Connection Caching: Model and Algorithms*

Edith Cohen
AT&T Labs–Research
180 Park Avenue
Florham Park, NJ 07932 USA
edith@research.att.com

Haim Kaplan *Uri Zwick*
School of Computer Science
Tel-Aviv University
Tel-Aviv 69978, Israel
{haimk,zwick}@cs.tau.ac.il

Abstract

We introduce a theoretical model for connection caching. In our model each host maintains (caches) a limited number of open connections to other hosts. A request may utilize an open connection in which case it is a *hit*, or it may require to open a new connection in which case it is a *miss*. Establishment of a new connection may force termination (eviction) of another connection at each of the endpoints. The goal is to serve the request sequence with minimum number of misses. This model differs from the standard caching model as it involves many caches which affect each other: a decision to terminate a connection by one node affects the cache of another node that is forced to accept the termination. Our motivation to study the problem stems from Web applications, namely the transmission of HTTP (Hyper Text Transfer Protocol) messages over persistent TCP (Transmission Control Protocol) connections.

We consider both the off-line connection caching problem where the request sequence is given in advance, and the online connection caching problem, where the algorithm has to serve a request when it arrives without knowledge of future requests. In the off-line settings we show that finding the optimal strategy is NP-hard. We also derive natural algorithms from the optimal cache replacement algorithm for standard caching and prove that the miss rate of these algorithms is within a factor of 2 from optimal.

In the online setting we study several families of distributed algorithms that can be implemented by running an independent process at each node. The algorithms differ by the amount of communication which they utilize between pairs of hosts engaged in an open connection. We show optimal k-competitive deterministic algorithms that utilize one communication bit per open connection, where k is the size of the largest cache in the network. On the other hand without such communication bit the best algorithms which we describe are only $(2k-1)$-competitive. We also analyze what one can gain by using randomization at different levels of allowed communication.

1 Introduction

We present a theoretical model for *connection-caching* that generalizes the standard caching model. In this model each host has a cache of limited size which it uses to hold connections. A connection is considered open only if it is cached by *both* its endpoints. To serve a request for a connection (u, v) we have to open the connection (u, v), if it is not already open. Before opening the connection (u, v), each one of the endpoints u and v may have to shut down another connection, to make room in its cache for the new connection. (A precise definition of the connection caching model is given in the next section.)

*Preliminary versions of the results contained in this paper appeared in [10], and [11].

A fundamental difference between *connection-caching* and standard caching, is that connection-caching involves multiple distributed caches that affect each other. Each open connection is cached by two entities. If one of the entities decides to terminate ("evict") the connection, the other entity is forced to accept this decision. We mainly consider scenarios where communication is also limited, and thus, replacement decisions at each node are made without full knowledge of the state of other caches.

The connection-caching model is motivated by several important applications:

- **HTTP over TCP**.

  Information exchange between clients and servers in the Web is performed using HTTP (Hyper Text Transfer Protocol). HTTP uses TCP (Transmission Control Protocol) as the underlying transport protocol. Thus, request and response messages of HTTP are carried on a TCP connection established between the two parties.

  A typical HTTP session includes multiple requests: Web pages often consist of an HTML page and multiple embedded images, each requested separately. Users are also likely to follow up with further requests to the same server. These requests can be carried on existing TCP connections or necessitate new connection establishment. In the latter case, additional latency is incurred (due to 3-way handshake and "TCP slow start").

  Studies indicate that connection-establishment is a significant component of request-response time [12] and that a short TCP connection associated with a single HTTP request-response rarely gets past the "slow start" phase [17]. With *persistent HTTP* [23, 14], TCP connections are kept open and re-used to carry imminent future HTTP requests. Persistent connections are a default under HTTP/1.1. A Connection is kept open, and idle, for a certain period of time until either a new request arrives or, until the connection is closed by the client or the server.

  With persistent connections, clients and servers keep many connections open, some for idle periods, in anticipation of future requests. Requests that utilize an already-open connection benefit from reduced latency. Recent study [12] indicates that connection-caching and re-use at proxy caches is more effective for reducing user-perceived latency than document caching. Since TCP is implemented on top of the connection-less IP, the overhead associated with keeping connection state information only exists at the two endpoints. Open connections consume a socket and memory for socket buffers, at each end. Since memory and sockets are limited, each entity – individual client, proxy, or Web server, has a limit on the number of TCP connections that can be simultaneously opened.

  With the increased deployment of HTTP/1.1, persistent TCP connections become cacheable re-usable resources. Indeed, HTTP/1.1 compliant Web browsers, proxies, and servers (for example [3]) implement policies to manage TCP connections. Various policies for server-side connection management were proposed and studied at [9].

- **IP over ATM**. Connections are treated as a cacheable resource in the context of maintaining virtual circuits for routing IP-traffic over ATM networks. This was studied extensively (see [20, 21, 19]), but these studies treated it as a standard paging problem.

- **Secure connections**. An application (pointed to us by Dan Boneh) is the caching of secure connections. For example, secure HTTP (HTTPS) utilizes SSL. Authentication performed at connection establishment time consumes extensive resources. Thus caching these connections reduces the total resource utilization.

We explore fundamental properties of the connection-caching model. Since connection-caching is an inherently distributed problem, different algorithms use different types and amounts of communication between hosts. We consider three different models of communication. In the first and most basic type-1 model, hosts are allowed no communication whatsoever. In particular, when a host close a connection it does not notify the other endpoint about its decision. In this model, a connection (u, v) that was closed by v may still reside in the cache of u. In the second, type-2 model, hosts are notified of connection closures but are allowed no additional communication. Here, when a connection is closed, it is immediately removed from the caches of both its endpoints. In the third, type-3 model, hosts are notified of connection closures as with type-2. In addition to that, hosts are allowed to exchange (a minimal amount of) information regarding shared open connections. We refer to algorithms operating under these three models as *type-1*, *type-2* and *type-3* algorithms, respectively.

We can convert any standard cache replacement A into a connection-caching algorithm by running the standard replacement algorithm locally at each node of the network. In fact, two different connection-caching algorithms are obtained. We obtain a type-1 algorithm $\mathcal{A}_1$ by running A locally at each node without notifying the other end of connection closures, and a type-2 algorithm $\mathcal{A}_2$ if nodes do notify the other endpoints when they close connections. It may seem that $\mathcal{A}_1$ should perform worse than $\mathcal{A}_2$ on every request sequence, as $\mathcal{A}_1$ operates with less information available. We show that this is true for some natural algorithms. On the other hand we also show that it is not true for all algorithms.

For the HTTP over TCP application, type-2 communication models TCP under normal network conditions. Type-1 communication models operation under intermittent connectivity and crashes of host systems. Type-3 communication is not currently supported for TCP connections. We believe, however, that it is important to understand its potential benefits. The fundamental question that we try to explore is the relative power gained by the different levels of communication.

Our results. We first establish that, in contrast to standard paging, the off-line connection-caching problem of minimizing the number of requests encountering a closed connection ("misses") is NP-complete, and also APX-complete [4], i.e., there is no polynomial time approximation scheme (PTAS) for it, unless P=NP.

Utilizing the close connection between a standard (randomized or deterministic) caching algorithm A and its type-1 counterpart $\mathcal{A}_1$, we show that if A has competitive ratio c, then $\mathcal{A}_1$ has competitive ratio between c and $2c$. The factor of 2 here is tight in the sense that there are single-cache algorithms that are c-competitive, while their corresponding connection-caching algorithm is only $2c$-competitive.

This general result implies, for example, that $\mathcal{LRU}_1$, the type-1 algorithm derived from LRU (Least Recently Used), is $2k$-competitive, where k is the size of the largest cache in the network. It follows, however, from our more general results described below, that $\mathcal{LRU}_1$ is in fact $(2k-1)$-competitive. We also show that for every request sequence, $\mathcal{LRU}_2$, the type-2 connection-caching algorithm derived from LRU, performs at least as well as $\mathcal{LRU}_1$ so $\mathcal{LRU}_2$ is also $(2k-1)$-competitive. We provide an example showing that these upper bounds on the performance of $\mathcal{LRU}_1$ and $\mathcal{LRU}_2$ are tight.

We also consider the off-line algorithm LFD (Largest Future Distance, also known as Belady's algorithm). As LFD is an optimal single-cache caching algorithm, we get from the general result stated above, that $\mathcal{LFD}_1$ is a 2-approximation algorithm for the off-line connection-caching problem. We further show that $\mathcal{LFD}_2$ is also a 2-approximation algorithm, and that this upper bound is tight for both $\mathcal{LFD}_1$ and $\mathcal{LFD}_2$. Interestingly, in contrast to $\mathcal{LRU}_1$ and $\mathcal{LRU}_2$, the performances of

$\mathcal{LFD}_1$ and $\mathcal{LFD}_2$ are incomparable. We show sequences for which $\mathcal{LFD}_1$ outperforms $\mathcal{LFD}_2$ and vice versa.

Standard deterministic caching algorithms are known to have competitive ratio of at least k [26]. On the other hand, a large class of algorithms, called the marking algorithms, (which include LRU, FIFO and other natural algorithms), is known to be optimal, that is, all algorithms in the class have a competitive ratio of exactly k (see [13]). We define corresponding classes of connection caching algorithms and explore the performance of each class.

We define three families of deterministic *marking* connection-caching algorithms, one for each of the communication models. These policies are derived from marking algorithms for the standard caching problem. We show that all marking connection-caching algorithms of type-1 and type-2 are $(2k-1)$-competitive, and more interestingly, that all type-3 marking algorithms are k-competitive. Type-3 marking algorithms exchange only a single bit of information per open connection. These bounds are tight, since as we already mentioned that $\mathcal{LRU}_1$ and $\mathcal{LRU}_2$ which constitute type-1 and type-2 marking algorithm, respectively, are no better than $(2k-1)$-competitive. Furthermore, we also show that a type-2 marking algorithm called $\mathcal{FWF}$, which is derived from the single-cache algorithm FWF (Flush When Full) [18] is also k-competitive. The algorithm FWF (and $\mathcal{FWF}$) evicts pages as soon as they are unmarked, and hence, is not a *demand-caching* algorithm (A demand-caching algorithm is an algorithm that fills all empty slots before initiating evictions). We do not know of a demand-paging type-2 or type-1 algorithm which is better than $(2k-1)$-competitive.

We also define and analyze another wide class of *conservative* connection-caching algorithms. Our class of *conservative* connection-caching algorithms generalizes the class of conservative single cache algorithms defined by Young [30] (see also [6]). Conservative caching algorithms include FIFO (First In First Out) and LRU and are known to have a competitive ratio of k in the single cache setting [30, 6]. We show that *conservative* connection-caching algorithms are $2k$-competitive and furthermore we define a natural subclass of the *conservative* connection-caching algorithms containing algorithms which are all $(2k-1)$-competitive.

Our results for deterministic connection-caching algorithms extend to the more general (k,h)-connection caching model, where at each node v with cache size $k(v)$ the off-line adversary is only allowed to use a cache of size $h(v) \leq k(v)$.

We also analyze randomized marking connection-caching algorithms in the three communication models. These algorithms are derived from the single-cache algorithm MARK introduced by Fiat et al [13]. Algorithm MARK maintains a single bit for each cached item, which is set when the item is requested. When an eviction is due, MARK simply selects a page to evict uniformly at random from all unmarked pages (if all pages are marked it first unmarks them all). Fiat et al [13] proved that MARK is $2H_k$-competitive in the single-cache setting. (A more complicated proof gives an upper bound of $2H_k - 1$ which is tight [1]). The general result mentioned above implies that $\mathcal{MARK}_1$, is $4H_k$-competitive (in fact $4H_k - 2$ using the tight bound of [1]). We prove here that $\mathcal{MARK}_2$ is also $4H_k$-competitive. Due to its distributed nature, the analysis of $\mathcal{MARK}_2$ is much more involved than for the standard caching and type-1 variants.

Another natural connection caching algorithm derived from MARK is a type-3 algorithm, denoted by $\mathcal{MARK}_3$, in which there is only one mark bit associated with a connection that is accessible by both endpoints. Curiously, we construct request sequences that show that $\mathcal{MARK}_3$ is $\Omega(k)$-competitive. The example demonstrates how the extra bit of communication allows the adversary to impose a deterministic behavior on $\mathcal{MARK}_3$. These results contrast with our results for deterministic marking algorithms, where the extra bit of communication provably helps. Table 1 summarizes our results about the performance of deterministic and randomized marking algorithms.

	Deterministic Marking Algorithms	Randomized Marking Algorithms
Type-1	$2k-1$	$2(2H_k-1)$
Type-2	$2k-1$	$4H_k$
Type-3	k	k

Table 1: Upper bounds on the competitive ratios of all deterministic or randomized marking algorithms.

Recent related work. As in the case of a single-cache [15, 28, 29], the connection-caching model can be generalized by allowing connections to have varying sizes and fetching costs. In the HTTP over TCP context, sizes may correspond to number of socket buffers allocated. Establishment costs may capture the user-perceived latency caused by connection-establishment, and thus be asymmetric and vary according to connection-establishment-time or quality-of-service class. Albers [2] refers to this problem as *generalized connection caching.* Independently of some of our work, Albers proves [2] that a type-3 version of the Landlord algorithm proposed by Young [29] has competitive ratio of k. The Landlord algorithm generalizes marking algorithms and hence this result generalizes our Theorem 6.2. Albers also shows that the Harmonic algorithm [25] (which is randomized) is k competitive for generalized connection caching and that the competitive ratio of the BALANCE algorithm [7] is no better than $2k-1$. We discuss and analyze different-type castings of Landlord as a connection-caching algorithm in Section 6.2.

The structure of the rest of the paper is as follows. Section 2 includes necessary definitions. Section 3 establishes hardness results for the off-line problem. Section 4 establishes a general relation between the competitive ratio of a caching algorithm X and the connection-caching algorithm $\mathcal{X}_1$. Section 5 analyses the performance of $\mathcal{LFD}_2$. In Section 6 we define and analyst deterministic marking algorithms under the different communication models, and in Section 7 we do so for randomized marking algorithms. Section 8 is concerned with conservative algorithms. Finally, Section 9 highlights remaining open questions.

