New Results on Web Caching with Request Reordering

Susanne Albers*

Abstract

We study web caching with request reordering. The goal is to maintain a cache of web documents so that a sequence of requests can be served at low cost. To improve cache hit rates, a limited reordering of requests is allowed. Feder et al. [6], who recently introduced this problem, considered caches of size 1, i.e. a cache can store one document. They presented an offline algorithm based on dynamic programming as well as online algorithms that achieve constant factor competitive ratios. For arbitrary cache sizes, Feder et al. [7] gave online strategies that have nearly optimal competitive ratios in several cost models.

In this paper we first present a deterministic online algorithm that achieves an optimal competitiveness, for the most general cost model and all cache sizes. We then investigate the offline problem, which is NP-hard in general. We develop the first polynomial time algorithms that can manage arbitrary cache sizes. Our strategies achieve small constant factor approximation ratios. The algorithms are based on a general technique that reduces web caching with request reordering to a problem of computing batched service schedules.

Our approximation result for the Fault Model also improves upon the best previous approximation guarantee known for web caching without request reordering.

*Institut für Informatik, Albert-Ludwigs-Universität Freiburg, Georges-Köhler-Allee 79, 79110 Freiburg, Germany. `salbers@informatik.uni-freiburg.de` Work supported by the Deutsche Forschungsgemeinschaft, projects AL 464/4-1, and by the EU, projects APPOL and APPOL II.

1 Introduction

We study web caching, the problem of maintaining a cache of web documents so that a sequence of requests can be served with low cost. Caches can be built into web browsers or servers. If these local memories store frequently accessed documents, then requested data does not have to be downloaded from the web. This leads to improved user response times and lower network congestion. There has been considerable research interest in the design of effective web caching algorithms [2, 3, 5, 6, 7, 11, 14]. Almost all of the work assumes that requests must be served in the order of arrival. However web requests are essentially independent and request reordering is a promising approach to improve cache hit rates. Therefore, Feder et al. [6, 7] recently initiated the study of web caching when a limited reordering of requests is allowed.

Formally, in *web caching with request reordering* we are given a request sequence $\sigma = \sigma(1), \ldots, \sigma(m)$. Each request specifies a document to be accessed. Associated with each document d is a size and a cost, denoted by $size(d)$ and $cost(d)$, respectively. If a requested document is in cache, the request can be served at 0 cost. Otherwise the incurred cost is $cost(d)$ and, after the service operation, the missing document may be loaded into cache at no extra cost. We emphasize here that the loading operation is optional. In web applications referenced documents are not necessarily brought into cache. At any time we may also load a document d not currently requested. Again, the incurred cost is $cost(d)$. If a document d is cache, it takes up a space of $size(d)$ bits. The cache capacity is K, i.e. at any time the cache can store documents with a total size of at most K bits. In web caching with request reordering, requests do not have to be served in the order of arrival. However, it is not desirable to delay the service of a request for too long. Let r be a positive integer. Request $\sigma(j)$ may be served before $\sigma(i)$ if $j - i < r$. The goal is to serve the input sequence so that the total cost is as small as possible. Following [6], we also call this framework the *r-reordering problem*.

We are interested in both online and offline algorithms. In the online scenario, for the r-reordering problem to make sense, when $\sigma(i)$ is the first unserved request in the sequence, requests $\sigma(j)$ with $j - i < r$ are known. Requests $\sigma(i + r)$ and beyond are unknown. An online algorithm A is called c-competitive if there exists a constant a such that, for all request sequences σ, $A(\sigma) \leq c \cdot OPT(\sigma) + a$. Here $A(\sigma)$ and $OPT(\sigma)$ denote the costs incurred by A and by an optimal offline algorithm OPT. In the offline scenario, the entire request sequence σ is known in advance. The general r-reordering problem is NP-hard. A polynomial time algorithm A achieves an approximation ratio of c if there exists a constant a such that $A(\sigma) \leq c \cdot OPT(\sigma) + a$, for all inputs σ.

Several cost models are of interest. Unless otherwise stated $size(d)$, for any document d, is an arbitrary positive integer.

- *Uniform Model*: All documents have the same size and incur a cost of 1 when not available in cache, i.e. $size(d) = s$ and $cost(d) = 1$, for some positive integer s and all documents d.
- *Bit Model*: For all documents d, we have $cost(d) = size(d)$, i.e. we measure how many bits are to be transferred when a referenced document is not in cache.
- *Fault Model*: Here $cost(d) = 1$, for all d, i.e. we count the number of cache misses.
- *General Model*: For any document d, $cost(d)$ may be an arbitrary non-negative value.

We note that the Uniform Model is equal to the cost model in standard paging. In web caching, documents typically have variable sizes and costs. Nevertheless several web caching papers considered the Uniform Model as it gives insight how to attack the more involved models.

