Prefetching and Caching for Minimizing Service Costs: Optimal and Approximation Strategies

Guocong Quan, Atilla Eryilmaz, Jian Tan, Ness Shroff

The Ohio State University

Abstract

Strategically prefetching data has been utilized in practice to improve caching performance. Apart from caching data items upon requests, they can be prefetched into the cache before requests actually occur. The caching and prefetching operations compete for the limited cache space, whose size is typically much smaller than the number of data items. A key question is how to design an optimal prefetching and caching policy, assuming that the future requests can be predicted to certain extend. This question is non-trivial even under an idealized assumption that the future requests are precisely known.

To investigate this problem, we propose a cost-based service model. The objective is to find the optimal offline prefetching and caching policy that minimizes the accumulated cost for a given request sequence. By casting it as a min-cost flow problem, we are able to find the optimal policy for a data trace of length N in expected time $O(N^{3/2})$ via flow-based algorithms. However, this requires the entire trace for each request and cannot be applied in real time. To this end, we analytically characterize the optimal policy by obtaining an optimal cache eviction mechanism. We derive conditions under which proactive prefetching is a better choice than passive caching. Based on these insights, we propose a lightweight approximation policy that only exploits predictions in the near future. Moreover, the approximation policy can be applied in real time and processes the entire trace in $O(N)$ expected time. We prove that the competitive ratio of the approximation policy is less than $\sqrt{2}$. Extensive simulations verify its near-optimal performance, for both heavy and light-tailed popularity distributions.

1. Introduction

Proactively prefetching data items instead of passively caching them has been utilized in practice to accelerate data access, e.g, for content data networks [1, 2]. This strategy becomes even more appealing given that the advances in learning techniques provide effective tools to predict various data request patterns [3, 4, 5, 6]. For certain applications, the prediction can be reasonably accurate [7, 8]. To design an optimal strategy that combines prefetching and caching demands careful investigation. Proactively prefetching data brings the data into the cache before the actual requests occur. Passively caching data, on the other hand, only fetches the missed data from the backend storage after the requests arrive.

There is a trade-off between prefetching and caching. Due to competing the limited cache space, loading a prefetched data item into the cache typically has to trigger cache evictions, which may potentially introduce more cache misses for future requests. Although great efforts have been put to approximate the short-term and long-term data statistics to prefetch the most popular data [9, 10, 11, 12, 13, 14], a fundamental question remains to be answered: *even with a perfect knowledge of future requests, how to optimally prefetch data items beforehand instead of caching them upon requests?*

[*] This paper was supported by NSF grants CMMI-SMOR-1562065, CNS-ICN-WEN-1719371, CNS-NeTS-1514260, CNS-NeTS-1717045, CNS-NeTS-1717060, CNS-NeTS-2007231, CNS-SpecEES-1824337, ONR Grant N00014-19-1-2621, and DTRA grant HDTRA1-18-1-0050.

Email addresses: quan.72@osu.edu (Guocong Quan), eryilmaz.2@osu.edu (Atilla Eryilmaz), tan.252@osu.edu (Jian Tan), shroff.11@osu.edu (Ness Shroff)

Knowing the entire request sequence defines an offline algorithm, which nevertheless can help an online deployment. Using predictions has successfully made optimal offline policies practical in real applications [4, 5, 6]. Theoretically, the optimal offline policy provides an effective performance bound for online cases [15]. It can also be used to guide the online design. For example, by leveraging machine learning, an optimal offline policy can be used to train an online decision model using history information [16].

Previous studies on optimal offline prefetching and caching policies mainly focus on file systems [17, 18, 19, 20, 21, 22], where prefetching and caching are not clearly distinguished. However, in many other important scenarios, e.g., for CDNs, prefetching and caching have significant differences. First, prefetching and caching can design separate cache update rules. For prefetching, the prefetched data remain in the cache until the future requests arrive. For caching, when a miss occurs, the requested data item is directly fetched from the backend, but whether to put it into the cache or not is up to the cache policy [15]. Second, prefetching incurs lower costs than caching. If fetching a missed data item is scheduled after the arrival of a request, it must be performed urgently to satisfy delay requirements. Prefetching, on the other hand, is performed beforehand and is not time-sensitive. It can be performed at a lower rate and avoid congestions, which allows more flexibility for scheduling. For example, data can be prefetched at off-peak times to reduce costs [23, 24]. Third, prefetching can achieve a lower service delay by loading the data into the cache in advance. Therefore, the existing analysis on prefetching and caching for file systems does not directly apply in the above-mentioned scenarios.

To address this issue, we propose a cost-based service model to jointly optimize prefetching and caching, by assuming that prefetching incurs a lower cost than caching, due to less I/O consumption, more flexibility in scheduling and lower service delays. If the requested data is not cached, it can be served by fetching the data from the backend data storage after the request arrives, by paying a fetching cost. And the fetched data can be either loaded into the cache, or discarded to save space. Based on the predictions, we can prefetch the data items before they are requested. The prefetched data items need to remain in the cache until the requests arrive. Otherwise, they should not be prefetched at the first place, assuming that we know the future requests. With the goal to minimize the accumulated cost, we decide whether to cache a missed data item when a request occurs or prefetch it before the request arrives. We propose flow-based algorithms to find the optimal offline policy, as well as a lightweight "look-ahead" approximation policy that only knows the request information in the near future. These new designs not only reveal the fundamental trade-off between prefetching and caching, but also provide useful insights to improve real applications.

Our contributions are summarized as follows.

- We propose a cost-based caching model where different costs will be incurred depending on whether a missed data item is prefetched or fetched. With the objective to understand the fundamental trade-off between prefetching and caching, we investigate the optimal offline policy that minimizes the accumulated cost (see Section 2).
- We reformulate the optimal prefetching and caching problem as a min-cost flow problem. For a given request sequence of length N, the optimal policy can be obtained by flow-based algorithms in $O(N^{3/2})$ expected time (see Section 4).
- We analytically characterize the optimal policy by providing sufficient conditions under which prefetching the missed data is the optimal choice (see Section 5). Moreover, we prove that consistently prefetching is not always optimal, with a competitive ratio as high as 2, depending on the future requests and the prefetching cost (see Section 6).
- We propose a lightweight "look-ahead" approximation policy based on the insights revealed by the characteristics of the optimal policy. The approximation policy can be executed in real time and processes the entire trace in $O(N)$ expected time. Performance guarantees are provided by deriving the competitive ratio (see Section 6).
- We conduct extensive experiments using real CDN traces and synthetic data requests that are generated from both heavy and light-tailed popularity distributions. The approximation policy always achieves near-optimal average performance (see Section 8).

Related Works: Caching algorithms have been extensively studied. It is known that Belady's algorithm [25] is an optimal offline eviction policy that minimizes the number of misses, assuming that the data items have identical sizes.

Specifically, it evicts the data item that is requested farthest in the future. When the data sizes are not identical, Belady's algorithm is no longer optimal, and finding the optimal offline policy is NP-hard. A few approximation policies have been proposed with different complexities and performance bounds [26, 27, 28, 20]. One recent work [15] provides an asymptotically optimal solution and practical approximation algorithms with tight performance bounds for real traces. It leverages a flow-based representation, and shows that the optimal offline caching policy can be obtained by solving a min-cost flow problem. These offline policies have successfully guided the design of online algorithms [16].

Prefetching strategies, together with caching algorithms, have been widely explored in real applications, including processor architectures [29, 30], file systems [31, 32] and networks [33, 34, 35, 24, 36]. The offline optimal strategies have been studied for disk systems with an objective to minimize the stall time [17, 18, 19, 20, 21, 22]. It is shown in [37] that the optimal offline solution can be found in polynomial time for single-disk systems. This problem is reformulated as a min-cost multi-commodity flow problem in [21]. For disk systems, the existing work does not distinguish the costs caused by caching and prefetching, which makes proactive prefetching almost always a better choice than passive caching. In this paper, we consider the scenarios where prefetching and caching can have different costs. Interestingly, we show that consistently prefetching is not always optimal. The new insights can be used to further improve the design of prefetching and caching mechanisms.

2. Problem Formulation

Consider a set of data items $\mathcal{D} = \{d_i : 1 \leq i \leq M\}$ of unit sizes, and a sequence of data requests that arrive at the time points $\{\tau_n, 1 \leq n \leq N\}$. Let R_n ($R_n \in \mathcal{D}$) denote the data item that is requested at time τ_n and $\{R_n\}_{n=1}^N$ denote the entire request sequence. We assume that $\{R_n\}_{n=1}^N$ is known.

If the requested data item is already in the cache, then the request can be served without paying a cost. However, if it is not cached, we have two options to serve the corresponding data request at different costs. The first option is to fetch the data from the backend after the request arrives, paying a fetching cost 1. We can decide whether to load the fetched item into the cache or not. The second option is to prefetch the data item before it is requested, paying a prefetching cost c, $0 \leq c \leq 1$. Note that the prefetched data item has to be loaded into the cache. If the cache space is full, other items must be evicted before storing a new one. For ease of analysis, we first assume a best-case scenario where the prefetched data item is loaded into the cache right before it is requested. In Section 7, we will show that this assumption could be waived to some extent.

We observe that the optimal offline policy shall satisfy the following two properties.

Property 1 (Interval caching decisions): As illustrated in [15], the optimal offline policy will not evict a cached data item d_i between two requests for d_i. Consider an example where d_i is initially cached and requested at τ_1 and τ_5. It is suboptimal to evict d_i at some time (e.g., τ_3) between τ_1 and τ_5, because storing d_i in $(\tau_1, \tau_3]$ does not serve any requests and is a waste of caching resource. A wiser decision will be evicting d_i right after serving R_1 or caching it at least until R_5 arrives. Therefore, we focus on caching policies that only evict a cached data item immediately after serving a request for it.

Property 2 (Fetching without caching): When the optimal policy fetches a missed request, it must not load the fetched data into the cache and trigger evictions, because otherwise prefetching will be a better option. Therefore, it is sufficient to consider the policies such that data items will only be loaded into the cache by prefetching.

In the rest of the paper, we restrict the design space by only considering the policies that satisfy these two properties, and call them feasible policies. An optimal offline policy is guaranteed to exist in this design space.