2 The connection caching model

We begin by defining our connection caching model. Let $G=(V,E)$ be an undirected graph representing a communication network. The nodes of V correspond to nodes in the network. The edges of E correspond to possible (virtual) connections between them. Each possible connection $e \in E$ may be either *open* or *closed*. At any given time, a node $v \in V$ may maintain at most $k(v)$ open connections. The parameter $k(v)$ is referred to as the size of the *connection cache* of v. A connection (u,v) is *open* if it is in the connection cache of u and in the connection cache of v and it is *closed* otherwise. The graph $H=(V,F)$ composed of all the open connections is therefore a subgraph of $G=(V,E)$ in which the degree of each node v is at most $k(v)$. We denote by k the size of the largest connection cache in the network, i.e. $k=\max_v\{k(v)\}$.

Initially all connections are closed. The network is then supposed to service a sequence $\sigma = \langle\sigma_1,\dots,\sigma_\ell\rangle$ of *connection requests*. Each request $\sigma_i \in E$ is an edge in the graph $G=(V,E)$. If the connection σ_i is currently open, then the request can be served at no cost. This is called a *hit.* If the connection $\sigma_i=(u,v)$ is currently closed, then it must be opened (or reopened) to service the request. This is called a *miss.* If u already maintains $k(u)$ open connections, then one of these connections must be closed before the connection $\sigma_i = (u, v)$ can be opened. The same is true for v. Opening a connection may thus force the closure, and eviction, of two other connections. A *connection caching algorithm* is a scheme by which the network takes eviction decisions.

While servicing σ each opening of a connection (i.e. a miss) incurs a fixed activation cost. The cost of servicing the sequence is the sum of these activation costs. The connection caching problem is to find an algorithm which minimizes the total activation cost incurred while servicing the sequence σ.[1] Throughout most of the paper we shall assume that the activation cost is one for every connection. Albers [2] has recently addressed the connection caching model when activation costs and cache sizes vary.

We consider the connection caching problem both in an off-line and in an online settings. In the off-line connection caching problem the request sequence is given in advance and the connection caching algorithm has to find the optimal way to serve it. In the online connection caching problem the connection caching algorithm has to serve each request at the moment when it arrives without any knowledge of forthcoming requests. Let $\text{miss}(\sigma)$ be the minimum possible cost to serve the sequence σ, and let $\text{miss}(\mathcal{ALG}, \sigma)$ be the cost incurred by algorithm $\mathcal{ALG}$ while serving σ. In case $\mathcal{ALG}$ is an off-line algorithm we say that it is a c-approximation algorithm if for every σ $\text{miss}(\mathcal{ALG}, \sigma) \leq c * \text{miss}(\sigma) + d$ for some constant d. In case $\mathcal{ALG}$ is an online algorithm we say that it is c-competitive if for every σ, $\text{miss}(\mathcal{ALG}, \sigma) \leq c * \text{miss}(\sigma) + d$ for some constant d. In case $\mathcal{ALG}$ satisfies $\text{miss}(\mathcal{ALG}, \sigma) \leq c * \text{miss}(\sigma)$ (i.e. $d = 0$) we say that it is a strongly c-competitive algorithm or a strongly c-approximation algorithm respectively. Note that in the online setting c will typically be a function of the sizes of the connection caches.

A common generalization of the competitive ratio in the single cache setting is the following. A caching algorithm A is said to be $c(h, k)$-competitive, if for every sequence of requests, the number of misses incurred while A service the sequence using a cache of size k is at most $c(h, k)$ times the minimum number of misses required to service the sequence with a cache of size h, plus perhaps a fixed constant. In the previous definition of competitiveness, h was taken to be k. As we are comparing here the behavior of an on-line algorithm with the optimal off-line algorithm, we sometimes want to weaken the power the off-line algorithm. We can do that by taking $h < k$.

Similarly we can extend this definition to the connection caching setup. Let $G = (V, E)$ be the network and let k and h be two functions assigning cache sizes to the vertices of the network, such that for every v, $h(v) \leq k(v)$. We say that a connection caching algorithm $\mathcal{A}$ is $c(h(v), k(v))$-competitive if the miss rate of $\mathcal{A}$ when the size of the connection cache at v is $k(v)$ is at most $c(k(v), h(v))$ times the minimum miss rate with a cache of size $h(v)$ at every vertex v.

2.1 Types of distributed connection caching algorithms

As described in the introduction we will be mostly interested in distributed online connection caching algorithms. A *distributed* connection-caching algorithm, or policy, is an algorithm in which eviction decisions at every node v of the network are taken based only on information associated with the connections incident to v. A distributed connection caching algorithm can be implemented by an independent process running at every node of the network. The process that runs at v decides which connection to evict from the connection cache of v when we need to serve a miss (v, w) and the connection cache of v is full. The information associated with the connections incident to v (and available to v when it takes decisions) may depend not only on v but also on the other endpoints of

[1]In cases it is hard to find such the optimal algorithm we would be interested to find algorithms with some guarantees on their worst case cost.

these connections. We will distinguish between different distributed connection-caching algorithms by the amount of communication allowed between v and the other hosts it is connected to.

First we distinguish between *type-1* and *type-2* connection caching algorithms. In a type-1 algorithm when node v decides to evict the connection (v, u) in order to accomodate a request (v, w) it does not notify u about this decision. In contrast when a type-2 algorithm evicts a connection (v, u) it instantanously notifies u about this decision. For both types of algorithms when v evicts an open connection (v, u) we say that the connection is *dumped* from the cache of u by this decision. A dumped connection leaves a *hole* in the cache in which it is dumped from, i.e., a slot in the cache that does not hold a useful connection. With type-1 algorithms, connections are still dumped, and holes are still left, but nodes do not know that certain connections in their caches were dumped, until the next requests for these connections arrive. For both a type-1 and a type-2 algorithm we shall think of a connection (v, u) which was evicted by v as still residing in the cache of u until the latter evicts it if it is of type-1 or fills the hole if it is of type-2.

Consider a single-cache replacement policy A. We can easily convert A into a type-1 connection-caching algorithm $\mathcal{A}_1$ in the following way. Algorithm $\mathcal{A}_1$ instructs each node u to handle the caching of the connection at u using the policy A. Thus, for example, $\mathcal{LRU}_1$, the connection-caching algorithm obtained from the caching algorithm LRU behaves in the following way. If a connection has to be evicted from the cache of a node u, then the connection to be evicted would be the connection in u's cache whose last usage occurred before the usage of all the other connections in u's cache. In making this decision, $\mathcal{LRU}_1$ does *not* take into account any information of which of its connections is dumped and which is open. Thus, $\mathcal{LRU}_1$ may evict an open connection from u's cache, even if u's cache contains holes.

We can also convert algorithm A into a type-2 connection caching algorithm $\mathcal{A}_2$ in a natural way. According to $\mathcal{A}_2$ when a connection e has to be added to u's cache, and u's connection cache contains at least one hole, then the connection e fills, i.e., replaces, one of these holes. (It does not matter which hole is used, if there is more than one.) If all the connections in u's cache are open, then algorithm A is used to decide which one of them to evict.

Note that $\mathcal{A}_2$ is not necessarily defined for every single cache algorithm A, as $\mathcal{A}_2$ may encounter situations impossible in a single-cache context. While running $\mathcal{A}_2$ the contents of the cache of v may change by a *dump* followed by a miss that fills the hole. As a result we may reach a state which was impossible to reach in the single cache context. Nevertheless, for many natural paging algorithms A, the algorithm $\mathcal{A}_2$ will be well defined.

We also define a *type-3* distributed connection caching algorithm. A type-3 algorithm is a type-2 algorithm in which v is allowed to exchange some additional information with u while the connection (v, u) is open. Our type-3 algorithms will use a single common bit to u and v while the connection (v, u) is open.

2.2 Connection-caching and standard caching

The connection-caching model is easily seen to be a generalization of the standard caching model. A caching problem with a cache of size k and with n pages can be cast as a connection-caching problem in a star network in which the center of the star has a cache of size k and each one of the n leaves of the star has a cache of size 1.

3 Off-line connection-caching is hard

We first prove that the off-line connection-caching problem is NP-hard.

Theorem 3.1 *Finding the optimal way of serving a sequence σ of connection requests is an NP-hard problem.*

Proof: Let ϕ be a 3SAT instance. Let n be the number of variables appearing in ϕ and let m be the number of clauses in ϕ. We construct a graph $G = (V, E)$ and a sequence σ of connection requests such that σ can be served with $m + n$ hits if and only if the formula ϕ is satisfiable. (If the formula is not satisfiable, the number of hits would necessarily be smaller.)

The sequence σ is the concatenation of three sequences σ_1, σ_2 and σ_3 of connection requests. The sequences σ_1 and σ_3 correspond to the variables of ϕ. While serving them, the connection-caching algorithm is effectively choosing a truth assignment to the variables appearing in ϕ. The sequence σ_2 corresponds to the clauses of ϕ. It is used to check whether the chosen truth assignment satisfies ϕ.

For each variable x appearing in ϕ we introduce a star with three leaves. Label the center x_1 and the leaves x, $\bar{x}$, and x_2. The capacity at the nodes x_1, x, and $\bar{x}$ is 2. The capacity of x_2 is 1. For each variable x appearing in ϕ, the sequence σ_1 contains the three requests

$$(x_1, x) \ , \ (x_1, \bar{x}) \ , \ (x_1, x_2) \ ,$$

and the sequence σ_3 contains the two requests

$$(x_1, x) \ , \ (x_1, \bar{x}) \ .$$

The capacity of x_1 is 2 and the connection (x_1, x_2) is requested after requesting the connections (x_1, x) and $(x_1, \bar{x})$. As a consequence we get that after the network serves the requests in σ_1, the connection cache of x_1 contains at most one of the connections (x_1, x) and $(x_1, \bar{x})$. Caching (x_1, x) corresponds to setting the variable x to false, while caching $(x_1, \bar{x})$ corresponds to setting x to true. With any one of these choices, we score at most one hit while serving the requests (x_1, x) and $(x_1, \bar{x})$ of σ_2, a total of at most n hits for the whole of σ_2.

We now construct the sequence σ_2. For each clause C of ϕ, e.g., $C = x \vee \bar{y} \vee z$ we introduce 2 more nodes, C and $\hat{C}$. Node C has capacity 2 and $\hat{C}$ has capacity 1. For each clause C, the sequence σ_2 contains the following connection requests:

$$\begin{gathered} (C, x) \ , \ (\hat{C}, x) \ , \ (C, \bar{y}) \ , \ (\hat{C}, \bar{y}) \ , \ (C, z) \ , \ (\hat{C}, z) \ , \\ (C, \hat{C}) \ , \\ (C, x) \ , \ (C, \bar{y}) \ , \ (C, z) \ . \end{gathered}$$

As C has capacity 2, we can get at most one hit on each such subsequence. We can get that hit without dumping a connection of the form (x_1, x) if and only if at least one of the literals $x, \bar{y}$ or z is set to true. It is easy to verify that we get $m + n$ hits if and only if the formula is satisfiable. □

A variation on the proof establishes that the connection-caching problem is hard to approximate.

Theorem 3.2 *The problems of minimizing the number of misses, or maximizing the number of hits that occur while servicing a sequence σ of connection requests are APX-hard.*

Proof: The reduction stays exactly as above. We view it this time, however, as a reduction from MAX 3SAT(3), the version of the unweighted MAX 3SAT problem in which each variable appears in at most 3 clauses and clauses are of size at most 3. It is known that MAX 3SAT(3) is APX-hard [24] (or MAX SNP-hard).

Let I be an instance of the MAX 3SAT(3) problem. We may assume that each variable x appears positively in two clauses, and negatively in another, or vice versa. (If all the appearances of x are positive or negative then it is clear which value we should assign to x.)

Let $\sigma = \sigma_1\sigma_2\sigma_3$ be the sequence of connection requests obtained from I using the reduction used in the proof of Theorem 3.1. We claim that it is possible to get at least $n+k$ hits while servicing this sequence if and only if it is possible to satisfy at least k clauses of the MAX 3SAT(3) instance I.

To prove the claim, we have to show that if it is possible to get at least $n+k$ hits, then it is possible to do so while getting n hits for the portion σ_3 of σ. This suffices, as the hits obtained while serving σ_3 determine the required truth assignment.

Suppose now that we do not get hits for (x_1, x) and $(x_1, \bar{x})$. As either x or $\bar{x}$ participates in only one clause, we get that either x or $\bar{x}$ participate in at most one hit. We can give up this hit in exchange for a hit on one of the requests (x_1, x) or $(x_1, \bar{x})$.

This shows that the problem of maximizing the number of hits is APX-hard. As in the instances of the connection caching problem generated by the above reduction, the maximal number of hits and the minimal number of misses are both at least a certain percentage of the number of requests, it follows that the problem of minimizing the number of misses is also APX-hard. $\square$

The off-line connection-caching and generalized connection-caching problems can be solved approximately in polynomial time, when buffer sizes and activation costs of individual connections are small with respect to applicable cache sizes and the overall cost. The approximation is for the hit rate, and is obtained by generalizing the integer-program formulation and randomized-rounding application provided for an ordinary cache in [8].

We note that if the network is fixed, the connection-caching problem can be solved in linear time, in the length of the sequence, using dynamic programming.

4 Basic relations between single-cache algorithms and their type-1 and type-2 derivatives

In this section we begin to explore the behavior of distributed connection-caching algorithms derived from simple caching algorithms such as LRU and LFD. We establish a general relation between the competitive ratios of a single cache algorithm X and its type-1 derivative $\mathcal{X}_1$. As a corollary we obtain that $\mathcal{LFD}_1$ is a 2-approximation algorithm for the offline connection caching problem. We also show that although intuitively $\mathcal{X}_2$ should perform better than $\mathcal{X}_1$ there is no obvious relation that holds between the performance of these two algorithms for every single-cache algorithm X. Before proving these results, we introduce the notion of connection failures.

