Online Algorithms for Caching Multimedia Streams

Matthew Andrews* and Kamesh Munagala**

[1] Bell Laboratories, Murray Hill NJ 07974.
[2] Computer Science Department, Stanford University, Stanford CA94305.

Abstract. We consider the problem of caching multimedia streams in the internet. We use the dynamic caching framework of Dan et al. and Hofmann et al.. We define a novel performance metric based on the maximum number of simultaneous cache misses, and present near-optimal on-line algorithms for determining which parts of the streams should be cached at any point in time for the case of a single server and single cache. We extend this model to case of a single cache with different per-client connection costs, and give an 8-competitive algorithm in this setting. Finally, we propose a model for multiple caches in a network and present an algorithm that is $O(K)$-competitive if we increase the cache sizes by $O(K)$. Here K is the number of caches in the network.

1 Introduction

In the classical caching problem, requests are made on-line for *pages*. There is a *cache* that can hold a small number of pages. Whenever a request is made for a page we have a *cache hit* if that page is currently in the cache, and a *cache miss* if the page is not in the cache. The goal is to maintain a set of pages in the cache so that the total number of cache misses is minimized.

However, for the problem of caching multimedia streams in the Internet, a different model is more appropriate for two reasons. First, the size of some streams (e.g. video streams) can be extremely large which means that it is infeasible to fit entire streams in a cache. Second, one of the main reasons for using caches is to reduce the usage of bandwidth in the network. Hence, we are more concerned with the maximum number of *simultaneous* cache misses rather than the total number of cache misses.

To address the first issue we shall consider the framework of *dynamic caching* as proposed by Dan and Sitaram [3] and Hofmann et al. [5]. We restrict our attention to a single cache that can access streams from a single server. Suppose that there is a request for some data stream at time t_1 and another request for the same data stream at time $t_2 = t_1 + \Delta$. (In the terminology of [5], Δ is the *temporal distance* between the two requests.) Suppose also that there is enough

* Email: andrews@research.bell-labs.com.
** Email: kamesh@cs.stanford.edu. Supported by ONR N00014-98-1-0589. Part of this work was done while the author was visiting Bell Labs.

space in the cache to store Δ time units of the stream. Then, we can serve *both* requests using only *one* connection to the server (see Figure 1). We always cache the last Δ time units of the stream that were seen by the first request. The second request can always obtain from the cache the current stream data that it needs.

In this paper we consider the problem of determining which parts of the streams to maintain in the cache so as to minimize the number of simultaneous connections to the server. We hence minimize the bandwidth required on the link between the cache and the server. We also consider the case of multiple caches in a network and propose offline and online algorithms.

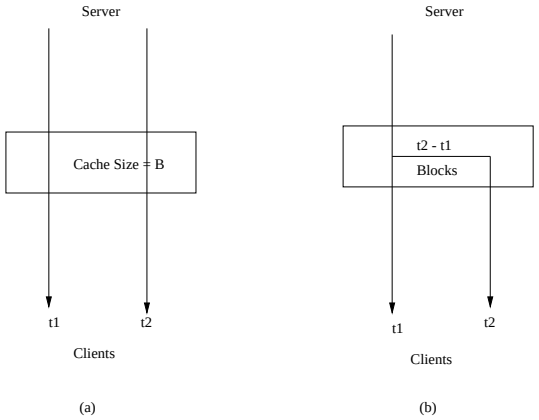

Fig. 1. *Dynamic Caching: (a) The two requests before caching. (b) After caching the interval $t_2 - t_1$, the bandwidth to the server reduces by one.*

1.1 The Models

Single cache. We begin by considering a cache and a server connected by a single link. There is a data stream (e.g. a video clip) of duration T stored in the server. We call one time unit of the clip a *block*, and denote by $b_1, b_2, \ldots, b_T$ the T blocks of the clip.

Requests for the clip arrive in an online fashion. We denote by t_i the time at which client i makes a request for the *entire* clip. Without loss of generality, $t_1 < t_2 < \ldots$. We assume that the client requests must be serviced *without delay*, *i.e.*, client i must receive the block b_j at time $t_i + j - 1$.

