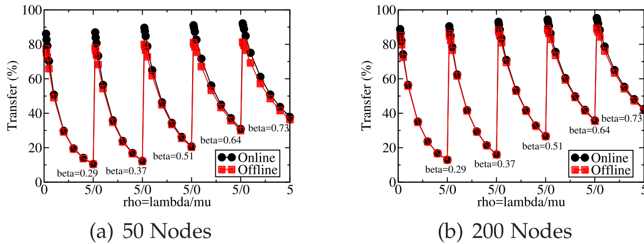
6.2.3 The Online Algorithm

In this experiment, we first evaluate the actual performance of the online algorithm by comparing it with the optimal off-line algorithm, and then implement a random



(a) 50 Nodes (b) 200 Nodes

Fig. 12. The relative performance changes of the online algorithm over the optimal off-line algorithm in different situations.



(a) 50 Nodes (b) 200 Nodes

Fig. 13. Percentages of transfers between the online and off-line algorithms across different βs.

copy-selection strategy, denoted by *Random*, as the baseline to measure the proposed *Balance* and *Priority* strategies in copy selections.

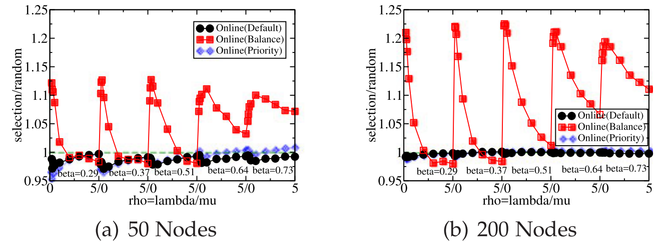
• *Performance Ratios.* With the same configuration of λ and μ as in Fig. 11, we show the cost ratios of the online algorithm to the optimal off-line algorithm based on different network scales in Fig. 12, where β is varied from 0.29 (unbalanced) to 0.73 (balanced).

From Fig. 12, we can make the following observations. First, the actual ratios of the online algorithm to its optimal off-line counterpart in all the studied cases are in range of $[1.3, 2.0]$, which are much better than the theoretical competitive ratio of 3. This demonstrates the effectiveness of the online algorithm for service cost reduction in practice.

Second, the cost ratio curves of the online algorithm for all the β values exhibit a similar trend, which starts from relatively a low value at $\rho = 0.2$, and then quickly increases to maximum around $\rho = 0.5$, after that, gradually decreases to minimum at $\rho = 5$. This reveals that the online algorithm would progressively behave better when $\rho = \lambda/\mu$ is increased over 1. We can draw this conclusion by investigating how both the algorithms perform in different cases as shown in Fig. 13, where the percentages of transfers for both algorithms across different βs are plotted side by side.

Based on Fig. 13, the online algorithm incurs more transfers than the off-line algorithm when $\rho = \lambda/\mu$ is small, and this discrepancy is gradually diminished as ρ is increased. This is easy to understand. When λ is small, compared to μ, the anticipatory period of AC is relatively short, reducing its effects in servicing more incoming requests, and thus, incurring more caching and transfer operations, which in turn increases the overall costs. These costs are gradually reduced as ρ becomes large, where the large values of λ limit the uses of transfers in the online algorithm (Fig. 13), as such, enabling the algorithm to exploit more cheaper caching operations and thereby exhibiting better performance.

Finally, the relative performance of the online algorithm becomes worse as the β is varied from 0.29 to 0.73, which demonstrates again the online algorithm, as with its off-line counterpart, is unfriendly to the load balancing. We can reason about this phenomenon by following the same arguments in the off-line case. By comparing Figs. 12a and 12b, we can further reveal that the online algorithm, as with its off-line counterpart again, is scalable to the network scales. We attribute this result to the design of the algorithm as well as the homogeneity of the cost model.



(a) 50 Nodes (b) 200 Nodes

Fig. 14. Changes of the relative performance of the online algorithm with different copy selections across different βs.

• *Copy Selections.* In this experiment, we measure the online algorithm that adopts different strategies to select a live copy for the next incoming request. To this end, we develop a random strategy, denoted by *Random*, as a baseline, which keeps track of all the live copies, and randomly selects one in a uniform way to serve the next request.