Previous work: We first review the results on web caching without request reordering. In the Uniform Model the best competitive ratios of deterministic and randomized online algorithms are k and $H_k = \sum_{i=1}^{k} 1/i$, respectively, [1, 9, 12, 13]. Here k is the number of documents that can simultaneously reside in cache. These bounds hold if requested documents have to be in cache. The offline problem can be solved in polynomial time [4]. Karloff et al. [10] investigated web caching in the Uniform Model assuming that documents have expiration times beyond which they are no valid. Web caching in the Bit Model, and hence in the General Model, is NP-hard. For the Bit and the Fault Models, Irani [11] presented polynomial time offline algorithms that achieve approximation ratios of $O(\log k)$, where $k = K/D_{\min}$ and $D_{\min}$ is the size of the smallest document ever requested. She also developed randomized online algorithms for both models that are $O(\log^2 k)$-competitive. For the General Model, Young [14] and independently Cao and Irani [5] gave a deterministic k-competitive algorithm. They require that requested documents must be in cache. Albers et al. [2] presented polynomial time offline algorithms for the Bit, the Fault and the General Model. The algorithms achieve constant factor approximation ratios but use a slightly larger cache capacity. The total cache capacity needed is of the form $K + bD_{\max}$, where b is a small constant depending on the cost model and $D_{\max}$ is the size of the largest document ever requested. Bar-Noy et al. [3] have presented a 4-approximation algorithm for the General Model that does not need extra space.

We next consider web caching with request reordering. In a first paper, Feder et al. [6] studied the case that at any time the cache can store only one document, i.e. the cache size is 1. They gave constant competitive online algorithms. Furthermore they developed an offline algorithm based on dynamic programming that achieves a polynomial running time if r is logarithmic in the length of the request sequence or if the number of distinct documents requested is constant. In an extended version [7] of their paper, Feder et al. also presented online algorithms for arbitrary cache sizes. Suppose that a cache can simultaneously store k documents. For the Uniform Model, Feder et al. [7] gave deterministic $(k + 2)$-competitive algorithms. For the Bit and Fault Models, they showed $(k + 3)$-competitive strategies.

Our contribution: We present improved results on web caching with request reordering. Let $k = K/D_{\min}$, where $D_{\min}$ is the size of the smallest document ever requested. For the online problem we present a deterministic algorithm that works in the General Model and achieves a competitive ratio of $k+1$. This deterministic competitiveness is optimal, for all cost models and cache sizes. Our algorithm is a simple modification of Young's Landlord algorithm [14]. Nevertheless our strategy substantially improves upon previous results. Recall that, for the General Model, previous algorithms could only handle caches of size 1.

Most of our paper is concerned with the offline problem. We present the first polynomial time algorithms for caches of arbitrary size. As the offline problem, in general, is NP-hard, we design approximation algorithms. Our solutions are based on a new technique that reduces the r-reordering problem to one of computing batched service schedules. We partition a request sequence into batches of size r and serve these batches independently one after the other. In the Uniform Model we devise a 2-approximation algorithm. In the Bit and Fault Models, using LP rounding techniques of [2], we present approximation algorithms that use some extra space in cache. In the Bit Model we obtain a 2-approximation that needs an extra cache capacity of $D_{\max}$. More generally, for any $\epsilon \geq 0$, we obtain a $(2 + \epsilon)$-approximation using $D_{\max}/(1 + \epsilon/2)$ extra space. In the Fault Model we derive a 4-approximation using an additional memory of $2D_{\max}$. Trading again memory for performance we develop, for any $\epsilon > 0$, a $(2 + \epsilon)$-approximation using $(1 + 2/\epsilon)D_{\max}$ extra space. In practice the extra space requirements are small as $D_{\max}$ is typically not more than 1–2% of the cache size. Finally, for the General Model, applying the approach of Bar-Noy et al. [3], we obtain an 8-approximation requiring no extra space in cache.

We remark that in the Fault Model, our results also improve upon the best approximation guarantees known for web caching without request reordering. In Section 3.4 we present a refined analysis of the rounding algorithm given in [2] and show a $(1+\epsilon)$-approximation using $(1+1/\epsilon)D_{\max}$ additional memory. The previous best bound was a $(1 + \epsilon)$-approximation using $(1 + 1/(\sqrt{1 + \epsilon} - 1))$ extra space.

2 A deterministic online algorithm for the General Model

We present a deterministic online algorithm for the r-reordering problem in the General Model. The algorithm is a modification of Young's Landlord algorithm [14]. Any document d has a credit that takes values between 0 and $cost(d)$. Initially, all documents have a credit of 0. Given a request sequence σ, we maintain a sliding window W that always contains r consecutive requests of σ. In each step the following algorithm is executed.