The cache has a buffer that can store B blocks of the clip. At time $t_i + j - 1$, if block j is in the cache, then client i can obtain it without cost (we have a *Cache Hit.*) If block b_j is not in the cache then it must be obtained from the server at a cost of one unit of bandwidth on the cache-server link (we have a *Cache Miss.*) In the latter case, when block b_j is obtained from the server, we

are allowed to place it in the cache, subject to the condition that we never cache more than B blocks.

Our goal is to determine which blocks of the clip should be cached at any point in time so as to minimize the maximum bandwidth that is ever used on the link. That is, we wish to minimize the maximum number of *simultaneous* cache misses. More specifically, let M_t^i be an indicator variable that equals 1 if cache i has a cache miss at time t and 0 otherwise. The bandwidth used at time t is $\sum_i M_t^i$. We wish to minimize $\max_t \sum_i M_t^i$.

Throughout this paper we shall focus on *interval strategies*. We define the interval i to be the set of blocks lying between the block currently required by request $i-1$ and the block currently required by request i. Hence at time t interval i consists of blocks $b_{t-t_i+1}, \ldots, b_{t-t_{i-1}}$. An interval strategy always tries to cache entire intervals.

We emphasize that our algorithms must be *online* in the sense that they cannot know about the request from client i until time t_i. However, since the request is for the entire stream of blocks, the algorithm will know at time t_i that block b_j will be needed at time $t_i + j - 1$. We shall use the framework of competitive analysis [13] and compare our candidate algorithms against the optimal *offline* algorithm that knows all the requests in advance.

Single cache with per-client connection costs. We next consider a more general model in which we have per-client costs. We assume that for each client there is a cost d_i for connecting the client to the cache, and a cost s_i for connecting the request directly to the server. These costs reflect the bandwidth required by these connections. There is also a cost c for connecting the cache to the server. For each block required by the client, if the block is present in the cache it can be obtained at a cost of d_i. Otherwise, the block can either be obtained via the direct connection to the server at a cost of s_i or else it can be obtained from the server and brought into the cache at a cost of $c + d_i$. Our goal is to minimize the maximum cost incurred at any time instant.

Multiple caches In our most general model we have many caches, each of which could be placed in a different part of the network. Cache j has size B and can be connected to the server at a cost of c_j. Client i can obtain a block from cache j at a cost of d_{ij}. If the block is not cached anywhere then client i can obtain it from the server and bring it into cache j at a cost of $c_j + d_{ij}$.[1]

Multiple clips Although the results we derive assume that the server has just one clip, the discussion can be easily extended to the case where the server has many clips.

1.2 Results

– In Section 2 we consider the single cache problem where we have a cache-server link and we wish to minimize the required bandwidth on this link. Let

[1] Note that we do not need to consider the cost of connecting to the server directly since we can assume that there is a cache at the server with $B_j = 0$ and $c_j = 0$.

U be the bandwidth of the optimal offline algorithm (that is not necessarily an interval strategy). We first present an interval strategy in which we always cache the smallest intervals that will fit into the buffer. We show that this strategy requires bandwidth at most $U + 1$. We also consider how to carry out this strategy when the requests arrive online. We first show how to do this when we have *auxiliary* buffer of size B in which we cache the first B blocks of the clip. We next show how to dispense with the auxiliary buffer at the expense of two additional connections to the server, i.e. the bandwidth required is $U + 3$.

- In Section 3 we analyze the single cache problem where we have per-client connection costs. We present an online algorithm that is 8-competitive with respect to the optimal offline interval strategy.
- In Section 4, we study the multiple cache problem. Let K be the number of caches. We first present an integer programming formulation for the problem of choosing the best set of intervals to store at each cache. We show that the solution of the LP-relaxation can be rounded to give an integral solution in which all caches are increased by a factor of $4K$ and the connection costs are increased by a factor $4K$. We also show that if the size of cache j is increased once more to $12KB$ then we can obtain an online algorithm that is $12K$-competitive with respect to the optimal offline interval strategy.