Fig. 14 illustrates how the performances of the online algorithm with different copy selections are changed relatively to *Random*. Overall, for both network scales (50 nodes and 200 nodes), *Default* and *Priority* are slightly better or competitive with *Random*, while *Balance* exhibits much worse performance than *Random*. *Default* prefers to extend the livespan of the most recent copy in the last serving node. As a result, it reduces the chances of other live copies to extend their lifespan by serving the incoming requests, which in turn minimizes the total cost. Similar to *Default*, *Priority* is in favor of the node (also the copy) with intense incoming requests, and thus it has similar performance merits with *Default*. In contrast, *Balance* adopts a balance strategy that tries to extend the lifespan of all the live copies, which could result in high caching cost, especially when ρ is small and network is large as in both cases either caching is costly or more live copies are available.

7 Conclusions

In this paper, we studied a new caching model problem for cost-effective accesses to current data services, which is characterized by exploiting the monetary cost and the trace of access trajectory information to derive the cache replacements. With homogeneous cost model, we first leveraged dynamic programming techniques to propose a fast optimal algorithm that can serve an off-line request sequence within $O(mn)$ time-space. Then, we introduced an idea of *anticipatory caching* to present an online algorithm with competitive ratio of 3. We provably achieve these results by making several key observations on how a sequence of requests is served in both online and off-line cases, and thereby conducting a strict analysis on the schedules in both algorithms. To validate our findings, we also developed some efficient data structures to implement the algorithms, and conducted simulation-based studies. Our results showed that the proposed algorithms are not only practical to the data caching

problem, but also appear to be of theoretical significance to a natural new paradigm in the realm of online algorithms.

Acknowledgments

The authors are deeply indebted to Dong Huang for implementing the ideas described here in a simulated systems. This work was supported in part by National Key R&D Program of China (No. 2018YFB1004804), National Science Foundation of China under grant No. 61672513, Shenzhen Oversea High-Caliber Personnel Innovation Funds (KQCX20170331161854), National Science Foundation of China under Grant No. 61572377, and Shenzhen Basic Research Program (JCYJ20170818163026031, JCYJ20170818153016513).

References

[1] P. Cao and S. Irani, "Cost-aware www proxy caching algorithms," in *Proc. USENIX Symp. Internet Technol. Syst. USENIX Symp. Internet Technol. Syst.*, 1997, pp. 18–18.
[2] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi, "Web caching with consistent hashing," in *Proc. 8th Int. Conf. World Wide Web*, 1999, pp. 1203–1213.
[3] X. Fan, J. Cao, H. Mao, W. Wu, Y. Zhao, and C. Xu, "Web access patterns enhancing data access performance of cooperative caching in IMANETs," in *Proc. 17th IEEE Int. Conf. Mobile Data Manage.*, Jun. 2016, pp. 50–59.
[4] G. Chockler, G. Laden, and Y. Vigfusson, "Data caching as a cloud service," in *Proc. 4th Int. Workshop Large Scale Distrib. Syst. Middleware*, 2010, pp. 18–21.
[5] G. Chockler, G. Laden, and Y. Vigfusson, "Design and implementation of caching services in the cloud," *IBM J. Res. Develop.*, vol. 55, no. 6, pp. 9:1–9:11, Nov. 2011.
[6] N. L. Scouarnec, C. Neumann, and G. Straub, "Cache policies for cloud-based systems: To keep or not to keep," in *Proc. IEEE 7th Int. Conf. Cloud Comput.*, Jun. 2014, pp. 1–8.
[7] I. Stefanovici, E. Thereska, G. O'shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey, "Software-defined caching: Managing caches in multi-tenant data centers," in *Proc. 6th ACM Symp. Cloud Comput.*, 2015, pp. 174–181.
[8] S. Jiang and X. Zhang, "LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst.*, 2002, pp. 31–42.
[9] S. Podlipnig and L. Böszörmenyi, "A survey of web cache replacement strategies," *ACM Comput. Surveys*, vol. 35, no. 4, pp. 374–398, 2003.
[10] M. Chaudhuri, "Pseudo-LIFO: The foundation of a new family of replacement policies for last-level caches," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2009, pp. 401–412.
[11] D. A. Jiménez, "Insertion and promotion for tree-based PseudoLRU last-level caches," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2013, pp. 284–296.
[12] L. Etienne, T. Devogele, and A. Bouju, "Spatio-temporal trajectory analysis of mobile objects following the same itinerary," in *Proc. Joint Int. Conf. Theory Data Handling Model. GeoSpatial Inf. Sci.*, May 2010, pp. pp 86–91.
[13] 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.
[14] 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.
[15] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, Jun. 1966.
[16] D. Sleator and R. Tarjan, "Amortized efficiency of list update and paging rules," *Commun. ACM*, vol. 28, pp. 202–208, 1985.
[17] 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. 2003.
[18] K. Candan, B. Prabhakaran, and V. Subrahmanian, "Collaborative multimedia documents: Authoring and presentation," Univ. Maryland, College Park, MD, USA, Tech. Rep. CS-TR-3596, UMIACS-TR-96–9, Jan. 1996.
[19] B. Veeravalli and E. Yew, "Network caching strategies for reservation-based multimedia services on high-speed networks," *Data Knowl. Eng.*, vol. 41, no. 1, pp. 85–103, Apr. 2002.
[20] A. Benoit, V. Rehn-Sonigo, and Y. Robert, "Replica placement and access policies in tree network," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 12, pp. 1614–1627, Dec. 2008.
[21] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proc. 29th Annu. ACM Symp. Theory Comput.*, 1997, pp. 654–663.
[22] M. Charikar, D. Halperin, and R. Motwani, "The dynamic servers problem," in *Proc. 9th Annu. ACM-SIAM Symp. Discrete Algorithms*, 1998, pp. 410–419.
[23] D. Halperin, J. Latombe, and R. Motwani, "Dynamic maintenance of kinematic structures," Stanford, CA, USA, 1995.
[24] Y. Mansouri, A. N. Toosi, and R. Buyya, "Cost optimization for dynamic replication and migration of data in cloud data centers," *IEEE Trans. Cloud Comput.*, 2017.
[25] M. Kallahalla and P. J. Varman, "PC-OPT: Optimal offline prefetching and caching for parallel I/O systems," *IEEE Trans. Comput.*, vol. 51, no. 11, pp. 1333–1344, Nov. 2002.
[26] B. S. Gill, "On multi-level exclusive caching: Offline optimality and why promotions are better than demotions," in *Proc. 6th USENIX Conf. File Storage Technol.*, 2008, pp. 4:1–4:17.
[27] A. Jain and C. Lin, "Back to the future: Leveraging Belady's algorithm for improved cache replacement," in *Proc. 43th Int. Symp. Comput. Archit.*, 2016, pp. 78–89.
[28] W. Shi and C. Su, "The rectilinear Steiner arborescence problem is NP-complete," in *Proc. 11th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2000, pp. 780–787.
[29] L. Breslau, P. Cue, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *Proc. IEEE INFOCOM*, 1999, pp. 126–134.
[30] S. Wong, Y. Yuan, and S. Lu, "Characterizing flows in large wireless data networks," in *Proc. ACM Annu. Int. Conf. Mobile Comput. Netw.*, 2004, pp. 174–186.