Recall that if a connection $e = (u, v)$ is open then e resides in both the cache of u and the cache of v. So a miss occurs if e is either not cached by u or not cached by v. We say that a connection request $e = (u, v)$ *fails at* u if $e = (u, v)$ is not in u's cache when the request arrives. Similarly, $e = (u, v)$ fails at v if $e = (u, v)$ is not in v's cache when the request arrives. (Note that we do not care here whether e is open at the time of the request. If the connection e was dumped by v but is still cached at u, then the request e would not fail at u.) Thus, each miss corresponds to either one or two failures.

Recall that $\text{miss}(\mathcal{ALG}, \sigma)$ is the number of misses incurred by $\mathcal{ALG}$ while serving the sequence σ. We similarly define $\text{fail}(\mathcal{ALG}, \sigma)$ to be the total number of failures, encountered by $\mathcal{ALG}$ at all nodes while servicing the sequence σ. Similarly, $\text{miss}(\sigma)$ and $\text{fail}(\sigma)$ are the minimum number of misses, and failures, respectively, needed to service σ. The following lemma immediatly follows from our definitions.

Lemma 4.1 *For any sequence σ of connection requests:*

$$\text{miss}(\sigma) \;\leq\; \text{fail}(\sigma) \;\leq\; 2 \cdot \text{miss}(\sigma) \;.$$

Minimizing the total number of misses and minimizing the total number of failures are two related, but different, problems. We saw in the previous section that minimizing the number of misses is a difficult problem. Minimizing the number of failures, on the other hand, is an easy problem, at least in the off-line setting.

Theorem 4.2 *Let* ALG *be an optimal caching algorithm. Then the connection-caching algorithm $\mathcal{ALG}_1$ services each request sequence of connections with the minimum total number of failures.*

Proof: We let $\text{fail}_v(\mathcal{ALG}, \sigma)$ be the number of failures that occur at the node v while servicing a request sequence σ using the connection caching algorithm $\mathcal{ALG}$. We clearly have:

$$\text{fail}(\mathcal{ALG}, \sigma) = \sum_v \text{fail}_v(\mathcal{ALG}, \sigma) \;.$$

The number of failures that occur at a node v of the network depends only on the caching decisions made at v. Dumps made by other nodes have no effect on the number of failures occurring at v. Thus, to minimize the number of failures at v we should manage v's cache using an optimal single-cache policy. This is exactly what is done by $\mathcal{ALG}_1$. $\square$

When activation costs and buffer sizes of the connections are uniform, LFD (Belady's algorithm) is an optimal caching algorithm ([5], see also [6]) therefore by Theorem 4.2 the connection-caching algorithm $\mathcal{LFD}_1$ minimizes the total number of failures. Thus, as an immediate corollary of Lemma 4.1, we get the following:

Corollary 4.3 *The connection caching algorithm $\mathcal{LFD}_1$ is a 2-approximation algorithm for the connection-caching problem.*

To see that the approximation ration of 2 for $\mathcal{LFD}_1$ is tight consider the network shown in Figure 1. The nodes u and v have caches of size k. All other nodes have caches of size 1. Let $A = \langle a_1, \ldots, a_{k-1} \rangle$, $B = \langle b_1, \ldots, b_{k-1} \rangle$, $C = \langle c_1, \ldots, c_{k-1} \rangle$ and $D = \langle d_1, \ldots, d_{k-1} \rangle$. Consider the sequence of requests $ABCDABCD\ldots$, i.e., the sequence $ABCD$ repeated over and over again. The algorithm $\mathcal{LFD}_1$ behaves in the following way. Initially, the connections $a_1, \ldots, a_{k-1}$ will be placed in the cache of u. When the requests $b_1, \ldots, b_{k-1}$ come, they will be served one at a time using the remaining location in u's cache. Node u prefers keeping the connections $a_, \ldots, a_{k-1}$ in its cache as they will all be requested again before any one of the b_i's. Unfortunately, when the requests $c_1, \ldots, c_{k-1}$ come, the connections $a_1, \ldots, a_{k-1}$ are dumped from u's cache, as the other endpoints of the $a_1, \ldots, a_{k-1}$ can only have one open connection at a time. Exactly the same happens now with v and the connections $c_1, \ldots, c_{k-1}$ and $d_1, \ldots, d_{k-1}$. As a result $\mathcal{LFD}_1$ suffers a miss for each request. It is possible, however to suffer misses only for about half of the requests. The nodes u and v should simply keep in their caches the connections $b_1, \ldots, b_{k-1}$ and $d_1, \ldots, d_{k-1}$ which will never be dumped, and serve the requests $a_1, \ldots, a_{k-1}$ and $c_1, \ldots, c_{k-1}$ using the remaining slot in their cache.

Remark. The nodes in this example have different sizes of caches. We can describe a slightly more complicated version of this example in which all caches have the same size. We now show the following

Another simple consequence of Theorem 4.2 and Lemma 4.1 is the following result:

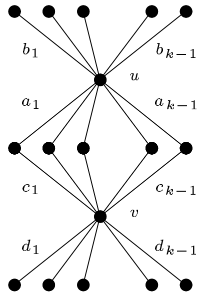


Figure 1: Comparing $\mathcal{LFD}_2$ and $\mathcal{LFD}_1$ with $\mathcal{OPT}$.

Theorem 4.4 *Let A be a c-competitive caching algorithm. Then $\mathcal{A}_1$ is a $(2c)$-competitive algorithm for the connection-caching problem.*

Proof: Let σ be a sequence of connection requests. As $\mathcal{A}_1$ runs A distributively at each node, and as A is c-competitive, we get:

$$\begin{aligned} &\mathrm{miss}(\mathcal{A}_1,\sigma) \;\le\; \mathrm{fail}(\mathcal{A}_1,\sigma) \;=\; \textstyle\sum_v \mathrm{fail}_v(\mathcal{A}_1,\sigma) \\ &\le\; \textstyle\sum_v c\cdot \mathrm{fail}_v(\sigma) \;=\; c\cdot\mathrm{fail}(\sigma) \;\le\; 2c\cdot\mathrm{miss}(\sigma)\ , \end{aligned}$$

as required. □

Two remarks are in order here. First, if A is c-competitive with an additional additive term, then $\mathcal{A}_1$ would also have an additive term and this term would depend on the size of the network. Second, if, as is usually the case, the competitiveness ratio of A depends on the size of the cache, i.e., if A is $c(k)$-competitive for caches of size k, then $\mathcal{A}_1$ would be $(2c(k))$-competitive, where k is the size of the largest cache in the network.

We can easily generalize Theorem 4.4 to the scenario when the connection caching algorithm competes against a weakened adversary.

Theorem 4.5 *Let A be a $c(h,k)$-competitive caching algorithm. Then, $\mathcal{A}_1$ is $2\cdot\max_v c(h(v),k(v))$-competitive.*

LRU (Least Recently Used) is the cache replacement policy which evicts the page that was requested least recently. It is easy to see that LRU, with a cache of size k, is k-competitive ([27], see [6] and [16]). Furthermore, LRU, with a cache of size k, is $k/(k-h+1)$-competitive versus an adversary that uses a cache of size h. As corollaries of Theorems 4.4 and 4.5, we get:

Corollary 4.6 (*i*) *$\mathcal{LRU}_1$ is a $(2k)$-competitive connection-caching algorithm, where k is the size of the largest cache in the network.*
(*ii*) *$\mathcal{LRU}_1$ is $(2\max_v \frac{k(v)}{k(v)-h(v)+1})$-competitive connection-caching algorithm versus an adversary that keeps a cache of size $h(v)$ at each node v of the network.*

As we shall see, tighter bounds on the competitiveness of $\mathcal{LRU}_1$ follow from general results given in Section 6.

Note that Theorems 4.4 and 4.5 apply also to generalized connection caching when activation costs and connection sizes vary. As a consequence, using the $(k/(k-h+1))$-competitive algorithm LANDLORD (LL) of [29], we obtain:

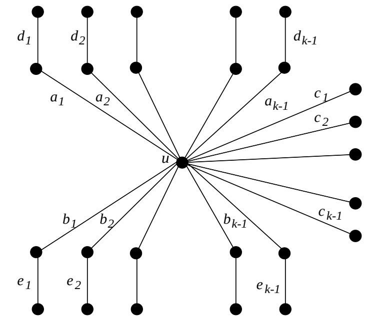


Figure 2: Comparing $\mathcal{LFD}_2$ with $\mathcal{LFD}_1$.

Corollary 4.7 *$\mathcal{LL}_1$ is a $(2\max_v \frac{k(v)}{k(v)-h(v)+1})$-competitive connection-caching algorithm, even in the presence of varying costs and sizes, versus an adversary that keeps a cache of size $h(v)$ at each node v of the network.*

4.1 Type-1 alorithms versus type-2 algorithms

Now that we have established various upper bounds on the performance of several type-1 connection caching algorithms, we ask whether these results also hold for the analogous type-2 connection caching algorithms. The following example shows that there is no general relation between the performance of a type-1 algorithm and the performance of a type-2 algorithm derived from the same single cach algorithm.

Lemma 4.8 *The performances of $\mathcal{LFD}_2$ and $\mathcal{LFD}_1$ are incomparable.*

Proof: We describe a network and corresponding sequences where $\mathcal{LFD}_2$ performs strictly worse than $\mathcal{LFD}_1$, and vice versa. Consider the network depicted in Figure 2. The center node u has cache size k and all other nodes have cache size 1. Let A be the sequence of requests $\langle a_1, a_2, \ldots, a_{k-1}\rangle$. The sequences B, C, D, and E are defined similarly.

Consider the sequence which constitutes from repetitions of $ADEBACEBC$. We show that $\mathcal{LFD}_1$ outperforms $\mathcal{LFD}_2$ on this sequence. First observe that no policy obtains hits on requests for E or B edges. B and E edges are paired such that the B edge in each pair touches the E edge on a node with cache size 1, and requests for edges in each pair are interleaved. Similarly, no policy can obtain hits on D requests or on the second set of A requests in each repetition. Hence, only C requests or requests on the first set of A requests in each repetition can constitute cache hits, but the two exclude each other. Therefore, the optimal policy always prefers to cache C requests, and obtains hits on all (and only on) C requests. Hence, it misses on 7/9 of the requests.

The now consider $\mathcal{LFD}_2$ and claim that it misses on all requests. After the first set of A requests (step 1 of the first repetition), u's cache contains all the A edges. During the second step (D requests), A edges are dumped from u. In the 4th step, B edges are cached. The 5th step (A requests) is all misses and requests are served using a single slot in u's cache (since currently-cached B edges are expected sooner). Similarly, the C requests in the 6th step are served using one slot. The 7th step causes B edges to be dumped, and hence, there are no hits when they are re-inserted in the 8th step. The C edges in the 9th step are served using a single slot, since currently-cached B edges are expected sooner than C. In subsequent repetitions, B edges remain in the cache during steps 1 and

2, until dumped at step 3, and re-inserted in step 4. In steps 5–9, the execution is identical to the first repetition. Thus, $\mathcal{LFD}_2$ misses on all requests.

Consider the policy $\mathcal{LFD}_1$. A edges are inserted to u's cache in the first step of the first repetition. Since u is unaware of the dumps caused in the second set, A edges remain in its cache until requested in step 5. A edges are evicted for C edges in step 6 (since C is expected sooner than A). C remains in u's cache through step 9, where it results in hits. In subsequent repetitions, C is replaced with A in the first step, and the process proceeds just like the first repetition. Note that $\mathcal{LFD}_1$ obtains hits on the requests for C in the 9th step of each iteration. Hence, it misses on 8/9 of the requests and outperforms $\mathcal{LFD}_2$ that misses on all requests.

We now show that for the repeated sequence ADB, the policy $\mathcal{LFD}_2$ is optimal and outperforms $\mathcal{LFD}_1$. Observe that no policy can obtain hits on A or D since they are interleaving. Hence, the optimal policy caches B edges and obtains hits on all B requests (except on the first repetition). Hence, the miss rate of the optimal policy approaches 2/3.

Consider the policy $\mathcal{LFD}_2$. In the first repetition, A is inserted to u's cache in the first step and dumped in the second step. Hence, B is inserted in the 3rd step. In the next iteration, A is not inserted in the first step, since B is expected before A. Therefore, B remains in the cache until the 3rd step, where it constitutes cache hits. This is repeated in subsequent iterations. Hence, in this example, $\mathcal{LFD}_2$ is optimal.

Consider the policy $\mathcal{LFD}_1$. In the first step of the first repetition, A is inserted into u's cache. The A edges are dumped in the second step, but remain cached by u. In the 3rd step, B edges are served using a single slot, since A requests are expected sooner than B requests. Hence, in the beginning of the second iteration (and subsequent iterations), A edges are cached by u, but are actually dumped, and hence, do not constitute cache hits. Hence, $\mathcal{LFD}_1$ misses on all requests and is outperformed by $\mathcal{LFD}_2$. □

For some algorithms however we can prove that on every request sequence the type-2 algorithm outperforms its type-1 analog. For example the following lemma estalish this fact for $\mathcal{LRU}_1$ versus $\mathcal{LRU}_2$.

Lemma 4.9 *For any sequence σ of connection requests we have* $\mathrm{miss}(\mathcal{LRU}_2, \sigma) \leq \mathrm{miss}(\mathcal{LRU}_1, \sigma)$.

Proof: We establish the following two invariants. The proof is an immediate consequence of the first invariant.

1. The set of open connections held by $\mathcal{LRU}_1$, i.e., connections that are cached by both their end-points, is always a subset of the open connections held by $\mathcal{LRU}_2$.
2. If (u, v) is cached by both $\mathcal{LRU}_1$ and $\mathcal{LRU}_2$ then every connection cached by $\mathcal{LRU}_2$ which was requested after (u, v) is also cached by $\mathcal{LRU}_1$.