1.3 Previous Work

There has been much work on the classical caching problem in which requests are for arbitrary blocks and we wish to minimize the aggregate number of cache misses [1, 2, 4, 9, 13]. For example, Sleator and Tarjan [13] showed that the Least-Recently-Used (LRU) protocol is B-competitive (i.e. the number of cache misses is at most B times the number of misses due to the optimal offline algorithm). However, the problem of caching streaming data in a network has received less attention.

Dan and Sitaram presented the framework of dynamic caching in [3]. They propose an interval strategy similar to the algorithm we present in Section 2 in which we aim to cache the intervals that require the smallest buffer space. However, Dan and Sitaram do not present a theoretical analysis of this approach and they do not consider how to carry out the algorithm in the online setting.

Hofmann et al. [5] present heuristics for dynamic caching when there are many co-operating caches in a network. The caches store different sections of the stream and control processes called helpers decide which cache(s) to use for any particular client so that connection cost is minimized.

Sen et al. [11] study the benefits of caching as a method to minimize the delay associated with the playback of a multimedia stream. They advocate caching the initial parts of a clip so that the playback can be started earlier.

More generally, the problem of caching in distributed environments such as the Internet is receiving increasing attention. Li et al. [8] present algorithms for the placement of caches. Karger et al. [7] and Plaxton and Rajaraman [10]

give methods for the placement of data and the assignment of client requests to caches.

2 Single cache

We begin by considering the single cache model in which we wish to minimize the required bandwidth on the cache-server link. Let $\Delta_i = t_{i+1} - t_i$. We call the We include the interval between the first request and the most recently expired request in our list of intervals. We denote the length of this interval by Δ_0.

At any time instant t, let $\Delta_{i_1} \leq \Delta_{i_2} \leq \ldots \Delta_{i_n}$, and let $N_t = max\{n | \sum_{j=1}^{n} \Delta_{i_j} \leq B\}$. Note that N_t changes with time as new intervals get added and old ones cease to exist.

We first assume that the cache has an *auxiliary buffer* of size B, in addition to its regular buffer. We present a caching strategy that we call the GREEDY-INTERVAL-STRATEGY. It is similar to the caching scheme proposed in [3], and can be described as follows:

(1) Basic Strategy: Assume the current time instant is t. Identify the N_t smallest intervals in the current set of requests. Merge adjacent intervals in this set to create *super-intervals*. For a super-interval of length L, connect the request that came earliest in time to the server, and allocate L blocks of the cache to this super-interval. Store the last L blocks of the connected request in this space. Service all other requests in that super-interval from the cached blocks.

(2) Dying streams: When a request i expires, we say that the interval between request $i-1$ and i has *died*. This may cause the set of N_t cached intervals to change. To maintain this set online, we do the following. After request $i-1$ expires, if the dying interval is one of the N_t smallest intervals then we only cache that part of it which is required by the request i. If the current time is t, then we cache only the last $t_i + T - t$ blocks of the dying interval in the cache. Note that if the dying interval is evicted from the cache by a newly arriving interval, it will never get re-inserted, as re-insertions occur only when an interval dies.

(3) Vacant Space: At any instant of time, let δ be the number of blocks of cache that are not used by any interval. We cache the first δ part the smallest uncached interval in this space. We will show below that this can be done online.

(4) New Requests: When a request arrives, it creates a new interval. We re-compute N_t and find the N_t new smallest intervals and store these in the cache. Note that except the new interval, we do not add any interval to the existing set of intervals. Since we have the first B blocks of the clip in the auxiliary buffer, it is trivial to add the new interval.

The following theorem shows that GREEDY-INTERVAL-STRATEGY is a valid online strategy.

Theorem 1. *When an interval is added to the set of cached intervals, it is already present in the cache.*

Proof. We prove this by induction on time. Assume the claim is true at time t, *i.e.*, the N_t smallest intervals are present in the cache. The only events that change the set of cached intervals are listed below. We will show in each case that the new set of N_t smallest intervals are already present in the cache.

1. A new interval arrives. Since we cache the first B blocks of clip on the auxiliary buffer, we have this interval in cache when it arrives.
2. An interval expires when both streams corresponding to its end-points die. This may cause the interval i_{N_t+1} to get cached. But in this case, by step **(3)** of the algorithm, we would have cached this interval completely in the vacant space.