Let $l_n = \max\{i < n : R_i = R_n\}$, which indicates that τ_{l_n} is the most recent time when R_n is requested. If R_n is the first request for that item, then set $l_n = 0$. Formally, we define three decision variables for each request R_n, $1 \leq n \leq N$:

$$\begin{aligned}
x_n &\triangleq \mathbf{1}(\text{Store } R_n \text{ in the cache during } (\tau_{l_n}, \tau_n]), \\
f_n &\triangleq \mathbf{1}(\text{Fetch } R_n), \\
p_n &\triangleq \mathbf{1}(\text{Prefetch } R_n),
\end{aligned}$$

where $\mathbf{1}(\mathcal{E})$ is an indicator function that takes value 1 if the event $\mathcal{E}$ occurs, and 0 otherwise. The optimal offline policy for a cache of size b is the solution of the following optimization problem.

$$\begin{aligned} \min \quad & \sum_{1\le n\le N} (1-x_n)(f_n + c\cdot p_n) && && (1)\\ \text{subject to} \quad & p_n + \sum_{i:l_i<n\le i} x_i \le b && \text{for } \forall n && (2)\\ & x_n + f_n + p_n \ge 1 && \text{for } \forall n && (3)\\ & x_n, f_n, p_n \in \{0,1\} && \text{for } \forall n && (4) \end{aligned}$$

If $x_n = 0$, i.e., R_n is not stored in cache at τ_n, then a cost $f_n + cp_n$ will be induced depending on whether R_n is fetched or prefetched as shown in the objective function (1). The cache capacity constraint is described by (2). If R_n is prefetched (i.e., $p_n = 1$), then the cache should have an available space to accommodate the prefetched data at τ_n. Furthermore, Constraints (3) and (4) guarantee that the request R_n must be either directly served from the cache or prefetched/fetched from the backend data storage.

Solving the optimal solution is equivalent to answering the following two questions:
Q1: *If a request is not cached, should we prefetch it beforehand or fetch it upon the request?*
Q2: *If a data item is prefetched and the cache is full, which item should be evicted?*
We will analytically answer these two questions in Section 5. And here are the short answers:
A1: *Prefetching the requested data is a better choice, if*

- *there exist requests for popular items in the near future, and the popular items are not cached currently, or*
- *the prefetching cost is sufficiently low.*

A2: *The farthest-in-future data item should be evicted, if we choose to prefetch an item and the cache is full.*

3. Motivating Example

In this Section, we will introduce a motivating example to show that 1) prefetching is not always beneficial; 2) the optimal policy is non-trivial and depends on the prefetching cost.

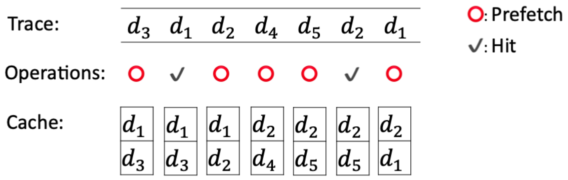
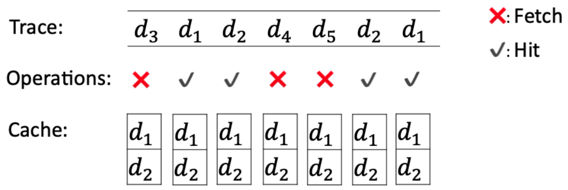

Trace:
d_3 d_1 d_2 d_4 d_5 d_2 d_1
Operations:
Cache:
d_1 d_1 d_1 d_2 d_2 d_2 d_2
d_3 d_3 d_2 d_4 d_5 d_5 d_1
○: Prefetch
✓: Hit

Figure 1: Always prefetching before the request.

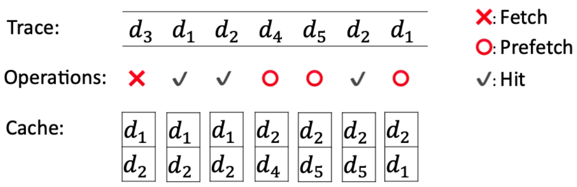
Example 1. *Consider a cache of size 2. Assume that d_1, d_2 are initially stored in the cache. For a given sequence of requests $d_3, d_1, d_2, d_4, d_5, d_2, d_1$, we apply three different strategies and compare their costs.*
Strategy 1 *(Always prefetching): We always prefetch the requested data that are not cached, and evict the cached item that is requested farthest in future. The operations as well as the cache content after serving each request are shown in Figure 1. Strategy 1 prefetches requests R_1, R_3, R_4, R_5, R_7 and evicts d_2, d_3, d_1, d_4, d_5, respectively. The total cost for the given sequence is* $5c$.
Strategy 2 *(Always fetching): We always fetch the missed data after the request arrives, and make caching decisions based on Belady's algorithm which is optimal for caching without prefetching [38]. For this specific example, Belady's algorithm will never update the cache content. As shown in Figure 2, a total cost 3 will be incurred for the given sequence.*
Strategy 3 *(Combination of fetching and prefetching): As shown in Figure 3, the first request is not stored in the cache and the requested data d_3 is fetched. Then, R_4, R_5, R_7 are prefetched with d_1, d_4, d_5 being evicted, respectively. A total cost* $3c + 1$ *is incurred by this strategy.*



Trace:	d_3	d_1	d_2	d_4	d_5	d_2	d_1
Operations:	✗	✓	✓	✗	✗	✓	✓
Cache:	d_1 d_2	d_1 d_2	d_1 d_2	d_1 d_2	d_1 d_2	d_1 d_2	d_1 d_2

✗: Fetch
✓: Hit

Figure 2: Always fetching upon the request.



Trace:	d_3	d_1	d_2	d_4	d_5	d_2	d_1
Operations:	✗	✓	✓	○	○	✓	○
Cache:	d_1 d_2	d_1 d_2	d_1 d_2	d_2 d_4	d_2 d_5	d_2 d_5	d_2 d_1

✗: Fetch
○: Prefetch
✓: Hit

Figure 3: Combination of fetching & prefetching.

We have the following two observations from this motivating example.

Observation 1: Prefetching is not always beneficial. If $c > 3/5$, Strategy 1 (always prefetching) will even incur a larger accumulated cost than Strategy 2 (always fetching). If the prefetching cost c is considerably small, then prefetching the data item before the request will be a better choice than fetching it upon the request.

Observation 2: The optimal policy is non-trivial and highly depends on the prefetching costs. Actually, all the three strategies described above are optimal policies for the given trace and some specific c values. Specifically, Strategies 1, 2, 3 are optimal for $c \in (0, 1/2]$, $(2/3, 1]$, $(1/2, 2/3]$, respectively.

Therefore, the optimal prefetching and caching decision depends on the joint effect of future requests and the prefetching cost c. Even for the same given trace, a fixed policy cannot work uniformly well for different c values. When the trace is long, the design space can be considerably large, since the possible combinations of fetching and prefetching will increase exponentially. How to efficiently find the optimal policy is a challenging task.

4. Optimal Policy via Min-Cost Flow

Instead of directly exploring the optimization problem (1), we leverage the underlying structure of prefetching and caching, and reformulate it as a min-cost flow problem which aims to send a certain amount of flow through a flow network at a smallest cost. We will show that the optimal prefetching and caching policy can be constructed from the min-cost flow.

In this paper, we use the flow notations that are shown in Figure 4. A flow network is represented by a directed graph where each edge is associated with a parameter tuple $(capacity, cost)$. The total amount of flow going through an edge must not exceed its $capacity$. And a $cost$ per unit flow will be charged for the flow going through the edge. The node i is associated with a number β_i representing its surplus/demand. If $\beta_i > 0$, then the node is a source node. If $\beta_i < 0$, then the node is a sink node. We will not label β_i in the graph if it is zero.

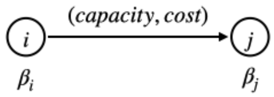


Figure 4: Notations for the flow network.

In [15], the min-cost flow representation is used to solve optimal offline caching without prefetching, where each request is represented by a node. By constructing a proper flow network, the optimal offline caching can be constructed from the min-cost flow solution. However, the flow network constructed in [15] does not support prefetching operations, and therefore cannot be applied to our settings. To this end, we propose a more general min-cost flow representation, which supports both proactive prefetching and passive caching. In Section 4.1, we will show how to construct a flow network for a given sequence of requests. Then, we will prove that the optimal prefetching and caching policy can be obtained by finding the min-cost flow in Section 4.2.

Trace:	d_1	d_2	d_1	d_3

Table 1: Example trace of requests for d_1, d_2 and d_3

4.1. Flow Network Construction

For a given sequence of requests, a corresponding flow network can be constructed, where each request is represented by four or five nodes and the nodes are connected by five types of edges including caching edges, fetching edges, prefetching edges, eviction edges and auxiliary edges. Detailed construction steps are presented as follows. The result of each step is illustrated in Figure 6 for the request sequence shown in Table 1 and a cache of size 2.

Step 1 (Generate nodes): For each request in the trace, if it is the first time to request the data, then generate five nodes where three of them are placed in the first row and two in the second, as shown in Figure 5. To facilitate the description of the following steps, we denote these nodes by n_0, n_1, n_2, n_3 and n_4. If the request is not the first request for the data, then we will only generate n_1, n_2, n_3 and n_4 nodes for that request. Moreover, for each data item, let the n_0 node of its first request be the source node, and the n_2 node of its last request be the sink node. These nodes are added for the example trace in Figure 6a.

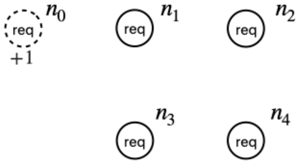


Figure 5: Representative nodes for each request

Step 2 (Add caching edges): As shown in Figure 6a. link the nodes in the second row by edges with *capacity* = b, *cost* = 0, where b is the cache size. These edges are named caching edges. A flow going through these edges represents that the corresponding request is stored in the cache.

Step 3 (Add prefetching edges): For each request, add an edge that is directed from the n_2 node of the last request for the same data and to the n_3 node of the request. If the request is the first request for that data, then add an edge directed from its source node n_0 to the n_3 node. The *capacity* of the edge is 1 and the *cost* is c. The flow going through these edges means that the corresponding requested data is prefetched. We add these prefetching edges in Figure 6b.