We prove the invariants by an induction on the request sequence. Suppose the invariants hold before a request to the edge $e = (u, v)$. We next prove that the invariants also hold after the request to $e = (u, v)$. Invariant (1) implies that the request cannot be a hit for $\mathcal{LRU}_1$ and a miss for $\mathcal{LRU}_2$. If the request is a hit for both $\mathcal{LRU}_2$ and $\mathcal{LRU}_1$ or a hit for $\mathcal{LRU}_2$ but a miss for $\mathcal{LRU}_1$ then it is easy to see that the invariants still hold after the request.

Assume that the request is a miss for both $\mathcal{LRU}_2$ and $\mathcal{LRU}_1$. If $\mathcal{LRU}_2$ does not evict any connection or evicts connections which are not in the cache of $\mathcal{LRU}_1$ then the invariants clearly hold after the request. Let $e' = (u, w)$ be a connection evicted by $\mathcal{LRU}_2$ which is also cached by $\mathcal{LRU}_1$. By the definition of $\mathcal{LRU}_2$ all the other connections in the cache of u were requested after (u, w). So by invariant (2) they are all cached by $\mathcal{LRU}_1$ prior to the request. By the definition of $\mathcal{LRU}_1$ it also evicts e' while serving e. It follows that the invariants hold after the request. (Same argument holds for an eviction of $\mathcal{LRU}_2$ at v.) □

As a consequence, we get that the results of Corollary 4.6 apply to $\mathcal{LRU}_2$ as well. A tighter bound on the competitive ratio of $\mathcal{LRU}_2$ is given in Section 6.

4.2 Lower bounds on competitive ratio

As mentioned in Section 2.2, connection-caching is a generalization of the classical single-cache paging model. It is known (see [6] and [16]) that the competitive ratio of any *deterministic* caching algorithm is at least k, where k is the size of the cache, and that the competitive ratio of any *randomized* caching algorithm is at least $H_k = \sum_{i=1}^{k} \frac{1}{k} > \ln k$ ([13], see also [6] and [16]). The same lower bounds apply, therefore, also to the connection-caching problem.

5 The performance of $\mathcal{LFD}_2$

Corollary 4.3 says that $\mathcal{LFD}_1$ is a 2-approximation algorithm for the connection-caching problem. What about $\mathcal{LFD}_2$? We described examples that show that there are sequences on which $\mathcal{LFD}_2$ behaves better $\mathcal{LFD}_1$, while there are other sequences on which $\mathcal{LFD}_1$ behaves better than $\mathcal{LFD}_2$. Nevertheless, we can show that $\mathcal{LFD}_2$ is also a 2-approximation algorithm for the connection-caching problem.

We let $\mathrm{dump}_v(\mathcal{A}, \sigma)$ be the number of connections that are dumped from the cache of v, by v's neighbors, while servicing the request sequence σ using the connection-caching algorithm $\mathcal{A}$. We let $\mathrm{dump}(\mathcal{A}, \sigma) = \sum_v \mathrm{dump}_v(\mathcal{A}, \sigma)$ be the total number of dumps that occur while servicing σ using $\mathcal{A}$. Recall that $\mathrm{miss}(\mathcal{A}, \sigma)$ and $\mathrm{fail}(\mathcal{A}, \sigma)$ are the total number of misses and failures that occur while servicing σ using $\mathcal{A}$. We need the following simple Lemma:

Lemma 5.1 *For any connection-caching algorithm $\mathcal{A}$ and any request sequence σ we have $\mathrm{dump}(\mathcal{A}, \sigma) \leq \mathrm{miss}(\mathcal{A}, \sigma)$.*

Proof: Charge a half of each dump to the miss that caused it and the second half to the miss that would fill the hole caused by the dump. The second half of the last dump that occurs in each slot of a cache is charged to the miss that caused this slot to be used for the first time. In this way, each miss is charged for at most one dump. □

We now claim:

Theorem 5.2 *$\mathcal{LFD}_2$ is a 2-approximation algorithm for the connection-caching problem.*

Proof: We let $\mathcal{A}$ denote any connection-caching algorithm. We let $\mathcal{B}$ stand for $\mathcal{LFD}_2$. We show that for every sequence σ and for every node v of the network we have:

$$\mathrm{miss}_v(\mathcal{B}, \sigma) \ \leq \ \mathrm{fail}_v(\mathcal{A}, \sigma) + \mathrm{dump}_v(\mathcal{B}, \sigma) \ . \qquad (*)$$

Here $\mathrm{miss}_v(\mathcal{B}, \sigma)$ is the total number of misses in requests that touch v incurred by $\mathcal{B}$ while servicing σ. By summing this inequality over all nodes we get

$$2 \cdot \mathrm{miss}(\mathcal{B}, \sigma) \ \leq \ \mathrm{fail}(\mathcal{A}, \sigma) + \mathrm{dump}(\mathcal{B}, \sigma) \ .$$

By Lemma 5.1 $\text{dump}(\mathcal{B},\sigma) \le \text{miss}(\mathcal{B},\sigma)$ and therefore $\text{miss}(\mathcal{B},\sigma) \le \text{fail}(\mathcal{A},\sigma)$. As $\text{fail}(\mathcal{A},\sigma) \le 2 \cdot \text{miss}(\mathcal{A},\sigma)$, we get that

$$\text{miss}(\mathcal{B},\sigma) \le 2 \cdot \text{miss}(\mathcal{A},\sigma) .$$

As this holds for any connection-caching algorithm, including the optimal one, it follows that $\mathcal{B} = \mathcal{LFD}_2$ is a 2-approximation algorithm.

Before proving $(*)$ we introduce some new notation. For brevity, we omit σ from now on from the expressions $\text{miss}_v(\mathcal{B},\sigma)$, $\text{dump}_v(\mathcal{B},\sigma)$ and $\text{fail}_v(\mathcal{A},\sigma)$. We let $\text{miss}_v(\mathcal{B})$, $\text{dump}_v(\mathcal{B})$ and $\text{fail}_v(\mathcal{A})$ be the number of misses, dumps and failures seen *so far* at node v. We also use $\text{cache}_v(\mathcal{A})$ and $\text{cache}_v(\mathcal{B})$ to denote the states of v's cache under $\mathcal{A}$ and under $\mathcal{B}$ after servicing a portion of the sequence σ. (To be more formal we would have to add an index of time to all these notations. We avoid that as this would make the notations very cumbersome.)

Consider a location a in $\text{cache}_v(\mathcal{A})$ and a location b in $\text{cache}_v(\mathcal{B})$. Let $e(a)$ and $e(b)$ be the connections stored in a and b. (The locations a and b may be empty, in which case the connections $e(a)$ and $e(b)$ are dummy connections.) We say that b *dominates* a, and write $b \ge a$, if either a is empty, or if $e(b)$ is *active* and either $e(a) = e(b)$, or the next request for $e(b)$ will occur before the next request for $e(a)$. (Note that this definition treats the connections in $\text{cache}_v(\mathcal{A})$ and $\text{cache}_v(\mathcal{B})$ in a different manner. If a is not empty, then for b to dominate a, the connection $e(b)$ must be active even if the connection $e(a)$ is not active.)

We define the *advantage* that $\text{cache}_v(\mathcal{A})$ has over $\text{cache}_v(\mathcal{B})$, denoted $\text{adv}_v(\mathcal{A},\mathcal{B})$, as the minimal $c \ge 0$ such that there is a matching between the locations of $\text{cache}_v(\mathcal{A})$ and those of $\text{cache}_v(\mathcal{B})$ such that at most c of the locations in $\text{cache}_v(\mathcal{A})$ are *not* dominated by their mates in $\text{cache}_v(\mathcal{B})$.

We now prove the following strengthening of $(*)$. At every node of the network, and at any time during the servicing of σ we have:

$$\text{miss}_v(\mathcal{B}) + \text{adv}_v(\mathcal{A},\mathcal{B}) \;\le\; \text{fail}_v(\mathcal{A}) + \text{dump}_v(\mathcal{B}) . \qquad (**)$$

This is clearly a strengthening of $(*)$ as we always have $\text{adv}_v(\mathcal{A},\mathcal{B}) \ge 0$.

We prove $(**)$ by induction. The claim clearly holds before servicing the first request. Suppose that $e = (v,w)$ is the next request to be serviced by $\mathcal{A}$ and $\mathcal{B}$.

Servicing $e = (v,w)$ by $\mathcal{B}$ may affect the states of at most four caches in the network. The caches $\text{cache}_v(\mathcal{B})$ and $\text{cache}_w(\mathcal{B})$ can clearly change as a result of this request. Furthermore, if e results with a miss, then v may be forced to dump a connection (v,v'), thereby changing $\text{cache}_{v'}(\mathcal{B})$, and w may be forced to dump a connection (w,w'), thereby changing $\text{cache}_{w'}(\mathcal{B})$ (v' and w' are not necessarily distinct).

Similarly, the servicing of $e = (v,w)$ by $\mathcal{A}$ may clearly change $\text{cache}_v(\mathcal{A})$ and $\text{cache}_w(\mathcal{A})$. Again, v and w may be forced to dump connections (v,v'') and (w,w''). But, as we do not distinguish between active and non-active connections in $\text{cache}_{v''}(\mathcal{A})$ and $\text{cache}_{w''}(\mathcal{A})$, we can consider these two caches to be unchanged.

We have to show that $(**)$ continues to hold, after the servicing of e by $\mathcal{A}$ and $\mathcal{B}$, at all the nodes of the network. If $u \notin \{v,w,v',w'\}$, then $(**)$ clearly continues to hold at u. Consider now the situation at v'; $\text{miss}_{v'}(\mathcal{B})$ and $\text{fail}_v(\mathcal{A})$ remain unchanged while $\text{adv}_{v'}(\mathcal{A},\mathcal{B})$ and $\text{dump}_{v'}(\mathcal{B})$ may increase. However, as each connection dumped from $\text{cache}_{v'}(\mathcal{B})$ increases $\text{adv}_{v'}(\mathcal{A},\mathcal{B})$ by at most one, as at most one domination is destroyed, $(**)$ continues to hold. The same applies to w''.

All that remains, therefore, is to show that $(**)$ continues to hold at v and w. We consider v. The same argument would apply of course also to w. We consider four different cases:

- **Case 1.** $\mathcal{B}$ gets a hit, $\mathcal{A}$ does not fail at v.
  We may assume that (e,e) is one of the good pairs is the matching that determines the value of $\mathrm{adv}_v(\mathcal{A},\mathcal{B})$. The pair (e,e) clearly remains good after servicing c. Thus, none of the quantities $\mathrm{miss}_v(\mathcal{B})$, $\mathrm{adv}_v(\mathcal{A},\mathcal{B})$, $\mathrm{fail}_v(\mathcal{A})$ and $\mathrm{dump}_v(\mathcal{B})$ change as a result of the request and $(**)$ continues to hold.

- **Case 2.** $\mathcal{B}$ gets a miss, $\mathcal{A}$ fails at v.
  The quantities $\mathrm{miss}_v(\mathcal{B})$ and $\mathrm{fail}_v(\mathcal{A})$ both increase by one while $\mathrm{dump}_v(\mathcal{B})$ stays unchanged. We have to show that $\mathrm{adv}_v(\mathcal{A},\mathcal{B})$ does not increase. The request for e forces $\mathcal{A}$ and $\mathcal{B}$ to add e to $\mathrm{cache}_v(\mathcal{A})$ and $\mathrm{cache}_v(\mathcal{B})$. Let a and b be the locations in $\mathrm{cache}_v(\mathcal{A})$ and in $\mathrm{cache}_v(\mathcal{B})$ in which e is placed. Let a' and b' be the locations matched with a and b. If at most one of these pairs was good, then $\mathrm{adv}_v(\mathcal{A},\mathcal{B})$ does not increase as the dominated pair that might have been lost can now be replaced by the dominated pair $b \geq a$. Assume, therefore, that before the request we had $a' \geq a$ and $b \geq b'$. As e was placed in b, and as $\mathcal{B}$ evicts the connection with the largest future distance, the next request of $e(b)$ must be after the next request of $e(a')$. Thus, we also have $a' \geq b'$. This would still hold after the request. After the request, we also have $b \geq a$, as $e(b) = e(a) = e$ and e is active. Thus, the number of dominated pairs does not decrease and $\mathrm{adv}_v(\mathcal{A},\mathcal{B})$ does not increase.

- **Case 3.** $\mathcal{B}$ gets a miss, $\mathcal{A}$ does not fail at v.
  We have to show that $\mathrm{adv}_v(\mathcal{A},\mathcal{B})$ decreases by at least one. As e is already cached at $\mathrm{cache}_v(\mathcal{A})$ before the request for e, algorithm $\mathcal{A}$ has no reason to evict a connection from $\mathrm{cache}_v(\mathcal{A})$. Let a be location of e in $\mathrm{cache}_v(\mathcal{A})$. Algorithm $\mathcal{B}$, on the other hand, may be forced to evict a connection. Let b be the location in $\mathrm{cache}_v(\mathcal{B})$ into which e is placed. Before the request, $e = e(a)$ is the request in $\mathrm{cache}_v(\mathcal{A})$ with the nearest next request time, and $e(b)$ is either inactive or it has the latest request time among the active connections in $\mathrm{cache}_v(\mathcal{B})$. Let a' and b' be the locations matched with a and b. It is not difficult to see that the value of $\mathrm{adv}_v(\mathcal{A},\mathcal{B})$, before the request for e, can be realized using a matching in which $a' \not\geq a$ and $b \not\geq b'$. After the request for e we have $b \geq a$ so $\mathrm{adv}_v(\mathcal{A},\mathcal{B})$ decreases by at least one.

- **Case 4.** $\mathcal{B}$ gets a hit, $\mathcal{A}$ fails at v.
  We have to show that $\mathrm{adv}_v(\mathcal{A},\mathcal{B})$ increases by at most one. Let b be the location of $\mathrm{cache}_v(\mathcal{B})$ containing e. Let a be the location of $\mathrm{cache}_v(\mathcal{A})$ into which e is placed. Let a' and b' be the mates of a and b. After the request we can match a with b and the number of good pairs decreases by at most one.

This completes the proof of $(**)$ and of the Theorem. □

Note that the example in Section 4 which showed that the approximation ratio of 2 is tight for $\mathcal{LFD}_1$ also shows that the approximation ratio of 2 is tight for $\mathcal{LFD}_2$. Both $\mathcal{LFD}_1$ and $\mathcal{LFD}_2$ act the same on that example.