Note further that once a dying interval is removed from the cache by a newly arriving interval, it will never be re-inserted.

2.1 Competitive Analysis

The performance metric we use is the maximum bandwidth used by the algorithm. We compare it with an optimum offline strategy, which we call OPT. Let us assume that OPT uses U units of bandwidth on a sequence of requests. We can explicitly characterize OPT. It looks at all the X requested blocks at at any instant of time t, and caches those $X - U$ blocks for which the previous requests were closest in time. Note that these blocks must have been cached at some time in the past, so the algorithm is effectively looking into the future to decide which blocks to cache. The algorithm is *successful* if and only if it caches at most B blocks at any instant of time. We state the following results without proof.

Lemma 1. *Given* any *sequence of requests, if it is possible to cache the blocks in such a way that the maximum bandwidth required is U, then* OPT, *as described above is successful.*

Lemma 2. *If* OPT *caches the block required by request i_j at time t then it must also cache the block required by i'_j for all $j' < j$.*

We now compare the performance of GREEDY-INTERVAL-STRATEGY with the bandwidth-optimal strategy OPT.

Theorem 2. *If* GREEDY-INTERVAL-STRATEGY *uses bandwidth U at time t, then there exists time $t' \geq t$ at which* OPT *uses bandwidth $U - 1$.*

Proof. Suppose that OPT uses bandwidth $U - 2$ throughout the interval $[t, t + \Delta_{i_{N_t+1}})$.

There are two cases to consider.

- **Case 1** All but one of the streams that are *not* cached by the GREEDY-INTERVAL-STRATEGY at time t are still in the system at time $t + \Delta_{i_{N_t+1}}$. Then, since OPT uses bandwidth $U-2$ during the time period $[t, t+\Delta_{i_{N_t+1}})$, one of these streams must be cached by OPT throughout the time period $[t, t+\Delta_{i_{N_t+1}})$. By Lemma 2, this means that OPT caches N_t+1 intervals at time t. But, if we consider the N_t+1 smallest intervals at time t, we have $\sum_{j=1}^{N_t+1} \Delta_{i_j} > B$ (by definition of GREEDY-INTERVAL-STRATEGY). This is a contradiction.
- **Case 2** Two or more streams that are not cached by the GREEDY-INTERVAL-STRATEGY at time t die before time $t + \Delta_{i_{N_t+1}}$. However, by the definition of the GREEDY-INTERVAL-STRATEGY, the gap between these streams must be at least $\Delta_{i_{N_t+1}}$. This is a contradiction.

Note that if a stream is not cached at time t and dies during $t + \Delta_{i_{N_t+1}}$ then it must be the earliest stream in the system at time t.

Corollary 1. *The* GREEDY-INTERVAL-STRATEGY *requires at most one more unit of bandwidth than* OPT.

2.2 Removing the Auxiliary Buffer

We now present a scheme LOOKAHEAD-GREEDY that removes the auxiliary buffer of size B. First of all we modify GREEDY-INTERVAL-STRATEGY so that we only begin to cache an interval when there is enough free buffer space to cache the entire interval. This modification creates one extra connection to the server.

Suppose that a new request arrives at time t. If the new interval (of length δ, say) needs to be cached, we start caching the stream corresponding to the previous request starting from time t. By the above comment there is free buffer space equal to δ at time t. This space is completely filled up only at time $t+\delta$. Therefore, at time $t+\gamma$, there is unused space of size $\delta - \gamma$. We cache as many blocks of the beginning of the new stream in this space as possible. Note that the cached section of the new stream increases from 0 to $\frac{\delta}{2}$ blocks, but then decreases to 0, as the free space decreases.

Suppose that the next interval that must be cached arrives at time $t+\mu$. The size of this interval is at most μ. There are three cases depending on the value of μ:

1. If $\mu < \frac{\delta}{2}$, we have the beginning μ blocks in the cache, and we can serve the new stream directly from the cache, as in GREEDY-INTERVAL-STRATEGY.
2. If $\delta > \mu > \frac{\delta}{2}$, we will have the first $\delta - \mu$ blocks in the cache. This means that we can serve the new stream from the cache for the interval $[t+\mu, t+\delta]$. At time $t+\delta$ the first interval will be fully cached.
3. If $\mu > \delta$, the new stream does not arrive until the first interval is fully cached.