Step 4 (Add eviction edges): For each request, add an edge directed from its n_4 node to its n_1 node with *capacity* = 1 and *cost* = 0, as shown in Figure 6c. The flow going through these edges indicates that the corresponding requested data is evicted.

Step 5 (Add fetching edges): For each request, if it is not the last request for that data item, add an edge directed from the n_2 node of the request and to the n_1 node of the next request for the same data item, as shown in Figure 6d. Moreover, if it is the first request for the corresponding data item, add an edge directed from its n_0 to its n_1 node. The parameter tuple $(capacity, cost)$ for these edges is set to be $(1, 1)$. The flow going through these edges indicates that the corresponding request is a miss and the requested data is fetched.

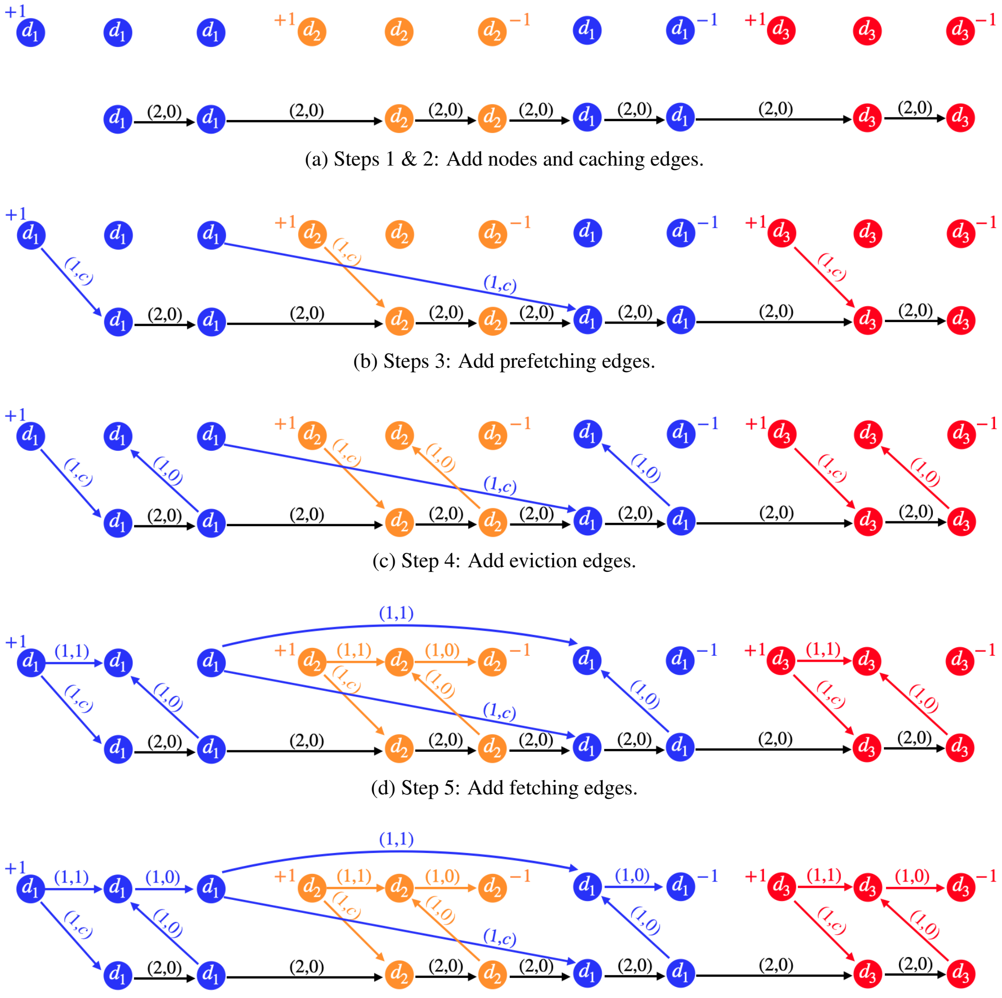
Step 6 (Add auxiliary edges): For each request, add an auxiliary edge directed from its n_1 node to the n_2 node, as shown in Figure 6e. The parameter tuple $(capacity, cost)$ is set to be $(1, 0)$. The capacity of the auxiliary edge guarantees that the amount of flow going through the prefetching and the fetching edges must not exceed 1. The function of these auxiliary edges is to ensure that an integer flow routing solution can correspond to a feasible prefetching and caching policy (see Section 4.2).

According to the proposed six steps, a flow network can be constructed for a given data trace. See Figure 6e for the flow network constructed for the example trace in Table 1. In Section 4.2, we will demonstrate how the constructed flow network can be leveraged to solve the caching problem.

4.2. Optimal Prefetching and Caching Policy

In this section, we will leverage the constructed flow network to find the optimal offline prefetching and caching policy. Specifically, we will show that there is an one-to-one correspondence between feasible prefetching and caching policies and integer flow routing solutions in the flow network.

Theorem 1. *For a given data trace, a feasible prefetching and caching policy corresponds to an integer flow routing solution in the constructed flow network, and vice versa. An optimal prefetching and caching policy corresponds to an integer min-cost flow, and vice versa.*



(a) Steps 1 & 2: Add nodes and caching edges.

(b) Steps 3: Add prefetching edges.

(c) Step 4: Add eviction edges.

(d) Step 5: Add fetching edges.

(e) Step 6: Add auxiliary edges.

Figure 6: Flow network construction.

The proof of Theorem 1 is straightforward. For a feasible prefetching and caching policy, we can construct an integer flow routing solution in the constructed flow network where the amount of flow going through each edge is an integer. Specifically, any cache operation (i.e., fetching, prefetching or eviction) corresponds to an edge (i.e., fetching, prefetching or eviction edge) in the flow network. Based on how the request is served, we can route the flow through the corresponding edges. Similarly, given an integer flow routing solution, a corresponding feasible prefetching and caching policy can be constructed from it.

Furthermore, the cost achieved by a prefetching and caching policy is the same as the cost of its corresponding flow routing solution. Therefore, the optimal prefetching and caching policy can be obtained by finding the integer min-cost flow solution.

Theorem 1 shows that an optimal prefetching and caching policy can be obtained by finding an integer min-cost flow. Formally, we define the flow-based optimal offline policy as follows.

Flow-based optimal offline policy (π_{OPT}): Given a sequence of data requests, solve the integer min-cost flow for the flow network constructed according to the steps introduced in Section 4.1. Then prefetch, fetch or evict the data item if the min-cost flow is routed through the corresponding prefetching, fetching or eviction edges, respectively.

Note that the integer min-cost flow always exists, since the capacities, surpluses and demands are all integers in the flow network [39]. Moreover, the integer min-cost flow can be efficiently solved, if the prefetching cost c is assumed to be a rational number. Given a trace with N requests, there are at most $5N$ nodes and $6N - 1$ edges in the corresponding flow network. For rational prefetching costs, the problem is solvable in $O(N^{3/2})$ expected time [40, 41, 42].

Although the proposed flow-based policy π_{OPT} is offline optimal, the problem is not satisfactorily solved for the following reasons:

- The flow-based algorithm cannot reveal the underlying insights of the optimal decision. It does not provide analytical answers to the two questions proposed in Section 2, i.e., whether to prefetch or fetch the missed data, and which data item to evict.
- The flow-based algorithm requires the knowledge about all future requests to find the optimal policy. Moreover, the policy cannot be executed in real time, since the optimal decision for a request is unavailable unless the process for the entire data trace is completed. In real practice, the request sequence could be too long to make this method practical [15].

These unanswered questions motivate us to

- analytically characterize the properties of the optimal policy and explicitly answer the questions proposed in Section 2 (see Section 5);
- design a lightweight approximation policy that requires only near-future information, can be executed in real time and achieves near-optimal performance (see Section 6).

5. Characteristics of Optimal Policy

In this section, we will analyze the characteristics of the optimal policy and explicitly answer the two questions proposed in Section 2. Notably, the questions will be addressed in reverse order. We will first show that evicting the farthest-in-future item is optimal when prefetching. Then, we will provide sufficient conditions under which prefetching is a better choice than fetching.

5.1. Optimal Eviction Mechanism

The farthest-in-future item is defined as the data item that is stored in the cache and will be reused after the other cached items. It is known that evicting the farthest-in-future item minimizes the number of misses for caching without prefetching [38]. In this section, we will generalize this result and show that evicting the farthest-in-future item is the optimal choice to minimize the accumulated costs for caching with prefetching. We start by proving the following lemma.

Lemma 1. *Assume that a policy π_n evicts the farthest-in-future items for the first n prefetching operations. Then, there exists a policy π_{n+1} that evicts the farthest-in-future items for the first $n+1$ prefetching operations and does not incur a larger cost than π_n.*

Proof. Assume that τ_{m_1} is the first time when π_n prefetches a request (i.e., R_{m_1}) but does not evict the farthest-in-future item. Let d_i be the item that is evicted by π_n when prefetching R_{m_1}. We will construct a policy π_{n+1} that prefetches R_{m_1} and evicts the farthest-in-future item without incurring a larger cost than π_n. The idea is to let π_{n+1} eventually have the same cache content as π_n at some time point without introducing additional costs before that.

Assume that the next request for the farthest-in-future item arrives at τ_{m_2}, $m_2 > m_1$. We may interchangeably use R_{m_2} to denote the farthest-in-future item and the request for it. Next, we will prove the lemma by considering two possible cases.

Case 1: Consider the case where R_{m_2} is evicted by π_n before τ_{m_2}. For each request, let π_{n+1} make the same prefetching/fetching decision as π_n if the data is not cached by π_{n+1}, and evict the same data as π_n if the data is stored in the cache. Then, there will be at most one item that is cached by π_{n+1} but not cached by π_n. When π_n evicts R_{m_2}, which is not cached by π_{n+1} according to the described update rule, let π_{n+1} evict the only item that is not cached by π_n. Then the two policies lead to the same cache content, and no additional cost is introduced by π_{n+1}.