6 Deterministic marking algorithms

A single-cache marking algorithm acts as follows. Each page in the cache has a *mark bit* associated with it. When the bit is set we say that the page is *marked*. When this bit is cleared, we say that the page is *unmarked*. When a page is inserted into the cache it is marked; only unmarked pages can be evicted; when all pages are marked and an eviction is due, all current pages get unmarked. When an unmarked page is requested and is currently in the cache, it *may* be marked by the algorithm.[2] LRU belongs to the class of marking algorithms where a page is always marked when requested. The policy FWF (Flush When Full) evicts pages as soon as they are unmarked. FWF is the weakest marking algorithm in the sense that a page is in the cache of FWF only if it would have been cached under any other policy in that class. FIFO is a marking algorithm where a cached page is never marked when requested.

We consider three types of connection-caching marking algorithms, depending on the type of communication, and on whether local or global mark bits are used. In marking algorithms of type-1 and type-2 each node locally marks the connections in its cache. Thus, a connection (u, v) may be marked in cache(u), the cache of u, but unmarked in cache(v), the cache of v, or vice versa. When a connection (u, v) is evicted by v, it is unmarked at v, but it may be marked or unmarked at u. Following the definitions of Section 2, type-1 and type-2 marking algorithms differ by whether a node is notified about dumped connections or not.

In marking algorithms of type-3, the mark bits are global. Each open connection (u, v) has just a single mark bit, it is marked at cache(u) if and only if it is marked at cache(v). When a connection (u, v) is brought into the caches of u and v, it is marked. At some future stage, either u or v may be forced to unmark the connection (u, v). The other node is immediately notified that (u, v) was unmarked.

Of particular interest is the type-3 marking algorithm $\mathcal{FWF}_3$, which evicts connections as soon as they are unmarked. Algorithm $\mathcal{FWF}_3$ does not require additional communication for unmarkings, since they are communicated by connection closures. In fact, $\mathcal{FWF}_3$ is the same as its type-2 version, $\mathcal{FWF}_2$, which evicts a connection as soon as one of the endpoints unmarks it.

We shall need the following lemma that shows that the difference between the number of evictions of a particular algorithm to the number of misses it generates is bounded by a constant. We omit the proof which is easy by induction on the length of the sequence.

Lemma 6.1 *Let* miss(σ) *be the number of misses,* evict(σ) *be the number of evictions, and* hole(σ) *be the number of holes remaining in all the caches of the network after servicing a sequence σ of connection requests using some connection caching algorithm. Also let $K = \frac{1}{2}\sum_{v \in V} k(v)$. Then*

$$\text{miss}(\sigma) \;=\; \text{evict}(\sigma) + (K - \frac{1}{2} \cdot \text{hole}(\sigma)) \;.$$

The main theorem of this section is the following.

Theorem 6.2 *Any deterministic type-3 marking algorithm is k-competitive. (Recall that k is the size of the largest cache in the network.)*

Proof: Let $\mathcal{ALG}$ be a type-3 marking algorithm. Let $\mathcal{OPT}$ be an optimal off-line connection caching algorithm. We show that the number of connections unmarked by $\mathcal{ALG}$ is at most k times the number of misses of $\mathcal{OPT}$. The claim of the Theorem follows as each connection is unmarked before it is evicted, and as by Lemma 6.1, the number of misses of $\mathcal{ALG}$ is at most the number of evictions of $\mathcal{ALG}$, plus an additive constant.

Suppose node u is about to unmark all the connections in its cache as a result of a request for the connection (u, v). Consider the situation *after* $\mathcal{OPT}$ services this request but *before* $\mathcal{ALG}$ services it. At that moment, (u, v) resides in cache$_{\mathcal{OPT}}(u)$, the cache of $\mathcal{OPT}$ at u, but not in cache$_{\mathcal{ALG}}(u)$,

[2] Our definition of marking algorithms is somewhat more general than previous definitions, since we do not require a requested unmarked cached page to be marked.

the cache of $\mathcal{ALG}$ at u. As the caches of $\mathcal{OPT}$ and $\mathcal{ALG}$ at u are of the same size, there must be at least one connection (u, w) in $\text{cache}_{\mathcal{ALG}}(u)$ which is not in $\text{cache}_{\mathcal{OPT}}(u)$. We charge the unmarkings of the edges of u to the *most recent* request r that made $\mathcal{OPT}$ insert (u, w) into its cache. (As (u, w) is currently in $\text{cache}_{\mathcal{ALG}}(u)$, there must have been at least one request for this connection, and therefore (u, w) must have been inserted into $\text{cache}_{\mathcal{OPT}}(u)$ at some stage.)

Note that the request (u, v) that causes u to unmark its connections, may also cause v to unmark its connections. However, as the unmarkings at u are charged to a request for a connection (u, w), where $w \neq v$, and the unmarkings at v are charged to a request for a connection (v, w'), where $w' \neq u$, these unmarkings are charged to different misses of $\mathcal{OPT}$.

It remains to show that each miss of $\mathcal{OPT}$ is charged at most once with the unmarking of k connections by $\mathcal{ALG}$. We have already seen that a miss of $\mathcal{OPT}$ cannot be charged for unmarkings by two nodes that take place at the same time. We now show that a miss of $\mathcal{OPT}$ is also not charged for unmarkings by two nodes that take place at different times.

Let r be a request for connection (u, w) that causes a miss of $\mathcal{OPT}$. Let t_0 be the time in which r occurs. Let t_1, where $t_0 < t_1$, be the time of the first unmarking, by u or by w, for which r is charged. (If r is never charged then we are done.) After servicing the request at time t_1, the connection (u, w) is either out of $\text{cache}_{\mathcal{ALG}}(u)$ and $\text{cache}_{\mathcal{ALG}}(w)$, or is still present in these caches but is unmarked. The connection (u, w) is out of $\text{cache}_{\mathcal{OPT}}(u)$ and $\text{cache}_{\mathcal{OPT}}(w)$. Suppose, for contradiction, that r is also charged, by u or by w, at some later time t_2, where $t_1 < t_2$. At time t_2, the connection (u, w) must be in $\text{cache}_{\mathcal{ALG}}(u)$ and in $\text{cache}_{\mathcal{ALG}}(w)$ and it must be marked. As a connection is only marked as a result of a request, it follows that at least one request for (u, w) occurred between times t_1 and t_2. Let r' be the first such request. Note that $r' \neq r$. As (u, w) is out of $\text{cache}_{\mathcal{OPT}}(u)$ and $\text{cache}_{\mathcal{OPT}}(w)$ at time t_1, and as r' is the first request for (u, w) after time t_1, it follows that request r' is a miss for $\mathcal{OPT}$. Thus, at time t_2, the request r is not the most recent request that made $\mathcal{OPT}$ insert (u, w) into its cache. It follows that r will never be charged again. □

Theorem 6.3 *Any deterministic type-1 or type-2 marking algorithm is* $(2k - 1)$*-competitive.*

Proof: The proof is a modification of the proof of Theorem 6.2. Let $\mathcal{ALG}$ be a type-1 or type-2 marking algorithm. Let $\mathcal{OPT}$ be an optimal off-line connection caching algorithm. For a connection $e = (u, v)$, we let $\text{mark}_u(e)$ and $\text{mark}_v(e)$ be the mark bits of e in the caches of u and v, as assigned by ALG. If e is not in the cache of u, then we let $\text{mark}_u(e) = 0$. Similarly, if e is not in the cache of v, then $\text{mark}_v(e) = 0$. (Note that in type-1 algorithms, e may be in $\text{cache}_{\text{ALG}}(u)$ but not in $\text{cache}_{\text{ALG}}(v)$, or vice versa.) For the purpose of the proof, we also associate with each connection $e = (u, v)$ a *global mark bit* $\text{mark}(e) = \text{mark}_u(e) \wedge \text{mark}_v(e)$, i.e., the logical and of $\text{mark}_u(e)$ and $\text{mark}_v(e)$. An unmarking of e, by u or by v, that changes the global mark $\text{mark}(e)$ from 1 to 0, is said to be a *global unmarking* of e. Note that not every unmarking of e is a global unmarking of e, as the global mark of e may have been 0 before e is unmarked by, say, u. It is still true, however, that for any sequence of requests, the number of evictions carried out by $\mathcal{ALG}$ is at most the number of global unmarkings performed by $\mathcal{ALG}$. To prove the Theorem, we show that the number of global unmarkings performed by $\mathcal{ALG}$ is at most $2k - 1$ times the number of misses of $\mathcal{OPT}$.

Suppose node u is about to unmark all the connections in its cache as a result of a request for the connection (u, v). Consider the situation *after* $\mathcal{OPT}$ services this request but *before* $\mathcal{ALG}$ services it. At that moment, (u, v) resides in $\text{cache}_{\mathcal{OPT}}(u)$, the cache of $\mathcal{OPT}$ at u, but not in $\text{cache}_{\mathcal{ALG}}(u)$, the cache of $\mathcal{ALG}$ at u. As the caches of $\mathcal{OPT}$ and $\mathcal{ALG}$ at u are of the same size, there must be at least one connection (u, w) in $\text{cache}_{\mathcal{ALG}}(u)$ which is not in $\text{cache}_{\mathcal{OPT}}(u)$. Some of the markings going to be made by u are global unmarkings while some may be not. We charge the *global unmarkings* of the edges of u to the *most recent* request r that made $\mathcal{OPT}$ insert (u,w) into its cache. (As (u,w) is currently in cache$_{\mathcal{ALG}}(u)$, there must have been at least one request for this connection, and therefore (u,w) must have been inserted into cache$_{\mathcal{OPT}}(u)$ at some stage.)

As in the proof of Theorem 6.2, it is easy to see that a miss of $\mathcal{OPT}$ cannot be charged for more than k global unmarkings that take place at the same time unit. We show further, that each miss of $\mathcal{OPT}$ can be charged for at most k global unmarkings that take place at some time t_1 and then for at most $k-1$ additional global unmarkings that take place at some time t_2, for some $t_1 < t_2$, and no more. The claim of the Theorem would then follow.

Let r be a request for connection (u,w) that causes a miss of $\mathcal{OPT}$. Let t_0 be the time in which r occurs. Let t_1, where $t_0 < t_1$, be the first time unit in which global unmarking are charged to r by either u or w. For concreteness, let us assume that these at most k global unmarkings were charged by u. It is easy to check, as in the proof of Theorem 6.2, that no additional global unmarkings performed by u will ever be charged to r. However, r may be charged for additional global unmarkings performed by w. Let t_2, where $t_1 < t_2$, be the first time unit in which global unmarkings initiated by w are charged to r. After servicing the request at time t_1, the connection (u,w) is unmarked at u. It is easy to see that no request for (u,w) can be made between t_1 and t_2. Thus, at time t_2, the connection (u,w) is either still unmarked at u, or is out of cache$_{\mathcal{ALG}}(u)$. Thus, the unmarking of (u,w) by w at time t_2 is *not* a global unmarking. It follows that at most $k-1$ global unmarkings are charged by w to r at time t_2. Again, it is easy to verify that r cannot be charged by additional global after time t_2. This completes the proof. $\square$

We can easily extend Theorems 6.2 and 6.3 to obtain bounds on the competitive ratios achieved by marking algorithms against off-line connection caching algorithms that use smaller caches.

Theorem 6.4 *Any deterministic type-3 marking algorithm is* $(\max_u \frac{k(u)}{k(u)-h(u)+1})$*-competitive against an off-line connection caching algorithm which at every node u of the network is only allowed to use a cache of size $h(u) \leq k(u)$.*

Proof: The proof is a simple extension of the proof of Theorem 6.2. Suppose again that u is about to unmark the connections in its cache as a result for a request for a connection (u,v). Consider the situation *after* $\mathcal{OPT}$ services this request but *before* $\mathcal{ALG}$ services it. At that moment, (u,v) is in cache$_{\mathcal{OPT}}(u)$ but not in cache$_{\mathcal{ALG}}(u)$. As cache$_{\mathcal{ALG}}(u)$ is of size $k(u)$, while cache$_{\mathcal{OPT}}(u)$ is only of size $h(u)$, it follows that there are $d = k(u)-h(u)+1$ connections $(u,w_1),(u,w_2),\ldots,(u,w_d)$ that are in cache$_{\mathcal{ALG}}(u)$ but not in cache$_{\mathcal{OPT}}(u)$. Let $r_1,r_2,\ldots,r_d$ be the most recent misses of $\mathcal{OPT}$ on the connections $(u,w_1),\ldots,(u,w_d)$. We charge each one of these misses of $\mathcal{OPT}$ by $k(u)/d = \frac{k(u)}{k(u)-h(u)+1}$ unmarkings of $\mathcal{ALG}$. As in the proof of Theorem 6.2, it is easy to verify that a miss of $\mathcal{OPT}$ is charged on at most one such occasion. $\square$

Using very similar arguments, we also get:

Theorem 6.5 *Any deterministic type-1 or type-2 marking algorithm is*

$$(\max_{(u,v)\in E}[\frac{k(u)}{k(u)-h(u)+1} + \frac{k(v)-1}{k(v)-h(v)+1}]) - competitive$$

against an off-line connection caching algorithm which at every node u of the network is only allowed to use a cache of size $h(u) \leq k(u)$.

Note that even if $h(u) = k(u)$ for every $u \in V$, Theorem 6.5 is a slight strengthening of Theorem 6.3, as it states that any type-1 or type-2 marking algorithm is in fact $(\max_{(u,v)\in E}[k(u)+k(v)-$

1])-competitive. In particular, a type-1 or type-2 marking algorithm may be $(2k-1)$-competitive, and not better, only on networks in which there are two adjacent nodes with caches of size k.

6.1 Lower bounds

Recall that the standard caching model is a special case of the connection caching model (see Section 2.2). Therefore, Theorem 6.2 is tight since we know that in the single cache setting no deterministic algorithm can have competitive ratio smaller than k. Theorem 6.3 is tight in the sense that we next show a type-1 and a type-2 marking algorithms whose competitive ratio is exactly $2k-1$. However, we do not have a general lower bound that applies to all type-1 and type-2 marking algorithms.