This implies that LOOKAHEAD-GREEDY requires at most one more connection to the server than the modified version of GREEDY-INTERVAL-STRATEGY.

Recall that the modified version of GREEDY-INTERVAL-STRATEGY requires one more connection than the original version. We have the following result.

Theorem 3. *If* OPT *requires maximum bandwidth U on any sequence of requests,* LOOKAHEAD-GREEDY *requires bandwidth at most $U + 3$.*

3 Single Cache with Connection Costs

We will now consider online algorithms in a more general cost model than the one used above. If some request is not part of any super-interval then it is not necessary for that request to be routed via the cache.

In this new model any client can either connect to the server directly or can connect via the cache. We assume that the connection cost per unit bandwidth from the cache to the server is c. (see Figure 2). Furthermore, for each request i we pay cost d_i if we connect the request to the cache. We pay s_i if we connect the request directly to the server. Without loss of generality we assume that $d_i \leq s_i \leq c + d_i$.

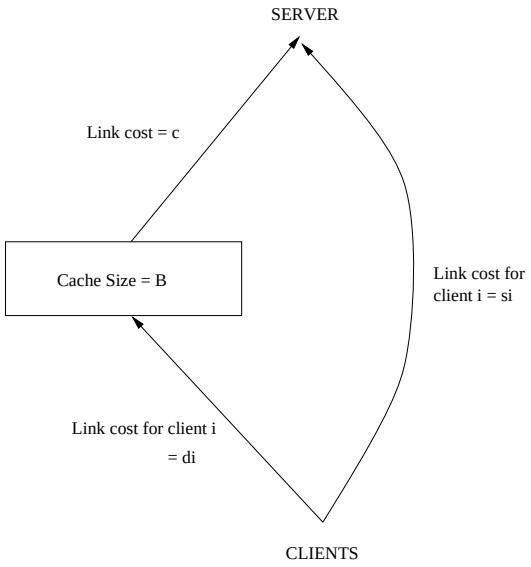

Fig. 2. *The cost model for a single server and single cache. Note that $d_i \leq s_i \leq c + d_i$.*

We shall compare our online scheme only against schemes which cache complete intervals from those formed by the active requests, at any instant of time. We call such schemes *Interval Caching* schemes. We do not consider offline algorithms that are not interval strategies. This does not matter as long as we

assume that the caching policy at a cache is *oblivious* to the algorithm that assigns requests to caches. More precisely, the caching protocol assumes that the requests currently assigned to the cache will remain assigned to that cache until they die.

The cost of an interval caching scheme at any time instant is simply the total cost of the bandwidth used in the cost model described above. Our objective is to derive an online strategy that is competitive (with respect to the maximum cost needed by any interval caching scheme) on a given sequence of requests.

As before, at any time instant, we number the requests $1, 2, \ldots, n$ according to the order in which they arrive. A *super-interval* is a contiguous set of cached intervals, each of which is serviced by one connection to the server. Note that the cost of connecting one super-interval to the server is c, irrespective of the number of requests cached within it.

If there are n active requests at any instant of time, we can compute the optimum interval caching strategy[2] for that time instant using dynamic programming. The running time of this dynamic program is $O(n^2 \sum s_i)$. However, we can obtain a polynomial time approximation scheme for this problem by scaling all the costs.

3.1 An Online Algorithm

We now show how to obtain a 8-competitive online algorithm with respect to the optimum interval caching strategy for this problem. The algorithm is very simple to describe. We periodically compute the optimum, and try to cache the intervals corresponding to it. We call this strategy the GREEDY-PREFETCH strategy.

Compute Optimum For the given set of requests, we compute the optimum solution. This solution may differ in some super-intervals from the previous optimum. We remove all super-intervals in the old optimum but not in the new one from the cache, and start caching the new super-intervals.