Case 2: Consider the case where π_n does not evict R_{m_2} before τ_{m_2}. Similar to Case 1, let π_{n+1} always make the same decisions as π_n when possible. There is only one item that is cached by π_{n+1} but not cached by π_n before τ_{m_2}. Then, at

τ_{m_2}, let π_{n+1} prefetch R_{m_2} and evict the item that is not cached by π_n. So far, the two policies will have the same cache content. Next, we will show that this prefetching operation by π_{n+1} will not result in a larger accumulated cost. Since R_{m_2} is the farthest-in-future item at τ_{m_1}, there must exist a request for d_i between R_{m_1} and R_{m_2}. Note that d_i should be prefetched or fetched by π_n, but can be directly served by π_{n+1} from the cache. This prefetching/fetching operation by π_n compensates for the prefetching operation performed by π_{n+1} at τ_{m_2}. Therefore, π_{n+1} will not incur a larger cost than π_n. □

Next, we will prove the optimal eviction mechanism by leveraging Lemma 1.

Theorem 2. *There exists an optimal policy that evicts the farthest-in-future item for all prefetching operations.*

Proof. Let π^* be an optimal prefetching and caching policy. Assume that π^* evicts the farthest-in-future items for the first n prefetch operations. Applying Lemma 1, we can construct a new policy which evicts the farthest-in-future items for the first $n + 1$ prefetching operations without increasing the accumulated cost. Using an induction argument, we can conclude that there must exists an optimal policy that evicts the farthest-in-future item for all prefetching operations. □

Theorem 2 shows that evicting the farthest-in-future item is optimal when a data item is prefetched and the cache is full, which provides an explicit answer to the question Q2 proposed in Section 2. In the rest of the paper, we always follow the farthest-in-future eviction principle unless other specific mechanisms are stated.

5.2. Optimal Conditions for Prefetching

In this section, we will analytically answer the question: whether we should prefetch or fetch an item if it is not stored in cache, assuming that the cache content is updated according to the farthest-in-future principle. In particular, we will provide sufficient conditions, under which prefetching is the optimal choice.

Let $\mathcal{S}_n$ denote the set of cached data items before serving R_n. If $R_n \notin \mathcal{S}_n$, then R_n should be prefetched or fetched. Without loss of generality, assume that the cache is initially full and let $\mathcal{S}_1$ denote the initial cache content. It suffices to analyze whether R_1 should be prefetched or fetched given that $R_1 \notin \mathcal{S}_1$. Define

$$\sigma = \max\{n > 1 : R_n \in \mathcal{S}_1 \text{ and } R_n \text{ is not requested in } [\tau_1, \tau_n)\}.$$

R_σ is the farthest-in-future item, and τ_σ is the first time when R_σ is requested after τ_1. Define

$$\omega = \min\{n > 1 : R_n \in \mathcal{S}_1 \text{ and } R_n \text{ is not requested in } (\tau_n, \tau_\sigma)\}.$$

Note that R_σ is always the farthest-in-future item in the time interval $[\tau_1, \tau_\omega]$, if no prefetching operation is performed.

We start by proving the following lemma.

Lemma 2. *Assume $c < 1$. For any optimal policy π^*, if π^* decides to fetch R_1, then it must also fetch all requests R_n, $1 \leq n \leq \omega$, such that $R_n \notin \mathcal{S}_1$.*

Proof. Suppose for the sake of contradiction that there exists an optimal policy π^* which fetches R_1 and prefetches R_n, $1 \leq n \leq \omega$. Assume without loss of generality that R_n is the first prefetched item after R_1. Therefore, we have $\mathcal{S}_1 = \mathcal{S}_i$ for all $1 \leq i \leq n$. Moreover, the farthest-in-future eviction principle yields $\mathcal{S}_{n+1} = \mathcal{S}_n \cup \{R_n\} \backslash \{R_\sigma\}$. To introduce a contradiction, we will design a new policy π° which achieves the same cache state $\mathcal{S}_{n+1}$ as π^*, but incurs a lower cost.

For the same request sequence and initial cache state $\mathcal{S}_1$, let π° prefetches R_1 and evicts R_σ at τ_1. For the requests arrive in the interval (τ_1, τ_n), let π° make the same caching decision as π^*. We have $\mathcal{S}_n = \mathcal{S}_1 \cup \{R_1\} \backslash \{R_\sigma\}$, since there is no prefetching operations by π° in (τ_1, τ_n). Then, let π° prefetch R_n and evict R_1. Consequently, it achieves the same cache state $\mathcal{S}_{n+1}$ as π^*. Notably, the caching decisions of π^* and π° in $[\tau_1, \tau_n]$ are all identical, except that π^* fetches R_1 but π° prefetches R_1. As a result, the cost of π° is lower than the cost of π^* since $c < 1$, which contradicts to our assumption that π^* is the optimal policy. Therefore, we prove the lemma. □

Leveraging the property of the optimal policy illustrated in Lemma 2, we will show that prefetching before the request is the optimal choice under some insightful conditions.

Theorem 3. *Assume that the upcoming request R_1 is not stored in the cache. Prefetching R_1 is the optimal choice, if any of the following two conditions is satisfied:*

- *C1: There is a request R_n, $1 \leq n \leq \omega$, such that $R_n \notin \mathcal{S}_1$ and R_n is requested at least twice in the time interval $[\tau_1, \tau_\sigma]$;*

- *C2: The prefetching cost c satisfies $c \leq L/(L+1)$, where $L = \sum_{n=1}^{\omega} \mathbf{1}(R_n \notin \mathcal{S}_1)$ is the number of requests that arrive in the time interval $[\tau_1, \tau_\omega]$ but do not belong to $\mathcal{S}_1$.*

Proof. Assuming that the upcoming request R_1 is a miss, we will prove that under any of the proposed conditions, prefetching R_1 is the optimal choice to minimize the accumulated cost.

First, we will show that under Condition $C1$, prefetching R_1 is a better choice than fetching. Assume that R_1 is requested at least twice in the time interval $[\tau_1, \tau_\omega]$. Suppose for the sake of contradiction that the optimal policy π^* fetches R_1. Next, we will construct a new policy π° that prefetches R_1 and incurs a lower accumulated cost than π^*.

Consider the case where π^* evicts R_σ when prefetching some missed request R_k for $1 < k < \sigma$. Then let π° prefetches R_1 and evict R_σ, and then prefetch R_k and evict R_1. Furthermore, let π° perform the same operation as π^* for other requests. Then, π° will reduce the cost of π^* by $1 - c$.

Then, consider the case where π^* keeps R_σ in the cache until τ_σ. Let $R_{n_1} = R_1$ for some $n_1 \in (1, \sigma)$. If R_1 is fetched by π^*, then let π° prefetches R_1 and evict R_σ, and then prefetch R_sigma and evict R_1. For other requests, let π° perform the same operation as π^*. Note that R_{n_1} will be a hit for π°, and therefore, π° reduces the accumulated cost of π^* by $2 - 2c$. Instead, if R_{n_1} is prefetched by π^* and meanwhile some cached data d_i is evicted, then the next request for d_i must arrive after τ_σ due to the farthest-in-future eviction principle. Let π° prefetches R_1 and evict R_σ, and then prefetch R_σ and evict d_i. For other requests, let π° make the same decisions as π^*. Then, π° reduces the cost of π^* by $1 - c$. Therefore, we prove that prefetching is the optimal choice if R_1 is requested twice before τ_σ.

Using a similar argument, we can prove that if there is a request R_n, $1 < n < \omega$, such that Condition C1 holds, then prefetching R_n is the optimal choice. Moreover, since $n < \omega$, we can conclude that prefetching R_1 is also the optimal by applying Lemma 2.

Next, we will show that if Condition C2 holds, then prefetching R_1 is optimal. Let R_{n_i}, $1 \leq i \leq L$, $1 = n_1 < n_2 < \cdots < n_L < \omega$, denote the L requests that are not in the set $\mathcal{S}_1$. Suppose for the sake of contradiction that the optimal policy π^* fetches R_1. Then, Lemma 2 indicates that π^* must also fetch all the L missed data items before τ_ω. Let π° prefetch R_1 and evict R_σ, prefetch $R_{n_{i+1}}$ and evict R_{n_i} for $1 \leq i \leq L-1$, and then prefetch R_σ and evict R_{n_L}. For these requests, π° pays a total cost $(L+1)c$ and π^* pay a total cost L. For other requests, let π° make the same decisions as π^*. Since $c \leq L/(L+1)$, π° induces a lower cost than π^*. Therefore, we prove that prefetching R_1 is the optimal choice if Condition $C2$ holds. □

Theorem 3 provides sufficient conditions under which prefetching is the optimal choice. And these conditions reveal the following useful insights that can be leveraged to guide practical designs:

- Prefetching the upcoming request is optimal, if there exist popular items that will be requested in the near future and are not stored in the cache currently. Note that the upcoming request may not necessarily be popular. The reason is that the popular data will be prefetched and trigger evictions. Thus, evicting these items earlier can be even more beneficial, because more prefetching opportunities (e.g., prefetching the upcoming request) will be provided. This insight is characterized by Condition C1.

- Prefetching is optimal if the prefetching cost c is sufficiently low. This insight is straightforward. Condition C2 characterizes the critical value of c to make prefetching a better choice than fetching. Note that, since $1/2 \leq L/(L+1)$ for $L \geq 1$, prefetching is always optimal for any data traces if $c \leq 1/2$.

Note that the proposed conditions only depend on the current cache content $\mathcal{S}_1$ and the request information between R_1 and R_σ. No information after τ_σ or before τ_1 is required. In Section 6, we will leverage this nice property to propose an approximation policy that only requires near-future information. However, if neither C1 nor C2 is satisfied, the optimal decision will depend on the future requests after R_σ.

6. Approximation Using Near-Future Information

In this section, we propose an approximation policy using near-future information, and show that it is close to optimal by deriving the competitive ratio.

6.1. Lightweight Approximation Policy

Based on the analytical results obtained in Section 5, we propose an approximation policy as follows.
Approximation Policy (π_A): Prefetch the missed request and evict the farthest-in-future item, if $c \leq \sqrt{2}/2$, or any of the conditions $C1$ and $C2$ introduced in Theorem 3 is satisfied. Otherwise, fetch the missed item but do not store it into the cache.

Applying Theorem 3, we know that, for $c \in [0, 1/2] \cup [\sqrt{2}/2, 1]$, π_A prefetches a data item only when prefetching is the optimal choice. The threshold $\sqrt{2}/2$ is chosen to achieve a better competitive ratio, as shown in Section 6.2.