We construct a network and a request sequences on which $\mathcal{LRU}_2$, which is a type-2 marking algorithm, incurs $2k-1$ times the number of misses incurred by another (the optimal) algorithm. Combining this example with Theorem 4.9 we obtain that $\mathcal{LRU}_1$ must have miss rate of $2k-1$ times the optimal on this sequence as well.

We start with the $k=2$ case on a graph with 6 nodes[3] and show a sequence where $\mathcal{LRU}_2$ incurs 3 times as many misses as the optimal off-line algorithm $\mathcal{OPT}$.

The initial cache configuration is identical for both $\mathcal{LRU}_2$ and $\mathcal{OPT}$, and consists of two triangles with interleaving relative ages of their edges. Note that any initial cache configuration can be obtained with an appropriate sequence of requests.

We describe a sequence of 12 requests on which $\mathcal{OPT}$ incurs 4 misses while $\mathcal{LRU}_2$ incurs 12 misses. The sequence brings us back to a configuration isomorphic to the starting configuration. The sequence is illustrated in Table 2. Edges in $\mathcal{LRU}_2$'s cache are labeled with their birth time-stamp. The initial cache configuration consists of two disjoint "triangles," (a, b, f) and (c, d, e), Both algorithms incur misses on the first two requests: $\mathcal{LRU}_2$ evicts 2 edges on each request (4 in total) while $\mathcal{OPT}$ evicts 2 edges on the first request and none on the second (2 in total). Consequently, there are 4 edges that are cached by $\mathcal{OPT}$ and not by $\mathcal{LRU}_2$. These edges are then requested (4 misses for $\mathcal{LRU}_2$ and 4 hits for $\mathcal{OPT}$), bringing both caches to an identical 6-cycle configuration (a, e, c, d, b, f). To allow for a clearer illustration of requests 6–12 in Table 2, we exchanged the positions of nodes b and e. Another sequence of 6 requests, where both algorithms incur misses on the first 2 requests, and only $\mathcal{LRU}_2$ incurs misses on the last 4, bring both caches back to a configuration isomorphic to the starting configuration.

The state of $\mathcal{LRU}_2$'s cache is captured by the current cache configuration and a partial order providing the relative "age" of all edges with common endpoints. The following lemma was not needed for the $k=2$ case but is used for the generalization.

Lemma 6.6 *If $\mathcal{LRU}_2$ and $\mathcal{OPT}$ are in the same cache configuration, the relative ages of the edges can be re-ordered arbitrarily without incurring any misses by either algorithm.*

Proof: We issue a series of requests that are hits for both $\mathcal{LRU}_2$ and $\mathcal{OPT}$. We request all edges one by one, starting from what we want to be the "oldest" edge to what we want to become the "newest" edge. □

We now generalize the construction above to $k>2$. The construction relies on the $k=2$ case and contains the hexagon subgraph and the 12-requests sequence (broken to 4 subsequences with new requests inserted).

In addition, we use $k-2$ sets of new edges (possible requests) $R_1, \ldots, R_{k-2}$. The R_i's are mutually disjoint. Each R_i contains 6 edges, and there is one edge incident to a hexagon node $(a$–$f)$

[3] This is the smallest number of nodes for which we can obtain a competitive ratio of 3 with $k=2$.

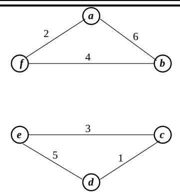
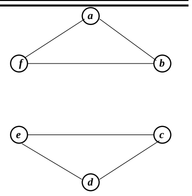
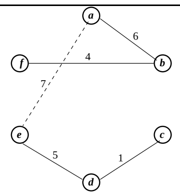
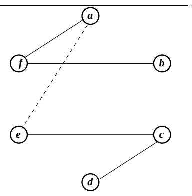
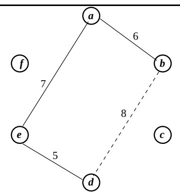
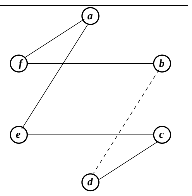
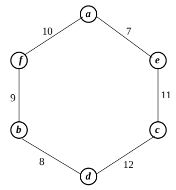
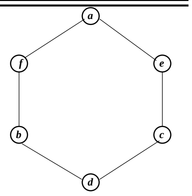
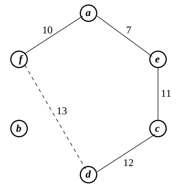
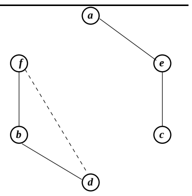
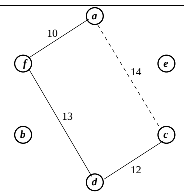
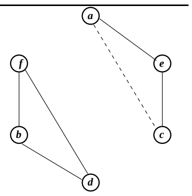
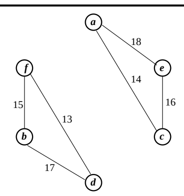
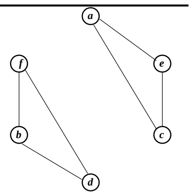
request		$\mathcal{LRU}_2$			$\mathcal{OPT}$		
#	edge	m/h	evict	state	m/h	evict	state
0							
1	(a, e)	m	(a, f) (e, c)		m	(a, b) (e, d)	
2	(b, d)	m	(c, d) (b, f)		m		
3	(b, f)	m	(a, b)		h		
4	(a, f)	m			h		
5	(c, e)	m	(d, e)		h		
6	(c, d)	m			h		
7	(d, f)	m	(b, d) (b, f)		m	(c, d) (a, f)	
8	(a, c)	m	(a, e) (c, e)		m		
9	(b, f)	m	(a, f)		h		
10	(c, e)	m	(c, d)		h		
11	(b, d)	m			h		
12	(a, e)	m			h		

Table 2: Set of 12 requests for $k = 2$ on which $\mathcal{LRU}_2$ incurs 12 misses while $\mathcal{OPT}$ incurs only 4 misses. The end configuration is isomorphic to the starting configuration.

	request		$\mathcal{LRU}_2$		$\mathcal{OPT}$
#	edge	m/h	evicted	m/h	evicted
1	(a,e)	m	$R_1(a)$, $R_1(e)$	m	(a,b) (e,d)
2	(b,d)	m	$R_1(b)$, $R_1(d)$	m	
3–6	$R_1(a), R_1(b), R_1(d), R_1(e)$	4m	$R_2(a), R_2(b), R_2(d), R_2(e)$	4h	
7–10	$R_2(a), R_2(b), R_2(d), R_2(e)$	4m	$R_3(a), R_3(b), R_3(d), R_3(e)$	4h	
...	...				
$(4k-9)$-$(4k-6)$	$R_{k-2}(a), R_{k-2}(b), R_{k-2}(d), R_{k-2}(e)$	4m	(a,f), (e,c), (c,d), (b,f)	4h	
$(4k-5)$-$(4k-2)$	same as 3-6 of Table 2				

Table 3: First half of a sequence of requests on which $\mathcal{LRU}_2$ incurs $2k-1$ times the number of misses incurred by $\mathcal{OPT}$. The sequence generalizes the sequence given in Table 2.

and the other side incident to a new node. The new nodes are such that no more than k edges from $R_1 \cup \cdots \cup R_{k-2}$ are incident to them.[4] Note that this implies that all evictions are initiated at the hexagon nodes (a–f).

The construction is such that $\mathcal{OPT}$ never evicts R_i's edges. Each 6-sequence-part of the original 12-sequence is replaced by a new sequence of length $6+4(k-2)$. The idea is that before evicting the 4 "hexagon" edges, $\mathcal{LRU}_2$ first evicts 4 R_1 edges, then these 4 edges are requested and $\mathcal{LRU}_2$ evicts 4 R_2 edges to accommodate them, and so forth until 4 R_{k-2} edges are evicted and requested. Only then, $\mathcal{LRU}_2$ evicts the 4 "hexagon" edges evicted in the $k=2$ construction. In this process, $\mathcal{OPT}$ incurs only 2 misses while $\mathcal{LRU}_2$ incurs $4k-2$ misses. A similar expansion is performed on the second part of the 12-sequence. In total, $\mathcal{OPT}$ incurs 4 misses while $\mathcal{LRU}_2$ misses on all $8k-4$ requests, establishing that the competitive-ratio is at least $2k-1$.

In the start of the first and second part (when $\mathcal{LRU}_2$ and $\mathcal{OPT}$ have the same configuration), we set the relative ages of edges from young to old to be such that the relative ages of the hexagon edges is the same as for the $k=2$ case, all R_i edges are "older," and age is decreasing with index. For example, the corresponding order at the start of the first part is

$$\{(a,b),(d,e)\} < \{(b,f),(c,e)\} < \{(a,f),(c,d)\} < R_{k-2} < \cdots < R_1 \ .$$

The extended first part of the sequence (augmenting the first 6 requests) is provided in Table 3. The extended second part is similar. Note that at the end of each part, $\mathcal{LRU}_2$ and $\mathcal{OPT}$ have identical configurations, and the configuration at the end of the second part is isomorphic to the starting point.

6.2 Connection-caching castings of the Landlord algorithm

Young [29] recently proposed the Landlord algorithm (LL) as a generalization of marking algorithms to a scenario when fetching costs and sizes vary. The Landlord algorithm maintains a credit for each cached page. The credit value is nonnegative and is at most the cost of the page. The credit is set to the cost when the page is brought into the cache and is decreased in proportion to the page size when page(s) need to be evicted to make room for a newly requested page. Upon a hit, the credit of a page can optionally be increased to at most the cost.

The uniform-sizes and costs restriction of LL corresponds to marking algorithms (as defined in the beginning of the section).

The LL algorithm has natural interpretations as a type-1, type-2 and type-3 connection caching algorithms. In essence, $\mathcal{LL}_1$, $\mathcal{LL}_2$, and $\mathcal{LL}_3$ generalize our definitions of connection-caching marking

[4] The construction requires $12 - \lceil 12/k \rceil$ nodes in total.

algorithms. For $\mathcal{LL}_3$, a single credit is maintained for each connection. The credit can be viewed and decreased by both endpoints, and can be increased only as a result of a cache hit. With $\mathcal{LL}_1$ and $\mathcal{LL}_2$, a separate credit for a connection is maintained by each endpoint. With $\mathcal{LL}_2$, a node may decrease credits of incident open connections only if there are not sufficiently many holes and then zero-credit connections to accommodate the new connection.

Young established that LL is $k/(k-h+1)$-competitive. His argument is based on the potential function

$$\Phi = (h-1) \sum_{f \in \text{cache(LL)}} \text{credit}(f) + k \sum_{f \in \text{cache(OPT)}} [\text{cost}(f) - \text{credit}(f)],$$

where $\text{credit}(f) \equiv 0$ when f is not in cache(LL). Albers [2] observed that Young's proof carries over for $\mathcal{LL}_3$. The proof holds when cache sizes are uniform across nodes (for both the network and the adversary). We note that a somewhat more general potential function:

$$\Phi = \max_u \frac{h_u - 1}{k_u} \sum_{f \in \text{cache(LL)}} \text{credit}(f) + \sum_{f \in \text{cache(OPT)}} [\text{cost}(f) - \text{credit}(f)]$$

can be used to handle non-uniform cache sizes and establish bounds as of Theorem 6.4 for $\mathcal{LL}_3$..

We do not know whether a similar argument can be used to generalize Theorem 6.5 or even Theorem 6.3 to $\mathcal{LL}_1$ or $\mathcal{LL}_2$. The current best upper bound we know for both $\mathcal{LL}_1$ and $\mathcal{LL}_2$ is

$$2 \max_u \frac{k(u)}{k(u) - h(u) + 1} \quad .$$

The bound for $\mathcal{LL}_1$ follows from Theorem 4.4. The bound for $\mathcal{LL}_2$ follows from a potential function argument, similar to the one used by Young, using the potential function

$$\Phi = \max_u \frac{h_u - 1}{k_u} \sum_{f \in \text{cache(LL)}} \text{credit}(f) + \sum_{f \in \text{cache(OPT)}} [2\text{cost}(f) - \text{credit}(f)] + \max_u \frac{k_u - h_u + 1}{k_u} \sum_{f \in \text{cache(LL)}} \text{cost}(f),$$

where in this case we define $\text{credit}(f)$ to be the *sum* of the credits assigned to f at the two endpoints. Hence, $0 \leq \text{credit}(f) \leq 2\text{cost}(f)$.

7 Randomized marking algorithms

We define three randomized marking algorithms. The algorithms differ by the type of communication used and whether they use a shared or local bits for each connection. $\mathcal{MARK}_1$ and $\mathcal{MARK}_2$ use local mark bits and $\mathcal{MARK}_3$ has shared mark bits. All three algorithms evict an unmarked connection uniformly at random.

The following upper bound on the performance of type-1 $\mathcal{MARK}$ follows from Theorem 4.4 and a tight bound on the performance of the marking algorithm for a single cache [1].

Theorem 7.1 *$\mathcal{MARK}_1$ is $(4H_k - 2)$-competitive.*

7.1 Type-2 randomized marking algorithms

In this section we prove that $\mathcal{MARK}_2$ is $4H_k$-competitive. We note that the distributed nature of connection caching made the analysis of $\mathcal{MARK}_2$ considerably more challenging than for the single-cache and type-1 setups. Our proof techniques incorporate several new ideas that are specific to the connection-caching model.

Theorem 7.2 *$\mathcal{MARK}_2$ is $(4H_k)$-competitive.*

Proof: Let u be a node of the network. We break the sequence of requests that touch u into *phases* in the following way. The first phase begins with the first request. Each subsequent phase of u starts with a request that causes a miss when cache(u) is full and all the connections in it are marked. Thus, the first request in each phase, except the first one, unmarks all the connections in cache(u) and evicts one of them, chosen uniformly at random. Each phase but the last ends with a request that fills cache(u) with marked connections. Note, however, that not every such request finishes a phase, as some of the connections may be dumped from cache(u) before the next miss. The last phase ends with the last request in the sequence.