Yang Wang received the BSc degree in applied mathematics from the Ocean University of China, in 1989, the MSc degree in computer science from Carleton University, in 2001, and the PhD degree in computer science from the University of Alberta, Canada, in 2008. He is currently with Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, as a professor. His research interests include cloud computing, file and storage systems, and Java virtual machine on multicores.

Shuibing He received the PhD degree in computer science and technology from the Huazhong University of Science and Technology, China, in 2009. He is now an associate professor with the Computer School of Wuhan University, China. His current research areas include parallel I/O system, file and storage system, high-performance computing, and distributed computing.

Xiaopeng Fan received the BE and ME degrees in computer science from Xidian University, in 2001 and 2004, respectively, and the PhD degree in computer science from Hong Kong Polytechnic University, in 2010. He is currently an associate professor with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. His research interests include big data analytics, mobile cloud computing, and software engineering. His recent research has focused on big data analytics in urban computing. He has published more than 30 papers in conferences and journals.

Chengzhong Xu received the PhD degree from the University of Hong Kong, in 1993. He is currently a tenured professor of Wayne State University and the director of the Institute of Advanced Computing and Data Engineering, Shenzhen Institutes of Advanced Technology of Chinese Academy of Sciences. His research interests include parallel and distributed systems and cloud computing. He has published more than 200 papers in journals and conferences. He serves on a number of journal editorial boards, including the *IEEE Transactions on Computers*, the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Cloud Computing*, the *Journal of Parallel and Distributed Computing* and the *China Science Information Sciences*. He is a fellow of the IEEE.

Xian-He Sun received the BS degree in mathematics from Beijing Normal University, China, in 1982, and the MS and PhD degrees in computer science from Michigan State University, in 1987 and 1990, respectively. He is a distinguished professor with the Department of Computer Science, Illinois Institute of Technology (IIT), Chicago. He is the director of the Scalable Computing Software Laboratory, IIT, and is a guest faculty with the Mathematics and Computer Science Division, Argonne National Laboratory. His research interests include parallel and distributed processing, memory and I/O systems, software systems, and performance evaluation and optimization. He is a fellow of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.