Notably, the proposed approximation policy makes caching and prefetching decisions merely based on the request information before τ_σ. If the data requests are generated independently from a popularity distribution (e.g., Zipf's distribution as observed in real practice [43]), then τ_σ is independent of the trace length N. Although τ_σ can depend on the cache size b, considering the fact that the trace length is typically far larger than the cache size in real practice, N is the dominant term in the time complexity and the impact of b is negligible. Therefore, π_A has an expected time complexity $O(1)$ to make decisions for a single request and $O(N)$ to process the entire trace. In Section 8, we verify through simulations that the required information for π_A to make decisions for a single request does not scale with the trace length. In summary, unlike the flow-based algorithm that need all future requests to make optimal decisions, π_A is lightweight and practical since

- it only exploits near-future information to make decisions;
- it does not have to process the entire trace to find the optimal decision for a single request and can be executed in real time.

6.2. Competitive Ratio Analysis

In this section, we will show that the proposed approximation policy achieves near-optimal performance by characterizing its competitive ratio.

We introduce two additional policies (i.e., always prefetching policy and always fetching policy) as benchmarks.
Always Prefetching Policy (π_P): Always prefetch the missed item. If the cache is full, evict the farthest-in-future item.
Always Fetching Policy (π_F): Always fetch the missed item. If the next request for the fetched item arrives before the farthest-in-future item, then evict the farthest-in-future item and store the fetched one in cache. Otherwise, do not load the fetched item into the cache.

π_P and π_F represent two extreme policies that always prefetch or fetch the missed data. To emphasize different consequences of fetching and prefetching, we eliminate the impact of eviction by applying the optimal eviction to both π_F and π_P. Notably, π_F is also known as the Belady's algorithm [38, 44], which is the optimal caching policy when prefetching is disabled and data sizes are all identical. We will use π_F and π_P as benchmarks to show that a wise combination of prefetching and fetching (e.g., policy π_A) can yield better performance. Specifically, we will prove that π_A always achieves the smallest competitive ratio among the three policies.

Let $cost(\pi, \{R_n\}_{n=1}^N)$ denote the accumulated cost achieved by a policy π for a given trace $\{R_n\}_{n=1}^N$. The competitive ratio for a given prefetching policy π is defined as

$$r_\pi = \sup_{\{R_n\}_{n=1}^N} \frac{\text{cost}\left(\pi, \{R_n\}_{n=1}^N\right)}{\text{cost}\left(\pi^*, \{R_n\}_{n=1}^N\right)},$$

where π^* represents the optimal policy. The competitive ratio evaluates the worst-case performance of a policy. From the definition, we have $r_\pi \geq 1$ for any policy π. Let r_A, r_P, r_F denote the competitive ratio of the proposed approximation policy, always prefetching policy and always fetching policy.

Theorem 4. *Given the prefetching cost c, the competitive ratios of π_F, π_P and π_A can be computed as*

$$r_F = 1/c \quad \textit{for } c \in (0, 1],$$

$$r_P = \begin{cases} 1 & \textit{for } c \in (0, 1/2], \\ 2c & \textit{for } c \in (1/2, 1], \end{cases}$$

$$r_A = \begin{cases} 1 & \textit{for } c \in [0, 1/2], \\ 2c & \textit{for } c \in (1/2, \sqrt{2}/2]. \end{cases}$$

In addition, for $c \in (\sqrt{2}/2, 1]$, r_A can be bounded as

$$\frac{b}{b+1} \cdot \frac{1}{c} \leq r_A \leq \frac{1}{c},$$

where b is the cache size.

Proof. For compactness, we omit the term $\{R_n\}_{n=1}^N$ in the cost expression $cost(\pi, \{R_n\}_{n=1}^N)$. The competitive ratio results for π_F, π_P and π_A are proven as follows.

Competitive ratio for π_F:

For the π_F policy, we will first show that its competitive ratio is upper bounded by $1/c$. Consider a new setting where fetching cost is the same as the prefetching cost c, $0 \leq c \leq 1$. Let π° and $cost^\circ$ denote the optimal policy and the minimum cost under the new setting. Then, for any request sequence, we have $cost^\circ \leq cost(\pi^*)$. Note that there is no additional benefits to prefetch in the new scenario. Therefore, there must exist a π° that always makes the same decision as π_F for the same request sequence. As a result, we have, for any request sequence

$$\frac{cost(\pi_F)}{cost(\pi^*)} \leq \frac{cost(\pi_F)}{cost^o} = \frac{1}{c}.$$

Next, we will show that for any $\epsilon > 0$, π_F can achieve a competitive ratio larger than $1/c - \epsilon$. For a given cache size b, assume without loss of generality that the set of items $\{d_i : 1 \leq i \leq b\}$ is initially stored in the cache. Consider the request sequence $d_{b+1}, d_{b+2}, \cdots, d_{b+k}, d_1, d_2, \cdots, d_b$. π_F will choose to fetch d_i, for $b+1 \leq i \leq b+k$ and achieves a cost k. In contrast, if we choose to prefetch every missed item and evict the farthest-in-future item, a cost $(k+1)c$ will be introduced. Thus, we have

$$r_F \geq \frac{k}{(k+1)c}.$$

By choosing the $k \geq 1/(\epsilon c)$, a lower bound $1/c - \epsilon$ can be achieved for $\forall \epsilon > 0$. Combining the upper and lower bounds, we prove the tight competitive ratio for π_F.

Competitive ratio for π_P:

For $c \in [0, 1/2]$, we will show that π_P is the optimal policy. Assume that R_1 is a miss. A cost 1 will be induced if R_1 is fetched. However, the request can be served at a lower cost $2c$, if we choose to prefetch R_1 and evict some cached data saying d_i, and then after serving R_1, prefetch d_i back to the cache and evict R_1. Therefore, when $c \in [0, 1/2]$, the optimal policy always choose to prefetch the miss requests. Moreover, according to Theorem 2, we can conclude that π_P is the optimal policy for $c \in [0, 1/2]$.

For $c \in (1/2, 1]$, we will first show that r_P is upper bounded by $2c$. We know that π_P is the optimal policy for $c = 1/2$. For a given request sequence, let $cost^\circ$ denote the cost achieved by π_P with $c = 1/2$. Then, we have, for the same request sequence and $c > 1/2$

$$cost^\circ \leq cost(\pi^*) \leq cost(\pi_P),$$

and therefore

$$\frac{cost(\pi_P)}{cost(\pi^*)} \leq \frac{cost(\pi_P)}{cost^\circ} = \frac{c}{1/2} = 2c,$$

which implies $r_P \leq 2c$.

Next, we will show that there exists a request sequence such that a competitive ratio $2c$ is achieved, if $M \geq b + 2$, where M is the total number of data items and b is the cache size. Assume without loss of generality that d_i, $1 \leq i \leq b$, are initially stored in the cache. Consider a periodic request sequence that repeats the request pattern $R_{b+2}, R_1, R_2, \cdots, R_b, R_{b+1}, R_1, R_2, \cdots, R_b$. π_P will induce a cost $4c$ in each period. And the optimal policy is always fetching the missed data, which induces a cost 2 in each period. Therefore, a competitive ratio $2c$ is achieved and we can conclude that $r_P = 2c$.

Competitive ratio for π_A:

For $c \leq \sqrt{2}/2$, π_A always makes the same decision as π_P, and therefore, achieve the same competitive ratio. We have

$$r_A = \begin{cases} 1 & \text{for } c \in [0, 1/2], \\ 2c & \text{for } c \in (1/2, \sqrt{2}/2]. \end{cases}$$

For $\sqrt{2}/2 < c \leq 1$, we will first show that r_A is upper bounded by $1/c$. According to Theorem 3, we know that if π_A decides to prefetch the missed item, then prefetching must be the optimal choice. Therefore, π_A only yields worse performance than π^*, if π_A decides to fetch the missed item while π^* decides to prefetch.

We will introduce a new policy π° to bound the competitive ratio. Given a request sequence, let π° make the same decision as π_A if the decision is optimal. However, when the decision of π_A is not optimal (i.e., when π_A decides to fetch, while π^* decides to prefetch), let π° prefetch the missed data without evicting any cached items. Then, after serving the request, let π° evict the prefetched data. Note that π° breaks the cache capacity constraint and is not a feasible solution. We will use π° to provide a lower bound for the cost achieved by the optimal policy π^*. Specifically, for a given request sequence, let $cost^\circ$ denote the cost induced by π°. We will show $cost^\circ \leq cost(\pi^*)$ for any given request sequence.

Assume without loss of generality that R_1 is a miss and π_A decides to fetch R_1, while the optimal choice is to prefetch. According to Theorem 2, the optimal policy π^* will evict the farthest-in-future item R_σ and serve R_1 at a cost c. In contrast, the policy π° will keep R_σ in cache and serve R_0 at the same cost c. Since π_A chooses to fetch R_1, according to Condition C1, the next request for R_1 must arrive after the farthest-in-future item R_σ. Based on the farthest-in-future principle, the cache content of π° is more beneficial than that of π^*. Therefore, we have $cost^\circ \leq cost(\pi^*)$. In addition, since π_A and π° always have the same cache content before serving each request, we have $cost(\pi_A)/cost(\pi^\circ) \leq 1/c$, which yields

$$\frac{cost(\pi_A)}{cost(\pi^*)} \leq \frac{cost(\pi_A)}{cost(\pi^\circ)} \leq \frac{1}{c},$$

for any request sequence.

Next, we will show the lower bound for r_A when $\sqrt{2}/2 < c \leq 1$. Assume without loss of generality that $\{d_i, 1 \leq i \leq b\}$ are stored in the cache initially. The lower bound can be achieved by a periodic trace that repeats the request pattern $\{d_{b+1}, d_1, d_2, \cdots, d_b\}$. π_A will always choose to fetch the missed item. And the cache content under π_A is also updated periodically with period $b(b + 1)$. For the first $b(b + 1)$ requests, the approximation policy achieves a total cost b, and the optimal policy is to always prefetch the missed item which yields a cost $(b + 1)c$. Therefore, we have $r_A \geq b/((b + 1)c)$. □

In Theorem 4, we provide upper and lower bounds for the competitive ratio achieved by π_A, as well as the exact competitive ratios for π_F and π_P. Moreover, the upper and lower bounds for r_A are asymptotically tight as the cache size goes to infinity. We plot r_F, r_P and the upper bound of r_A in Figure 7. It is easy to conclude from Theorem 4 and observe in Figure 7 that

$$r_A \leq \min\{r_F, r_P\} \leq \sqrt{2},$$

for $\forall c \in [0, 1]$. The proposed approximation policy always achieves the smallest competitive ratio which is at most $\sqrt{2} \approx 1.414$. Therefore, π_A is near optimal in terms of the worst-case performance. Furthermore, in Section 8, we will verify that π_A also achieves near-optimal average performance for both synthetic and real data traces.