Consider a phase of the sequence of requests touching u. The connections that reside in cache(u) at the start of the phase are said to be *old* connections. All the other connections requested during the phase are *new* connections. We divide the requests that generate misses during the phase into three categories:

A First requests to new connections.

B Requests to connections that were already in cache(u) during the phase but are not currently in cache(u) because they were evicted, i.e., removed from cache(u) by u.

C Requests to connections that were already in cache(u) during the phase but are not currently in cache(u) because they were dumped, i.e., removed from cache(u) by their other endpoint.

We let A, B and C, respectively, be the sets of misses that belong to these three categories. We further break each one of these categories into two sub-categories, depending on whether the request caused an eviction of a connection from cache(u), or filled a hole in cache(u). We denote the resulting classes by A_1, A_2, B_1, B_2 and C_1, C_2, where the misses of A_1, B_1 and C_1 are the requests that cause evictions, while the misses of A_2, B_2 and C_2 fill holes.

We let evict(u) be the number of evictions from cache(u) during the phase. Clearly evict$(u) = |A_1| + |B_1| + |C_1|$. Our next goal is to show that evict$(u) \leq |A| + |B|$. To that end, we show that $|C_1| \leq |A_2| + |B_2|$. We prove, in fact, the slightly stronger result evict$(u) \leq |A| + |B'|$, where B' is the subset of the misses in B that occur while cache(u) still contains unmarked connections. To that end, we show that $|C_1| \leq |A_2| + |B'_2|$, where $B'_2 = B_2 \cap B'$.

We start by defining a one to one mapping $f : C \to A_2 \cup B_2 \cup C_2$. Let $r \in C$ be a request for a connection (u, v) that was dumped from cache(u) during the current phase. Let r' be the request that fills the hole caused by the dumping of (u, v). (We assume that r fills the hole created by dumping (u, v) if this hole still exists when serving r. Therefore r' must exists.) As r' fills a hole, we get that $r' \in A_2 \cup B_2 \cup C_2$. We map r into r', i.e., we let $f(r) = r'$. It is easy to see that f is indeed one to one. We note, that an element of C_2 may be mapped into itself.

We can now define a directed graph on $A_2 \cup B_2 \cup C$ in the following way. If r is mapped by f to r', we put in an edge from r to r'. As the indegree and outdegree of each node are at most one, this graph is composed of a collection of cycles and directed paths. Each request of C_1 starts a path in this graph. The endpoints of the paths in the graph must be from $A_2 \cup B_2$, as requests from C_2 are mapped by f to other requests. We can thus define a one to one mapping $g : C_1 \to A_2 \cup B_2$ in the following way: map each request $r \in C_1$ to the last request r' on the path that starts at r. Thus $|C_1| \leq |A_2| + |B_2|$, as required.

To obtain the stronger inequality $|C_1| \leq |A_2| + |B'_2|$, we note that if $r \in C_1$, then r causes an eviction from cache(u). It follows that when r arrives, there are still unmarked connections cache(u).

It is not difficult to verify that if r is mapped by g into r', then r' occurs *before* r. Thus, there are still unmarked connection in cache(u) when r' arrives. It follows that g is a mapping from C_1 to $A_2 \cup B_2'$, and therefore $|C_1| \leq |A_2| + |B_2'|$, as required.

We should point out, at this point, that the partition of the requests touching u into phases, the sets A, B and C, and evict(u) are all random variables, as they are determined by the random choices made by u and by the other nodes in the network. Everything we said so far holds, however, for any random choice made by the algorithm.

We would now like to bound $E[\text{evict}(u)]$, the expected number of evictions at u during the phase. To that end, we want to bound $E[|A|]$ and $E[|B'|]$. Let N be the number of new connections accessed in the phase. (This is again a random variable as the length of the phase is random.) Clearly $|A| \leq N$, and thus $E[|A|] \leq E[N]$. We next turn our attention to bounding $E[|B'|]$.

The key observation used in bounding $|B'|$ is that if $r \in B'$, then r is the first request in the phase for an old connection (u, v) that was evicted from cache(u). The fact that (u, v) was evicted from cache(u) follows from the definition of B'. As (u, v) must have been unmarked when it was evicted, it follows that (u, v) is an old connection. The request r must be the first request to (u, v) during the phase, as after entering the cache (u, v) may be dumped but it will never be evicted again until the end of the phase.

With each one of the k old connections of a phase we associate a *critical event*. The critical event corresponding to an old connection (u, v) is the first request for (u, v) during the phase, or the first dump of (u, v) during the phase, if it comes earlier. It is possible, of course, that neither of these happen, in which case no critical event is associated with (u, v). We consider an old connection as unmarked from the beginning of the phase until the critical event associated with it happens or until the end of the phase. Note that the eviction of an old connection cannot be the critical event associated with a connection and we still consider the evicted old connection as marked.

It follows from the discussion above that $r \in B'$ if and only if r is a miss, there are still unmarked connections in cache(u), and r is the critical event corresponding to some old connection. We let X_j, where $1 \leq j \leq k$, be a random variable that gets the value 1 if the j-th critical event in the phase is a miss in B' (i.e. it is a miss occurring while there are still unmarked connections in cache(u).) Otherwise X_j takes the value 0. Clearly, $|B'| \leq \sum_{j=1}^{k} X_j$.

We now claim that the k-th critical event in a phase, if it happens, never causes a miss of type B', or in other words, that $X_k = 0$, always. If the k-th critical event is a dump, then we are done. Suppose, therefore, that the k-th critical event is the first request r in the phase for an old connection (u, v). If (u, v) is in cache(u) when r arrives, then we have a hit. Otherwise, since we know that all the other $k - 1$ old connections were already requested during the phase, or were already dumped, there is no unmarked connection in cache(u). Thus, $r \notin B'$.

Let N_j be the number of new pages requested during the phase before the j-th critical event. Let U_j be the number of unmarked pages that are out of cache(u) when the j-th critical event of the phase occurs. (Recall that when an unmark connection is evicted from cache(u), we still consider it to be unmarked. However, if it is dumped we no longer consider it to be unmarked.)

It is straightforward to prove by induction on the request sequence of the phase that the number of unmarked and dumped old connections outside cache(u) is at most the number of new connection plus the number of holes. Since the number of dumped connections outside the cache is at least the number of holes it follows that for every j, $U_j \leq N_j \leq N$. Also, it is not difficult to see that the set of unmarked pages that are out of cache(u) is chosen uniformly at random from among all the subsets of size U_j of the unmarked connections. It follows that

$$E[X_j] \;\le\; \frac{E[U_j]}{k-j+1} \;\le\; \frac{E[N]}{k-j+1} \;.$$

By summing up, we get that

$$E[\sum_{j=1}^{k-1} X_j] \;\le\; E[N] \cdot \sum_{j=1}^{k-1} \frac{1}{k-j+1} = E[N] \cdot (H_k - 1) \;,$$

and therefore

$$E[\text{evict}(u)] \;\le\; |A| + |B'| \;\le\; E[N] + E[N] \cdot (H_k - 1) \;=\; E[N] \cdot H_k \;.$$

By summing up over all the phases starting from the second (there are no evictions at the first phase if the caches are empty at the beginning), we get that

$$E[\sum_i \text{evict}_i(u)] \;\le\; E[N(u)] \cdot H_k \;,$$

where $N(u) = \sum_{i>1} N_i$, and N_i is the number of new connections seen by u during phase i. It is easy to see that the total number of evictions plus the number of connections in the caches of $\mathcal{MARK}_2$ at the end of the sequence is an upper bound on the number of misses of $\mathcal{MARK}_2$.

The number of different connections that touch node u during phases $i-1$ and i, for every $i > 1$, is at least $k + N_i$. Therefore, the optimal off-line policy must have had at least N_i misses incident with u during each pair of consecutive phases $i-1$, and i, for $i > 2$. Also if the caches are empty at the beginning of the sequence then the number of misses of the optimal off-line policy at the first two phases is at least $k + N_2$. Since each real miss may be counted in two different phases of two different nodes we conclude that the total number of misses of the optimal algorithm is at least $\frac{1}{4}(\cdot \sum_u N(u) + kn)$. The theorem now follows by dividing the upper bound on the number of misses of $\mathcal{MARK}_2$ by the lower bound on the number of misses of the optimal off-line policy. □

7.2 Type-3 randomized marking algorithms

We analyze the performance of MARK_3, which utilizes a shared mark bit per connection. Clearly, Theorem 6.2 applies to this algorithm and hence MARK_3 is k-competitive. Somewhat surprisingly, we show that this is tight! We describe a sequence on which the expected number of misses of MARK_3 is k times bigger than the number of misses which OPT incurs. Thus, the additional communication power available to MARK_3 over that of its type-1 and type-2 counterparts turns out to be counter-productive.

Theorem 7.3 *The competitive ratio of $\mathcal{MARK}_3$ is at least k.*

Proof: Let the set of nodes in our network be $V = \{x_i, y_i \mid 1 \le i \le k-1\} \cup \{v, w, z_1, z_2\}$,. We now define a family of sequences consisting only of connections from the set $E = A \cup B \cup C \cup \{(v, z_1), (v, z_2), (w, z_1), (w, z_2)\}$ where $A = \{(x_i, y_j) \mid 1 \le i, j \le k-1\}$, $B = \{(v, x_i), \mid 1 \le i \le k-1\}$, and $C = \{(w, x_i), \mid 1 \le i \le k-1\}$. (See Figure 3.) We assume that all caches are of size k.

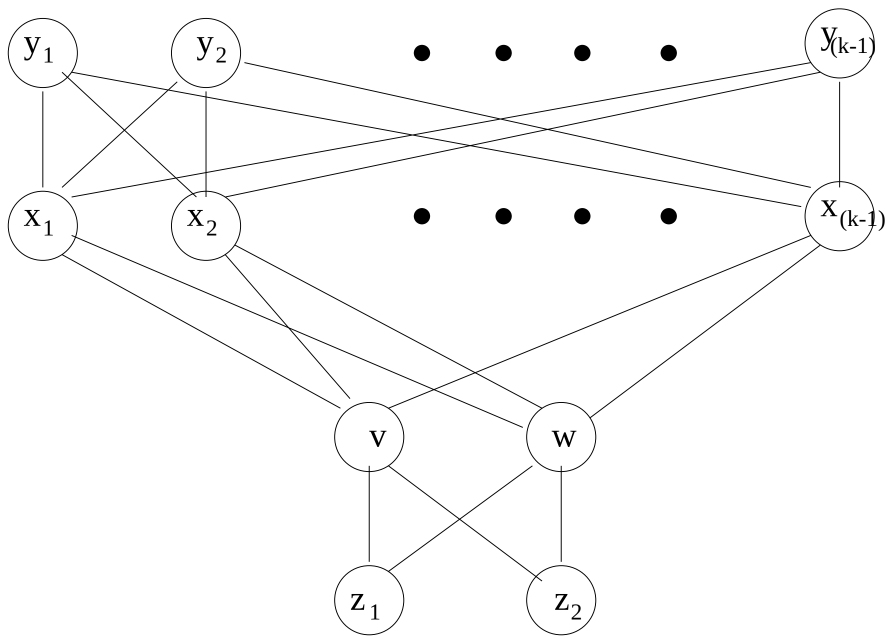


Figure 3: The connections requested by the family of sequences showing that the competitive ratio of type-3 randomized marking is at least k.

Each sequence starts with requests for all the connections in A, followed by two requests for connections (v, z_1) and (w, z_1). Then the l-th sequence in our family repeats the subsequence

$$s = B, (v, z_2), C, (w, z_2), B, (v, z_1), C, (w, z_1),$$

l time (by writing a set of connections in the sequence above we mean that each connection in the set is requested once in some fixed but arbitrary order).

It is easy to see that independently of its random bits type-3 randomized marking will miss every request in each of the sequences. It, in fact, acts deterministically since each time a node has to evict a connection it has only a single unmarked connection incident to it. Specifically, nodes $x_1, \ldots, x_{k-1}$ never unmark their incident connections and always have a single unmarked edge to evict. When x_i has to insert the connection (v, x_i) to its cache the connection (w, x_i) will be the only one which is unmarked (due to a previous unmarking at w) and a similar claim holds for v. On each request (v, z_1) or (v, z_2) node v unmarks its connections and similarly w unmarks its connections when it serves requests for (w, z_1) and (w, z_2).

The optimal off-line algorithm can serve each of these sequences such that it incurs misses only on requests for the connections (v, z_1), (v, z_2), (w, z_1), and (w, z_2). Thus aside for initial misses the optimal off-line algorithm incurs a single miss on every subsequence of k requests and the theorem follows. □

8 Conservative Algorithms

We explore classes of algorithms that generalize both LRU and FIFO. In particular, we define a class of *conservative* connection-caching algorithms analogous to the class of caching algorithms suggested by Young [30].

We first consider the following basic property:

Property 8.1 *Between the time a connection (u, v) enters* cache(u) *and the time it is evicted by u, there must be requests to at least k other connections touching u. (More specifically, there must be requests for connections $(u, v_1), \ldots, (u, v_k)$, where the v_i are all distinct and different from v.)*

It is not hard to see that $\mathcal{LRU}_1$ and $\mathcal{LRU}_2$ satisfy Property 8.1.