Greedy Caching For a new super-interval p, let L_p be the total uncached length. Note that this length need not be contiguous. Let m_p be the point on this super-interval where the length of the uncached portion is exactly $\lceil \frac{L_p}{2} \rceil$. Let the two halves be X_p and Y_p. Let S_p be the current total cost of connecting the requests in super-interval p to the server. Among X_p and Y_p we find that half with the larger current connection cost, and start caching that half of the super-interval with one connection to the server.

Prefetch When we compute the optimum and start caching the new super-intervals, the time it would take for the solution to stabilize is the size of the largest uncached part of any super-interval. Let us denote this amount of space as L. In this space, we cache the beginning of the clip with one extra connection to the server. We can cache at most $\lfloor \frac{L}{2} \rfloor$ blocks of the clip before this space starts to diminish in size. For any new request arriving within time $\lfloor \frac{L}{2} \rfloor$ of computing the optimum, we serve it from the cache. Note that the

[2] This problem is NP-hard, with a reduction from the Knapsack problem [6].

request arriving at time $\lceil \frac{L}{2} \rceil$ after computing optimum must be connected directly to the server. We can do this at an extra cost of c.

Recompute Optimum We recompute the optimum after time $\lceil \frac{L}{2} \rceil$ of the previous computation, and repeat the whole process. Note that if the optimal solution does not cache anything, we recompute the solution at the next request arrival.

3.2 Competitive Analysis

We will now show that GREEDY-PREFETCH is competitive with respect to the maximum optimum interval caching cost on any sequence of requests. Denote the times at which we compute the optimum solution by $x_1, x_2, \ldots$. Let the cost of the optimum interval caching strategy at time x_i be $M(i)$, and the cost of GREEDY-PREFETCH be $G(i)$. Also, let $S(i)$ be the total excess cost of connecting requests within a new super-interval to the server at time x_i. We state the following results without proof.

Lemma 3. *For the functions G, S and M as defined above, the following are true:*

1. $S(i+1) \leq 3M(i) + \lfloor \frac{S(i)}{2} \rfloor$.
2. $G(i) \leq 2M(i) + S(i)$.
3. *The cost G attains its maximum at one of the times x_i.*

Theorem 4. *Given any sequence of requests, if the optimum interval caching strategy can service them within a cost of C,* GREEDY-PREFETCH *requires cost at most $8C$ to service them.*

4 Multiple Caches

The most general version of this problem can be stated as follows. We have a single server, K caches and n clients in a network. The caches have capacity B and the connection cost per unit bandwidth to the server from cache j is c_j. The cost for connecting client i to cache j is d_{ij}. We first present an offline $O(K)$-approximation algorithm[3] for finding the optimum set of intervals to cache if we are allowed to increase each buffer size by a factor $O(K)$. We then show how to convert the offline scheme into an online scheme if we are allowed a further increase in the cache size.

4.1 Offline Approximation Algorithm

The server stores a single clip. Client i requests the clip at time t_i, and we assume that $t_1 \leq t_2 \leq \cdots \leq t_n$. Let $\Delta_i = t_i - t_{i-1}$. The objective is to compute

[3] The offline problem is NP-hard, with a reduction from the Generalized Assignment Problem [12].

the optimum set of intervals to store in each cache so that the total cost of the bandwidth used is minimized.

The above problem can be formulated as an IP as follows. Let x_{ij} be equal to 1 if request i is served via cache j, and 0 otherwise. Let y_{ij} be equal to 1 if request i is connected to the server via cache j. Let z_{ij} be equal to 1 if the interval $[t_{i-1}, t_i]$ is present in cache j. Note that request i need not be served by this interval[4].

$$\text{Minimize} \quad \sum_{j=1}^{K} \sum_{i=1}^{n} c_j y_{ij} + d_{ij} x_{ij}$$

$$\begin{aligned}
\textstyle\sum_{j=1}^{K} x_{ij} &= 1 && \forall i \in \{1, 2, \ldots, n\} \\
\textstyle\sum_{i=1}^{n} \Delta_i z_{ij} &\leq B && \forall j \in \{1, 2, \ldots, K\} \\
z_{ij} &\leq z_{i-1j} + y_{i-1j} && \forall i, j \\
x_{ij} &\leq z_{ij} + y_{ij} && \forall i, j \\
x_{ij},\ y_{ij},\ z_{ij} &\in \{0, 1\}
\end{aligned}$$

Definition 1. *A solution to the above IP is a* (α, β)**-approximation** *if the cost of the solution is at most* α *times the cost of the optimal fractional solution (obtained by relaxing the IP), and the required buffer space cache* j *used is at most* $\beta \cdot B_j$ *for all* j.