Note that applying π_P for $c \leq \sqrt{2}/2$ and π_F for $c > \sqrt{2}/2$ can achieve the same competitive ratio as π_A. However, experiments in Section 8 show that simply switching π_F and π_P at the threshold $c = \sqrt{2}/2$ will incur considerably larger average costs than the proposed approximation policy π_A.

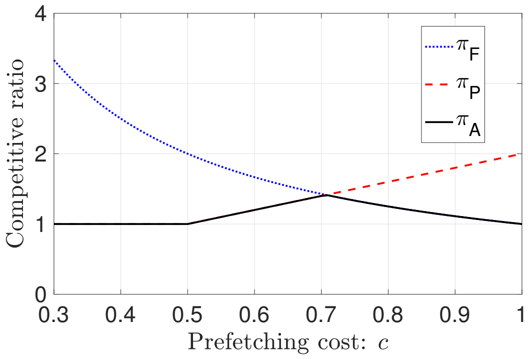


Figure 7: Competitive ratios of the proposed policies

7. Discussion and Generalization

In this section, we discuss the possibility of waiving the perfect prefetching time assumption, and the generalization of the proposed min-cost flow representation to support heterogeneous prefetching and fetching costs and variable data sizes.

7.1. Imperfect Prefetching Time

In our previous model, a perfect prefetching time is assumed to simplify the analysis. Specifically, we assume that the prefetched data item is loaded into the cache right before it is requested. In this section, we will argue that all the previous results will still hold, even when the prefetched data is loaded some time ahead of the request.

Recall that R_1 is the upcoming request and R_σ is the farthest-in-future item. Let $\{R_{-n}\}_{n\geq 1}$ be the sequence of historical requests as shown in Figure 8. Define

$$\theta = \min\{n \geq 1 : R_{-n} = R_\sigma\}.$$

$R_{-\theta}$ is the most recent request that is identical to R_σ. Assume that the policy decides to prefetch R_1. We claim that, it is equivalent to load R_1 into the cache at any time point in the interval $(\tau_{-\theta}, \tau_1)$, as long as the farthest-in-future eviction policy is adopted. According to the definition of θ, the farthest-in-future item is not requested in the time interval $(\tau_{-\theta}, \tau_1)$. Consequently, evicting R_σ (i.e., $R_{-\theta}$) and loading R_1 at any time between $\tau_{-\theta}$ and τ_1 does not impact the cost or the future decisions. In contrast, if R_1 is prefetched and loaded into the cache before $\tau_{-\theta}$ with R_σ evicted, then $R_{-\theta}$ will be a miss and can incur additional costs.

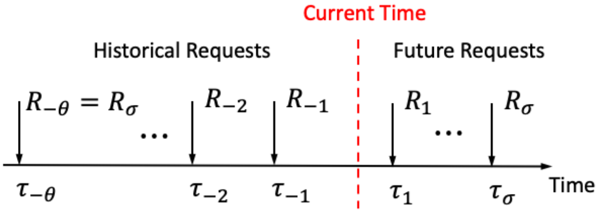


Figure 8: Timeline for data requests.

Since the pre-mentioned policies (including π_{OPT}, π_A and π_P) all adopt the farthest-in-future eviction, the main results of this paper still hold, if the prefetched data is loaded into the cache in the time interval $(\tau_{-\theta}, \tau_1)$. Specifically,

- The proposed policies (including π_{OPT}, π_A and π_P) still work. In particular, π_{OPT}, π_A and π_P will first make the decision of whether a data item should be prefetched or not. Then, we can find the $\tau_{-\theta}$ for each prefetched item and schedule the prefetching time accordingly.
- The π_{OPT} is still optimal, and the competitive ratio analysis for π_A, π_P and π_F still hold, since the optimal decision and the incurred cost will not be impacted if the prefetched data is loaded between $\tau_{-\theta}$ and τ_1.

The value of θ indicates the flexibility of choosing the prefetching time. When the requests are independently generated from a popularity distribution, θ is a random variable that depends on the cache size and is independent of the trace length and the prefetching cost. In Section 8, we will show that θ could be considerably large through simulations.

7.2. Heterogeneous Costs

In the previous setting, we assume that the prefetching/fetching costs are identical for every requests. However, in real practice, the cost to fetch a data item can depend on the traffic load and be time-varying. Similarly, the prefetching costs may also take different values to support more flexible model settings. Our flow-based method can be easily generalized to heterogeneous prefetching and fetching costs by setting the costs for prefetching and fetching edges in the flow network as the corresponding values. The cycle-canceling algorithm can still find the optimal solution as long as the costs are rational numbers.

7.3. Variable Data Sizes

For real CDN traces, data sizes can take disparate values ranging from a few bytes to gigabytes [45, 46]. The constructed flow network can be modified to accommodate variable data sizes. Specifically, for data item, we can set its surplus/demand and the capacities of prefetching, fetching, eviction and auxiliary edges as the data size. However, we may not be able to find the optimal policy via the min-cost flow, since it is possible that the min-cost flow prefetches/fetches a fraction of the data item, which is not feasible for a caching policy. Instead, by leverage the min-cost flow, we can construct the upper and lower bounds for the performance of the optimal offline policy.

The cost achieved by the min-cost flow is a lower bound for the cost achieved by the optimal policy, since fractional solutions are allowed for the min-cost problem. Additionally, we can construct a feasible caching policy from the min-cost flow by rounding up the fractional solution. Specifically, if a fraction of some request goes through the fetching edge in the flow network, then we will fetch the whole item. Otherwise, we will perform the same operation as the min-cost flow. The rounded solution is a feasible prefetching and caching policy and therefore provides an upper bound for the performance of the optimal policy. Notably, if the lower and upper bounds coincide, the rounded solution becomes optimal. It is shown in [15] that for caching without prefetching, the bounds are asymptotically tight under mild assumptions. Whether similar asymptotic results hold for prefetching deserves future investigations.

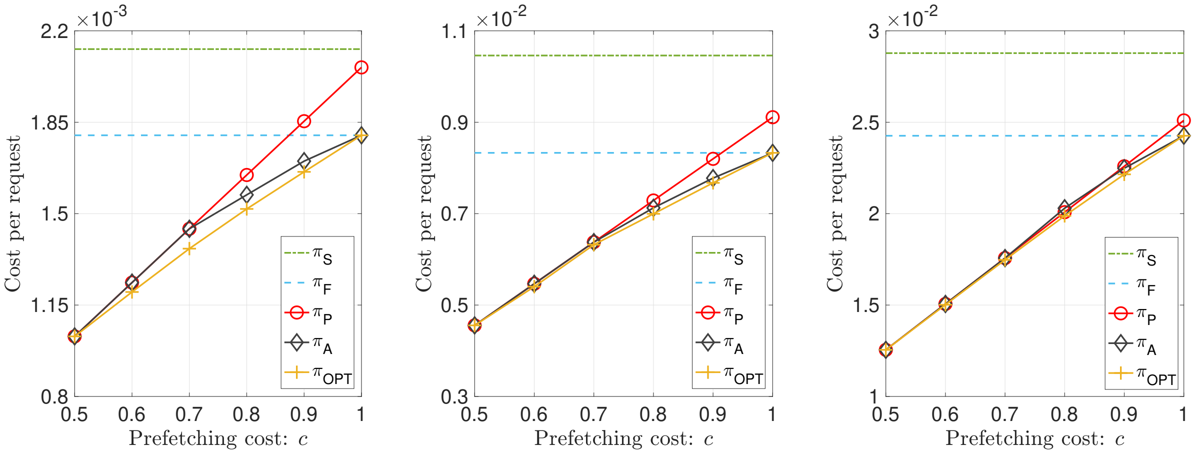
8. Evaluation

In this section, we evaluate the average performance for various policies of interest using both synthetic and real data traces. Specifically, in Experiment 1, we evaluate the policies for data requests generated from light-tailed and heavy-tailed popularity distributions. Real CDN traces are used for evaluation in Experiment 2. Moreover, in Experiment 3, we illustrate the amount of future information that is required by the approximation policy.

In addition to the pre-mentioned policies (π_{OPT}, π_A, π_P and π_F), we also simulate the optimal static policy (denoted by π_S). The optimal static policy stores the most popular data in the cache, and will neither update the cache content nor prefetch future requests. When the requests are generated independently from a popularity distribution, the optimal static policy can provide a lower bound for the costs incurred by a bunch of statistic-based policies (e.g., LRU, LFU) that only exploit data statistics and are unaware of the exact request sequence. In the following experiments, the cache is initially empty and the data items are prefetched, fetched or evicted based on specific policies.

Experiment 1. In this experiment, we compare the average performance of the proposed policies under both light-tailed and heavy-tailed data popularity distributions. In particular, we consider three popularity distributions with different tails. The first one is a light-tailed exponential distribution with $\mathbb{P}[R_n = d_i] = c_1 \cdot \exp(-0.3i)$, $1 \le n \le 10^5$, $1 \le i \le 10^6$, where $c_1 = 1/\sum_{i=1}^{10^6} \exp(-0.3i) \approx 0.3499$ is a normalization factor. The second popularity distribution is a heavy-tailed Weibull distribution with $\mathbb{P}[R_n = d_i] = c_2 \cdot \exp(-i^{0.6})$, $1 \le n \le 10^5$, $1 \le i \le 10^6$, where $c_2 \approx 0.8671$. The third one is a heavy-tailed Zipf's distribution with $\mathbb{P}[R_n = d_i] = c_3/i^2$, $1 \le n \le 10^5$, $1 \le i \le 10^6$, $c_3 \approx 0.6079$. For each popularity distribution, we generate a sequence of 10^5 requests independently and test the proposed policies for a cache of size 20. The average cost incurred by a request is plotted in Figure 9.