The caching algorithm FIFO defines the age of a cached item as the elapsed time since the item's most recent entry into the cache. Note that unlike LRU, FIFO does not update the age of an already-cached item when it is requested. When a miss occurs, FIFO, like LRU evicts the oldest item in its cache. We study three natural connection-caching algorithms derived from FIFO. The first two generalizations are the type-1 and type-2 algorithms $\mathcal{FIFO}_1$ and $\mathcal{FIFO}_2$. The third generalization, $\mathcal{FIFO}'_1$ can be viewed as a hybrid of $\mathcal{FIFO}_1$ and $\mathcal{LRU}_1$. It mimics $\mathcal{FIFO}_1$ on requests constituting hits, but mimics $\mathcal{LRU}_1$ in the event of a miss: if a request (v, w) is a miss, then if (v, w) is in cache(w) then $\mathcal{FIFO}'_1$ resets the age of (v, w) in cache(w) to zero. Similarly, $\mathcal{FIFO}'_1$ resets the age of (v, w) in cache(v) (note that since the request was a miss, (v, w) could have been cached in at most one of the two caches cache(v) and cache(w).) Recall that an upper bound of $2k - 1$ on the competitive ratio of $\mathcal{FIFO}_2$ follows from Theorem 6.3. It is evident that all three algorithms $\mathcal{FIFO}_2$, $\mathcal{FIFO}_1$, and $\mathcal{FIFO}'_1$ satisfy Property 8.1. We now establish the following:

Theorem 8.2 *Any connection caching algorithm satisfying Property 8.1 is $(2k - 1)$-competitive.*

Proof: Let $\mathcal{ALG}$ be an algorithm satisfying Property 8.1, and let $\mathcal{OPT}$ be the optimal off-line connection caching algorithm. Assume that $\mathcal{OPT}$ serves each request before $\mathcal{ALG}$. We will charge misses of $\mathcal{ALG}$ to misses of $\mathcal{OPT}$ such that at most $2k - 1$ misses of $\mathcal{ALG}$ are charged to each miss of $\mathcal{OPT}$.

Let m' be a miss of $\mathcal{ALG}$ generated by a request for connection (v, w). If (v, w) is ever evicted by $\mathcal{ALG}$ then let $u \in \{v, w\}$ be the endpoint at which $\mathcal{ALG}$ evicts (v, w). Otherwise let u be either v or w arbitrarily. We charge m' to the most recent miss of $\mathcal{OPT}$ on a connection incident to u (i.e. the last miss of $\mathcal{OPT}$ on a connection incident to u when serving the prefix of the request sequence up to and including r).

We shall prove that we charge at most $2k - 1$ misses of $\mathcal{ALG}$ to each miss of $\mathcal{OPT}$, by proving the following two claims. Let m be a miss of $\mathcal{OPT}$ incurred on request r for the connection (x, y).
Claim (1): We charge to m misses related to at most $k - 1$ different connections incident with x, and at most $k - 1$ different connections incident with y, other than (x, y).

Proof of Claim 1: Assume that we charge misses incurred on requests to k different connections $(x, v_1), \ldots, (x, v_k)$ to m, where $v_i \neq y$ for $1 \leq i \leq k$. Then by the time $\mathcal{ALG}$ gets the k-th such miss, say on the request for connection (x, v_k), $\mathcal{OPT}$ must have had a miss incident with x following m. This is because otherwise $\mathcal{OPT}$ must contain $(x, y), (x, v_1), \ldots, (x, v_k)$ in cache(x) after serving r which contradicts the fact that cache(x) is of size k. Therefore the miss on (x, v_k) cannot be charged to m and the claim follows.
Claim (2): We charge at most one miss on each particular connection to m.

Proof of Claim 2: In order to charge a second miss on say (x, z) to m connection (x, z) must be evicted at x first (or at y in case $z = y$). By property 8.1 in order to evict (x, z), x (or y) must see at least k different connections which are also different from (x, z). This implies that $\mathcal{OPT}$ must have had a miss at x (or y) after m and before the second miss on (x, z), which is a contradiction. $\square$

A sequence given in Section 6.1 shows that the $2k-1$ upper-bound is tight for $\mathcal{LRU}_1$ and $\mathcal{LRU}_2$ for every $k \geq 2$. It is possible to modify the sequence [5] such that $\mathcal{LRU}_1$ and $\mathcal{LRU}_2$ miss on every request of the modified sequence. Since FIFO$_2$ acts exactly as LRU$_2$ on sequences where LRU$_2$ misses on every request, and $\mathcal{FIFO}'_1$ acts as $\mathcal{LRU}_1$ on such sequences, we obtain a tight $2k-1$ upper bound for $\mathcal{FIFO}_2$ and $\mathcal{FIFO}'_1$.

Theorem 8.2 can be extended as follows: (similar to Theorem 6.4)

Theorem 8.3 *Any connection caching algorithm satisfying Property 8.1 is* $(\max_{(u,v)\in E}[\frac{k(u)}{k(u)-h(u)+1} + \frac{k(v)-1}{k(v)-h(v)+1}])$*-competitive against an off-line connection caching algorithm which at every node u of the network is only allowed to use a cache of size $h(u) \leq k(u)$.*

Proof: The proof is an extension of the proof of Theorem 8.2. We charge misses of $\mathcal{ALG}$ to misses of $\mathcal{OPT}$ as follows. Let r be a request for connection (v, w) on which $\mathcal{ALG}$ misses. If (v, w) is ever evicted by $\mathcal{ALG}$ then let $u \in \{v, w\}$ be the endpoint at which $\mathcal{ALG}$ is going to evict (v, w). Otherwise, pick u to be either v or w arbitrarily. We charge $1/(k(u) - h(u) + 1)$ of the miss on r to each of the $k(u) - h(u) + 1$ most recent misses of $\mathcal{OPT}$ incident to u.

Let m be a miss of $\mathcal{OPT}$ incurred on request r for the connection (x, y). We rephrase Claim 1 and Claim 2 as follows.

Claim (1): We charge to m misses related to at most $k(x) - 1$ different connections incident with x, and at most $k(y) - 1$ different connections incident with y, other than (x, y).

Claim (2): We charge a fraction of at most one miss on each particular connection to m.

It is easy to check that the theorem follows from these two claims. □

Young [30] defined a caching algorithm ALG as *conservative* if it satisfies the following property.

Property 8.4 *On any consecutive input subsequence containing k or fewer distinct page references,* ALG *will incur k or fewer misses.*

Young observed that LRU FIFO and other natural policies are conservative, and that all conservative caching algorithms are k-competitive.

Consider a type-1 or a type-2 connection-caching algorithm derived from a conservative caching algorithm. There may be more than k misses on requests for connections with v generated by a subsequence that includes requests for no more than k different connections with v. So Property 8.4 above may not hold from the perspective of a single node. This is due to dumps that occur at v. Therefore, in the context of connection caching we prefer to count evictions rather than misses (Note that misses and evictions are equivalent in caching algorithms). We define *conservative connection-caching* algorithms as follows.

Definition 8.5 *A connection caching algorithm is conservative if any subsequence of requests that contains only k distinct requests touching u, causes at most k evictions at u.*

Using this definition we have that any type-1 connection caching algorithm derived from a conservative single-cache policy is a conservative connection caching policy. Furthermore it is easy to see that any algorithm satisfying property 8.1 is a conservative connection caching algorithm. For conservative connection caching algorithms we can prove the following.

Theorem 8.6 *Any conservative connection caching algorithm is $(2k)$-competitive*

[5] we leave the details as an exercise to the reader.

Proof: We charge evictions of $\mathcal{ALG}$ to misses of $\mathcal{OPT}$ such that at most $2k$ eviction of $\mathcal{ALG}$ are charged to each miss of $\mathcal{OPT}$. The theorem then follows using Lemma 6.1.

We charge an eviction of $\mathcal{ALG}$ at u by a request r to the most recent miss of $\mathcal{OPT}$ on the prefix of the sequence of requests up to and including r on a connection with u. By the definition of a conservative connection caching algorithm a miss on a connection (x, y) is charged by at most k eviction at x and k evictions at y, giving the theorem. $\square$

Theorem 8.6 can be extended as follows: (along the lines of Theorem 6.4)

Theorem 8.7 *Any conservative connection caching algorithm is* $(\max_{(u,v)\in E}[\frac{k(u)}{k(u)-h(u)+1} + \frac{k(v)}{k(v)-h(v)+1}])$*-competitive against an off-line connection caching algorithm which at every node* u *of the network is only allowed to use a cache of size* $h(u) \leq k(u)$.

9 Conclusion and open problems

The underlying basic open questions are to further quantify the power of communication. Can the types be separated with appropriate upper and lower bounds?

We showed that all type-3 deterministic marking algorithms are k-competitive. One algorithm in this class, $\mathcal{FWF}$, is also a type-2 algorithm but not a demand caching algorithm. The best upper-bound we know of for the competitive-ratio of any type-1 and demand type-2 algorithms is $2k-1$ (tight for $\mathcal{LRU}_1$ and $\mathcal{LRU}_2$). Are there type-1 or demand-caching type-2 algorithms with better competitiveness? If so, are there type-1 or type-2 marking or conservative algorithms that achieve better performance?

Similar questions arise for randomized algorithms. The randomized marking algorithm (MARK) [13] has competitive ratio of $2H_k - 1 = O(\log k)$ [1]. Fiat et al [13] gave an H_k lower bound on the competitiveness of any randomized paging algorithm. This lower bound carries over for randomized connection-caching algorithms. McGeoch and Sleator closed the gap for paging algorithms by introducing a randomized algorithm with competitive ratio of H_k [22]. Their algorithm combined with Theorem 4.4 yields a $2H_k$-competitive randomized type-1 connection-caching algorithm. We had seen that $\mathcal{MARK}_1$ and $\mathcal{MARK}_2$ are $O(H_k)$-competitive whereas $\mathcal{MARK}_3$ is no better than k-competitive. Is there an H_k-competitive randomized connection-caching algorithm? What is the optimal competitiveness of randomized algorithms of different types? Finally, we had seen that in the deterministic setting optimal competitiveness of k is achieved by a type-2 algorithm. What is the weakest communication model that supports an optimally-competitive randomized connection-caching algorithm?

Acknowledgements

We thank Balachander Krishnamurthy, Jeffrey Oldham, and Jennifer Rexford for participating in early discussions on caching connections.

References

[1] D. Achlioptas, M. Chrobak, and J. Noga. Competitive analysis of randomized paging algorithms. In *Proceedings of the 4th European Symposium on Algorithms*, pages 419–430. Springer-Verlag, Lecture Notes in Computer Science Vol. 1136, 1996.

[2] S. Albers. Generalized connection caching. In *Proc. 12th annual ACM Symposium on Parallel Algorithms and Architectures*, pages 70–78. ACM, 2000.

[3] Apache HTTP server project.
`http://www.apache.org`.

[4] S. Arora and C. Lund. Hardness of approximation. In D. Hochbaum, editor, *Approximation algorithms for NP-hard problems*, pages 399–446. PWS Publishing Company, 1997.

[5] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM systems journal*, 5:78–101, 1966.

[6] A. Borodin and R. El Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998.

[7] M. Chrobak, H. Karloff, T. Paye, and S. Vishwanathan. New results on the server problem. *SIAM Journal on Discrete Mathematics*, 4:172–181, 1991.

[8] E. Cohen and H. Kaplan. LP-based analysis of greedy-dual-size. In *Proc. 10th ACM-SIAM Symposium on Discrete Algorithms*. ACM-SIAM, 1999.

[9] E. Cohen, H. Kaplan, and J. D. Oldham. Policies for managing TCP connections under persistent HTTP. In *Proceedings of the World Wide Web-8 Conference*, 1999.

[10] E. Cohen, H. Kaplan, and U. Zwick. Connection caching. In *Proc. 31st Annual ACM Symposium on Theory of Computing*. ACM, 1999.

[11] E. Cohen, H. Kaplan, and U. Zwick. Connection caching under various models of communication. In *Proc. 12th annual ACM Symposium on Parallel Algorithms and Architectures*, pages 54–63. ACM, 2000.

[12] A. Feldmann, R. Cáceres, F. Douglis, G. Glass, and M. Rabinovich. Performance of Web proxy caching in heterogeneous bandwidth environments. In *Proceedings of the IEEE INFOCOM'99 Conference*, 1999.

[13] A. Fiat, R. Karp, M. Luby, L. McGoech, D. D. Sleator, and N. Young. On competitive algorithms for paging problems. *J. Alg.*, 12:685–699, 1991.

[14] H. Frystyk Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, and C. Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proceedings of the ACM SIGCOMM'97 Conference*, Cannes, France, August 1997.

[15] S. Irani. Page replacement with multi-size pages and applications to web caching. In *Proc. 29th Annual ACM Symposium on Theory of Computing*. ACM, 1997.

[16] S. Irani and A. R. Karlin. Online computation. In D. S. Hochbaum, editor, *Approximation algorithms for NP-hard problems*. PWS publishing company, Boston, MA, 1995.

[17] V. Jacobson. Congestion avoidance and control. In *Proceedings of the ACM SIGCOMM'88 Conference*, August 1988.

[18] A. R. Karlin, M. S. Manasse, L. A. McGeoch, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):70–119, 1988.

[19] S. Keshav, C. Lund, S. Phillips, N. Reingold, and H. Saran. An empirical evaluation of virtual circuit holding time policies in IP-over-ATM networks. *J. on selected areas in communication*, 13, 1995.

[20] S. Keshav and H. Saran. An empirical evaluation of virtual circuit holding times policies in IP-over-ATM networks. In *Proceedings of the IEEE INFOCOM'94 Conference*, 1994. `http://www.cs.cornell.edu/skeshav/papers/hold.ps`.

[21] C. Lund, N. Reingold, and S. Phillips. IP over connection oriented networks and distributional paging. In *Proc. 35th IEEE Annual Symposium on Foundations of Computer Science*. IEEE, 1994.
`http://www.cs.cornell.edu/skeshav/doc/94/2-16.ps`.

[22] L. McGoech and D. D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6:816–825, 1991.

[23] V. N. Padmanabhan and J. C. Mogul. Improving HTTP latency. *Computer Networks and ISDN Systems*, 28(1/2):25–35, December 1995.

[24] C.H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[25] P. Raghavan and M. Snir. Memory versus randomization in on-line algorithms. In *Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, pages 687–703. Springer-Verlag, Lecture Notes in Computer Science 372, 1989.

[26] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985.

[27] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list updates and paging rules. *Comm. ACM*, 28:202–208, 1985.

[28] N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11:525–541, 1994.

[29] N. Young. On line file caching. In *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms*. ACM-SIAM, 1998.

[30] N. E. Young. *Competitive paging and Dual-Guided On-Line Weighted Caching and matching algorithms*. PhD thesis, Princeton University, 1991.