We first show a lower bound on the approximability via this formulation, without proof.

Theorem 5. *If the above IP is* (α, β)*-approximable by rounding its LP relaxation, then* $\alpha\beta \geq K$.

We will now show how to obtain a $(4K, 4K)$-approximation by rounding the relaxation of the IP described above. For each cache j, look at the variables in increasing order of i, applying the following rules. We denote the rounded x_{ij} as X_{ij}, y_{ij} as Y_{ij}, and z_{ij} as Z_{ij}.

1. $x_{ij} \geq \frac{1}{K} \Rightarrow X_{ij} = 1$, else $X_{ij} = 0$.
2. $z_{ij} \geq \frac{1}{2K}$, or $z_{ij} \geq \frac{1}{4K}$ and $Z_{i-1j} = 1 \Rightarrow Z_{ij} = 1$, else $Z_{ij} = 0$.
3. $y_{ij} \geq \frac{1}{2K}$, or $Z_{i+1j} = 1$ and $Z_{ij} = 0 \Rightarrow Y_{ij} = 1$, else $Y_{ij} = 0$.

Theorem 6. *The rounding scheme described above gives a* $(4K, 4K)$*-approximation.*

4.2 Online Scheme

We now show how to convert the offline algorithm into an online scheme. Suppose that we increase the buffer size at cache j to $12KB$. We compute the approximate offline solution every $4KB$ time steps. We use the additional buffer space as follows. At each cache we store the first $4KB$ blocks of the clip. Any new request i is served from the cache j with the smallest d_{ij} until the optimum is re-computed.

[4] For any super-interval, the first and last requests will be served by that super-interval. However, intermediate requests may not be served.

5 Open Problems

The most interesting question is whether it is possible to improve the approximation factor in the multiple cache offline problem. Another interesting question is whether it is possible to obtain online schemes in the multiple cache setting without blowing up the cache size. Finally, it would be interesting to see if it is possible to extend these schemes to the case where the requests are not for the complete clip, but for parts of it[5].

References

1. S. Albers, S. Arora and S. Khanna. Page replacement for general caching problems. *Proceedings of the 10th Annual Symposium on Discrete Algorithms*, 31–40, 1999.
2. L. Belady. A study of replacement algorithms for virtual storage computers. *IBM System Journal*, 5:78–101, 1966.
3. A. Dan and D. Sitaram. A generalized interval caching policy for mixed interactive and long video environments. *Multimedia Computing and Networking*, January 1996.
4. A. Fiat, R. Karp, M. Luby, L. McGeoch, D. Sleator and N. Young. Competitive paging algorithms. *Journal of Algorithms*, 12:685–699, 1991.
5. M. Hofmann, T.S.E. Ng, K. Guo, S. Paul and H. Zhang. Caching techniques for streaming multimedia over the internet. *Bell Laboratories Technical Memorandum*, May 1999.
6. D. Hochbaum. Various notions of approximations: Good, better, best, and more. In *Approximation Algorithms for NP-Hard Problems*, D. Hochbaum, Ed. *PWS Publishing Company*, 1995.
7. D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, 654–663, 1997.
8. B. Li, M. Golin, G. Italiano, X. Deng and K. Sohraby. On the optimal placement of web proxies in the internet. *Proceedings of INFOCOM '99*, 1282–1290, 1999.
9. L. McGeoch and D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6:816–825, 1991.
10. G. Plaxton and R. Rajaraman. Fast fault-tolerant concurrent access to shared objects. *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, 570–579, 1996.
11. S. Sen, J. Rexford and D. Towsley. Proxy prefix caching for multimedia streams. *Proceedings of INFOCOM '99*, 1310–1319, 1999.
12. D. Shmoys and É. Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62:461–474, 1993.
13. D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[5] If the requests are for arbitrary single blocks of the clip, then this reduces to the classical caching problem [13].