It can be observed from Figure 9a that the flow-based optimal offline policy π_{OPT} always incurs the smallest cost, and the proposed approximation policy π_A achieves near-optimal performance. The always-prefetching policy π_P



(a) Exponential popularities.

(b) Weibull popularities.

(c) Zipf's popularities.

Figure 9: Average costs under different popularities.

achieves the optimal performance when $c = 0.5$. For $c > 0.5$, π_P always incurs larger costs than π_{OPT}. Note that when c is close to 1, always prefetching can incur as high costs as the static optimal policy π_S, which loses the advantage of knowing the sample path. Moreover, the always-fetching policy π_F (a.k.a. the Belady's algorithm [38]) only achieves the optimal performance when the prefetching cost is 1. Although π_F and π_A have almost the same competitive ratios for $c \in [\sqrt{2}/2, 1]$, π_A achieves significantly better average performance when $c < 1$. Therefore, simply switching π_F and π_P at the threshold $\sqrt{2}/2$ (≈ 0.71) can incur much larger average costs than π_A and π_{OPT}.

The same trend can be observed for heavy-tailed Weibull distributions in Figure 9b. However, the performance difference between π_A and π_P are not as large as those for exponential distributions. Moreover, when the distribution tail is even heavier (e.g., for Zipf's popularities), both π_A and π_P achieve almost optimal performance as shown in Figure 9c. An intuitive explanation is that when the popularity is heavy-tailed, the requests will be less concentrated. As a result, it is more likely to have an unpopular data item stored in the cache, which provides a harmless eviction opportunity for prefetching. In such scenarios, prefetching is almost always the optimal choice, and therefore, π_A, π_P and π_{OPT} could achieve similar performance.

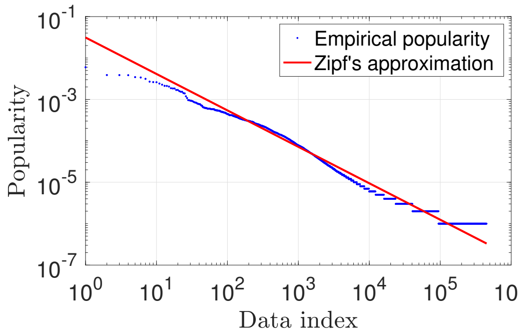


Figure 10: Popularity of CDN trace.

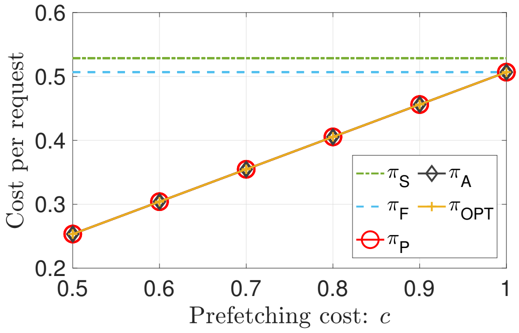
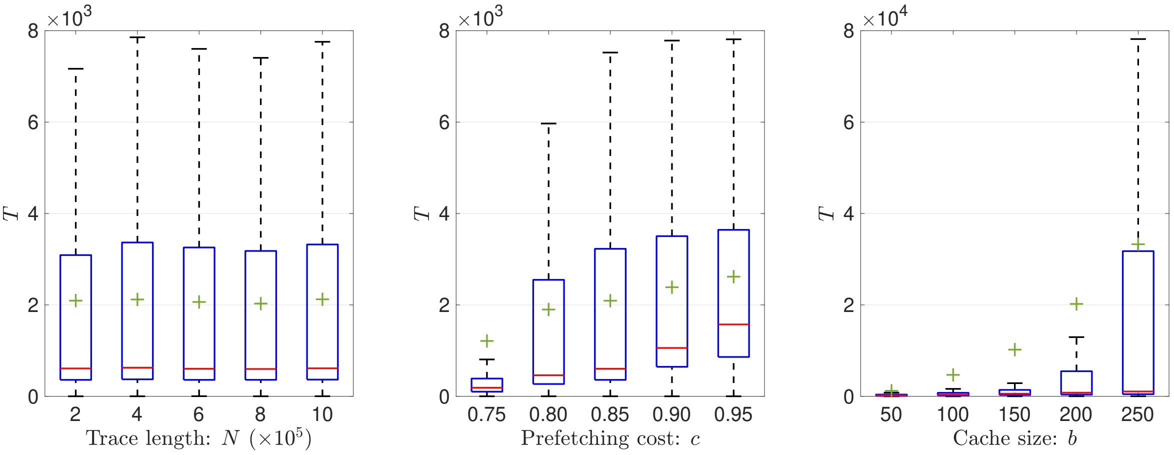


Figure 11: Average costs under CDN trace.

Experiment 2. In this experiment, we test the proposed policies using a real CDN data trace [1]. The trace contains a million requests for 449380 distinct data items where the data sizes are set to be 1. In Figure 10, we plot the empirical data popularities which can be approximated by a Zipf's distribution with $\mathbb{P}[R_n = d_i] = 0.0313/i^{0.88}$, $1 \leq i \leq 449380$, $1 \leq n \leq 10^6$. We simulate the proposed policies for a cache of size 20000 using the request sequence in the CDN trace. The average costs are plotted in Figure 11. It can be observed that π_A and π_P achieve almost the same performance as π_{OPT}, which coincides with the observation in Experiment 1 for Zipf's popularities.

[1] The trace is originally used for evaluation and labeled as "cdn1" in [15]. We use the first one million requests in the trace.

Experiment 3. In this experiment, we will verify that the approximation policy π_A only requires near-future information to make decisions for a single request. Let T denote the number of future requests required for making decision for a request. We will evaluate the statistics of T under different trace lengths, prefetching costs and cache sizes. Generate data requests from the Zipf's popularity distribution considered in Experiment 1. Note that we omit the experiments for exponential and Weibull distributions since the obtained results and the revealed insights are similar. First, set $c = 0.85$, $b = 50$. We collect the statistics of T for multiple traces with different lengths. The results are presented as box plots in Figure 12a, where the central red bar and the green "+" sign represent the median and the mean, respectively. The top and bottom edges of the box indicate the 75th and 25th percentiles. The whiskers extend to the extreme values within $1.5 \times IQR$ (interquartile range) [47]. All these statistics of T do not scale with the trace length N, which verifies the analysis in Section 6.



(a) Impact of trace lengths.

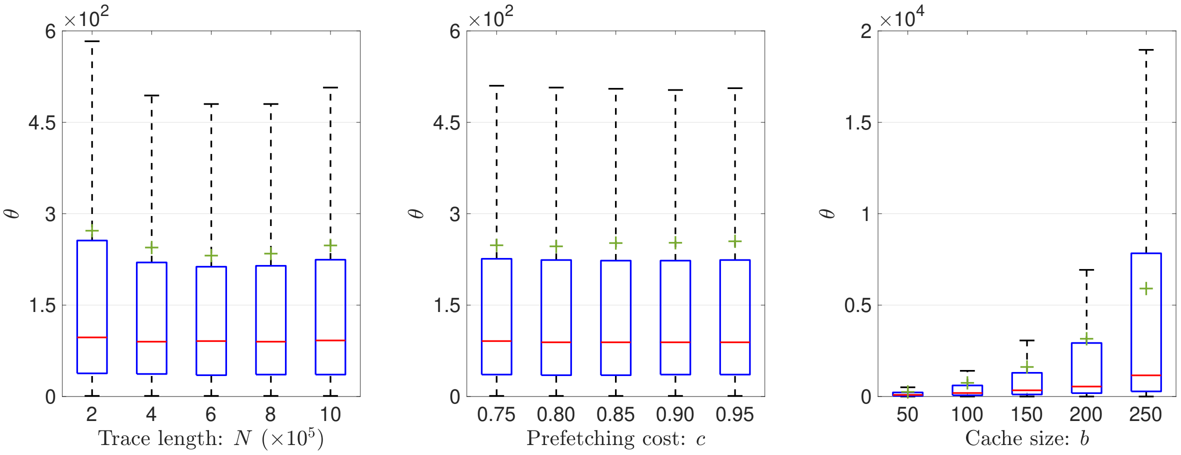
(b) Impact of prefetching costs.

(c) Impact of cache sizes.

Figure 12: Required information for making prefetching decisions.

Furthermore, we investigate how the cache size and the prefetching cost can impact T. Set $N = 5 \times 10^5$. For a fixed $b = 50$, we collect the statistics of T under different prefetching costs. The results are plotted in Figure 12b. For a fixed $c = 0.85$ and repeat the experiments for different cache sizes, and present the results in Figure 12c. As we expect, the required information will increase with the prefetching cost, since Condition $C2$ of π_A becomes stricter for a larger c. Moreover, T can scale considerably with the cache size, because τ_σ and τ_ω will take larger values, and the miss ratio will decrease. Consequently, more future observations are required to check whether Conditions $C1$ and $C2$ hold. For a typical scenario where the policy is applied for a fixed cache size and processes requests that keep arriving, the future information required by π_A will not scale up.

Experiment 4. It is shown in Section 7 that the main results of this paper will still hold as long as the prefetched data is loaded into the cache in the time interval $(\tau_{-\theta}, \tau_1)$ for R_1. And the value of θ could represent the flexibility of choosing the prefetching time. In this experiment, we show that θ could be considerably large by evaluating its statistics for the Zipf's popularity distribution defined in Experiment 1. Specifically, we first set $c = 0.85$ and $b = 50$, and conduct simulations for different trace lengths. Then, we repeat the experiment for $N = 5 \times 10^5$, $b = 50$ and different prefetching costs. Next, we conduct simulations for $N = 5 \times 10^5$, $c = 0.85$ and different cache sizes. The results are shown as the box plots in Figure 13, where the symbols represent the same statistics as those in Experiment 3. It can be observed from Figure 13a and 13b that θ will not be impacted by the trace length or the prefetching cost. The mean of θ is round 250, which is considerably large. Moreover, as shown in Figure 13c, θ will increase with the cache size. For $b = 250$, the mean of θ could be as large as 5907, which brings even more flexibility to choose the prefetching time. The intuition is that, for large cache sizes, the farthest-in-future item will be less popular. And therefore, the most recent time when the farthest-in-future item was requested (i.e., $\tau_{-\theta}$) is far in the past.



(a) Impact of trace lengths.

(b) Impact of prefetching costs.

(c) Impact of cache sizes.

Figure 13: Flexibility of prefetching time.

9. Conclusion

To characterize the fundamental trade-off between prefetching and caching, we developed a cost-based service model and investigated the optimal offline policy, assuming that the entire request sequence is known. We casted it as a min-cost flow problem, and found the optimal policy for a data trace of length N via flow-based algorithms in $O(N^{3/2})$ expected time. To apply this offline algorithm without the precise knowledge on future requests, we utilized the characteristics of the optimal solution and derived non-trivial conditions for an optimal prefetching and eviction policy. Based on these insights, we proposed a lightweight approximation policy using the predicted requests in the near future. The approximation policy can be applied in real time and process the entire trace in $O(N)$ expected time, with a competitive ratio $\sqrt{2} \approx 1.4$. Extensive experiments verified that it achieves near-optimal average performance for both light and heavy-tailed popularity distributions.

References

[1] E. T. Dao, Predictive prefetching of web content, US Patent App. 14/311,699 (Dec. 25 2014).

[2] L. Ariyasinghe, C. Wickramasinghe, P. Samarakoon, U. Perera, R. P. Buddhika, M. Wijesundara, Distributed local area content delivery approach with heuristic based web prefetching, in: 2013 8th International Conference on Computer Science & Education, IEEE, 2013, pp. 377–382.

[3] D. Gmach, J. Rolia, L. Cherkasova, A. Kemper, Workload analysis and demand prediction of enterprise data center applications, in: 2007 IEEE 10th International Symposium on Workload Characterization, IEEE, 2007, pp. 171–180.

[4] S. Rahman, M. Burtscher, Z. Zong, A. Qasem, Maximizing hardware prefetch effectiveness with machine learning, in: 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems, IEEE, 2015, pp. 383–389.

[5] N. B. Hassine, R. Milocco, P. Minet, Arma based popularity prediction for caching in content delivery networks, in: 2017 Wireless Days, IEEE, 2017, pp. 113–120.

[6] M. Aloui, H. Elbiaze, R. Glitho, S. Yangui, Analytics as a service architecture for cloud-based cdn: Case of video popularity prediction, in: 2018 15th IEEE Annual Consumer Communications & Networking Conference (CCNC), IEEE, 2018, pp. 1–4.

[7] Z. Huang, D. Zeng, H. Chen, A comparison of collaborative-filtering recommendation algorithms for e-commerce, IEEE Intelligent Systems 22 (5) (2007) 68–78.

[8] G. Adomavicius, A. Tuzhilin, Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions, IEEE transactions on knowledge and data engineering 17 (6) (2005) 734–749.

[9] A. Sidiropoulos, G. Pallis, D. Katsaros, K. Stamos, A. Vakali, Y. Manolopoulos, Prefetching in content distribution networks via web communities identification and outsourcing, World Wide Web 11 (1) (2008) 39–70.

[10] Y. Chen, L. Qiu, W. Chen, L. Nguyen, R. H. Katz, Efficient and adaptive web replication using content clustering, IEEE Journal on Selected Areas in Communications 21 (6) (2003) 979–994.

[11] J. Kangasharju, J. Roberts, K. W. Ross, Object replication strategies in content distribution networks, Computer Communications 25 (4) (2002) 376–383.

[12] A. Nanopoulos, D. Katsaros, Y. Manolopoulos, A data mining algorithm for generalized web prefetching, IEEE transactions on knowledge and data engineering 15 (5) (2003) 1155–1169.

[13] A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, M. Dahlin, The potential costs and benefits of long-term prefetching for content distribution, Computer Communications 25 (4) (2002) 367–375.

[14] B. Wu, A. D. Kshemkalyani, Objective-optimal algorithms for long-term web prefetching, IEEE Transactions on Computers 55 (1) (2005) 2–17.

[15] D. S. Berger, N. Beckmann, M. Harchol-Balter, Practical bounds on optimal caching with variable object sizes, Proceedings of the ACM on Measurement and Analysis of Computing Systems 2 (2) (2018) 1–38.

[16] D. S. Berger, Towards lightweight and robust machine learning for cdn caching, in: Proceedings of the 17th ACM Workshop on Hot Topics in Networks, 2018, pp. 134–140.

[17] P. Cao, E. W. Felten, A. R. Karlin, K. Li, A study of integrated prefetching and caching strategies, ACM SIGMETRICS Performance Evaluation Review 23 (1) (1995) 188–197.

[18] T. Kimbrel, P. Cao, E. W. Felten, A. R. Karlin, K. Li, Integrated parallel prefetching and caching, ACM SIGMETRICS Performance Evaluation Review 24 (1) (1996) 262–263.

[19] M. Kallahalla, P. J. Varman, Pc-opt: optimal offline prefetching and caching for parallel i/o systems, IEEE Transactions on Computers 51 (11) (2002) 1333–1344.

[20] M. Abrams, C. R. Standridge, G. Abdulla, S. Williams, E. A. Fox, Caching proxies: Limitations and potentials, Tech. rep., Department of Computer Science, Virginia Polytechnic Institute & State University (1995).

[21] S. Albers, C. Witt, Minimizing stall time in single and parallel disk systems using multicommodity network flows, in: Approximation, Randomization, and Combinatorial Optimization: Algorithms and Techniques, Springer, 2001, pp. 12–24.

[22] S. Albers, M. Büttner, Integrated prefetching and caching in single and parallel disk systems, Information and Computation 198 (1) (2005) 24–39.

[23] J. Tadrous, A. Eryilmaz, H. El Gamal, Proactive resource allocation: Harnessing the diversity and multicast gains, IEEE Transactions on Information Theory 59 (8) (2013) 4833–4854.

[24] J. Tadrous, A. Eryilmaz, H. El Gamal, Proactive content download and user demand shaping for data networks, IEEE/ACM Transactions on Networking 23 (6) (2014) 1917–1930.

[25] A. Jain, C. Lin, Rethinking belady's algorithm to accommodate prefetching, in: 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), IEEE, 2018, pp. 110–123.

[26] S. Albers, S. Arora, S. Khanna, Page replacement for general caching problems, in: SODA, Vol. 99, Citeseer, 1999, pp. 31–40.

[27] A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. Naor, B. Schieber, A unified approach to approximating resource allocation and scheduling, Journal of the ACM (JACM) 48 (5) (2001) 1069–1090.

[28] S. Irani, Page replacement with multi-size pages and applications to web caching, in: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing, 1997, pp. 701–710.

[29] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely Jr, J. Emer, Pacman: prefetch-aware cache management for high performance caching, in: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, 2011, pp. 442–453.

[30] W.-F. Lin, S. K. Reinhardt, D. Burger, Reducing dram latencies with an integrated memory hierarchy design, in: Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture, IEEE, 2001, pp. 301–312.

[31] J. Griffioen, R. Appleton, Reducing file system latency using a predictive approach., in: USENIX summer, 1994, pp. 197–207.

[32] R. H. Patterson, G. A. Gibson, Exposing i/o concurrency with informed prefetching, in: Proceedings of 3rd International Conference on Parallel and Distributed Information Systems, IEEE, 1994, pp. 7–16.

[33] M. Sun, H. Chen, B. Shu, Predict-then-prefetch caching strategy to enhance qoe in 5g networks, in: 2018 IEEE World Congress on Services (SERVICES), IEEE, 2018, pp. 67–68.

[34] F. Zhang, C. Xu, Y. Zhang, K. Ramakrishnan, S. Mukherjee, R. Yates, T. Nguyen, Edgebuffer: Caching and prefetching content at the edge in the mobilityfirst future internet architecture, in: 2015 IEEE 16th International Symposium on a World of wireless, mobile and multimedia networks (WoWMoM), IEEE, 2015, pp. 1–9.

[35] E. Bastug, M. Bennis, M. Debbah, Living on the edge: The role of proactive caching in 5g wireless networks, IEEE Communications Magazine 52 (8) (2014) 82–89.

[36] A. Sadeghi, F. Sheikholeslami, G. B. Giannakis, Optimal and scalable caching for 5G using reinforcement learning of space-time popularities, IEEE Journal of Selected Topics in Signal Processing 12 (1) (2017) 180–190.

[37] S. Albers, N. Garg, S. Leonardi, Minimizing stall time in single and parallel disk systems, Journal of the ACM (JACM) 47 (6) (2000) 969–986.

[38] L. A. Belady, A study of replacement algorithms for a virtual-storage computer, IBM Systems journal 5 (2) (1966) 78–101.

[39] C. A. Floudas, P. M. Pardalos, Encyclopedia of optimization, Springer Science & Business Media, 2008.

[40] R. Becker, M. Fickert, A. Karrenbauer, A novel dual ascent algorithm for solving the min-cost flow problem, in: 2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX), SIAM, 2016, pp. 151–159.

[41] R. Becker, A. Karrenbauer, A combinatorial o (m 3/2)-time algorithm for the min-cost flow problem, arXiv preprint arXiv:1312.3905 129 (2013).

[42] S. I. Daitch, D. A. Spielman, Faster approximate lossy generalized flow via interior point algorithms, in: Proceedings of the fortieth annual ACM symposium on Theory of computing, 2008, pp. 451–460.

[43] Y. Yang, J. Zhu, Write skew and zipf distribution: Evidence and implications, ACM transactions on Storage (TOS) 12 (4) (2016) 1–19.

[44] R. L. Mattson, J. Gecsei, D. R. Slutz, I. L. Traiger, Evaluation techniques for storage hierarchies, IBM Systems journal 9 (2) (1970) 78–117.

[45] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al., Scaling memcache at facebook, in: Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13), 2013, pp. 385–398.

[46] Q. Huang, K. Birman, R. Van Renesse, W. Lloyd, S. Kumar, H. C. Li, An analysis of facebook photo caching, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, 2013, pp. 167–181.

[47] W. C. Navidi, Statistics for engineers and scientists, McGraw-Hill Higher Education New York, NY, USA, 2008.