Algorithm Modified Landlord (MLL)

1. For all $d' \in$ cache such that W contains unserved requests to d', serve those requests;
2. **if** first request in W is unserved **then**
3. Let d be the document referenced by this first request;
4. Serve all requests to d in W;
5. Set $\mathit{credit}(d) \leftarrow \mathit{cost}(d)$ and $C \leftarrow \{d\} \cup \{d' \mid d' \in \text{cache}\}$;
6. **while** $\sum_{d' \in C} \mathit{size}(d') > K$ **do**
7. Let $\Delta = \min_{d' \in C} \mathit{credit}(d')/\mathit{size}(d')$;
8. For each $d' \in C$, decrease $\mathit{credit}(d')$ by $\Delta \mathit{size}(d')$;
9. Delete from C and from the cache any document d' with $\mathit{credit}(d') = 0$;
10. **if** $\mathit{credit}(d) > 0$ **then** bring d into cache;
11. Shift W one position to the right;

Figure 1: The Modified Landlord algorithm

Let $k = K/D_{\min}$, where $D_{\min}$ is the size of the smallest document that can be referenced.

Theorem 1 *The algorithm MLL is $(k+1)$-competitive.*

Proof: Consider an arbitrary request sequence $\sigma = \sigma(1)\sigma(2)\ldots\sigma(m)$ and let D be the set of documents referenced in σ. We analyze MLL's performance using a potential function Φ. At any time let OPT be the set of documents in OPT's cache. Furthermore, at any time let S be the set of documents not stored in OPT's cache for which at least one request is already served by OPT but not by MLL. Define

$$\Phi = k \sum_{d' \in D} \mathit{credit}(d') + (k+1) \sum_{d' \in OPT \cup S} (\mathit{cost}(d') - \mathit{credit}(d')).$$

The potential is always non-negative since each $\mathit{credit}(d')$ takes values between 0 an $\mathit{cost}(d')$. We assume that MLL and OPT start with an empty cache so that the initial potential is 0.

Let time t be the point of time when $\sigma(t)$ is the first request in W. We prove that the amortized cost incurred by MLL between t and $t+1$ is at most $k+1$ times the cost paid by OPT during that time interval. This establishes the theorem. In the analysis we assume that OPT serves requests first and MLL serves second. We show the following statement.

(1) If OPT serves requests for a document d and incurs a cost of $\mathit{cost}(d)$, then Φ increases by at most $(k+1)\mathit{cost}(d)$. All other actions of OPT cannot increase the potential.

If OPT serves requests for a document d at a cost, then d is not in OPT's cache before the service operation. The document may be loaded into cache or become element of S, in which case the potential increases by $(k+1)(\mathit{cost}(d) - \mathit{credit}(d)) \leq (k+1)\mathit{cost}(d)$. If OPT serves requests at no cost, then the referenced documents are in cache and the potential does not change. Whenever OPT evicts documents from cache, the potential can only decrease.

Next we analyze the moves of MLL. The algorithm maintains the property that documents not in cache have credit 0. The only exception is the credit of a requested document d which is set to $\mathit{cost}(d)$ in line 5 of the algorithm. If the credit remains positive in the following execution of the while loop, then d is loaded into cache in line 10. Thus, when we start executing MLL for request $\sigma(t)$, documents not in cache do have credit 0. The execution of line 1 does not incur cost and cannot increase the potential since documents d' can only leave the set S. We investigate the case that request $\sigma(t) = d$ is unserved and prove the following statement.

(2) If MLL serves requests for document d at $cost(d)$, then Φ decreases by at least $cost(d)$.

Immediately before the execution of line 4 of MLL we have $d \in OPT \cup S$ because $\sigma(t)$ is already served by OPT. Moreover, $credit(d) = 0$. In line 5 of the algorithm $credit(d)$ is set to $cost(d)$. Hence the potential change due to lines 4 and 5 of MLL is $k\,cost(d) - (k+1)\,cost(d) = -cost(d)$.

It remains to show that Φ does not increase during the execution of the while loop. Consider an arbitrary iteration. For a set $X \subseteq D$, let $size(X) = \sum_{d' \in X} size(d')$. In line 8 the potential change is $\Delta(-k\,size(C) + (k+1)size(C \cap (OPT \cup S)))$. We have $C \cap S = \emptyset$ because when line 8 is executed, MLL has served all requests to documents $d' \in C$ that are in W and OPT cannot have served requests that are beyond W. Thus the potential change is $\Delta(-k\,size(C \setminus OPT) + size(C \cap OPT)) \leq \Delta(-k\,size(C \setminus OPT) + K)$. We argue that $C \setminus OPT$ contains at least one document. If $d \notin OPT$, then this is obvious. If $d \in OPT$, then there must exist a $d' \in C$, $d' \neq d$, with $d' \notin OPT$. Otherwise $C \subseteq OPT$, which implies $size(C) \leq K$ and the iteration of the while loop would not have started. Each document has a size of at least s and hence the potential change is upper bounded by $\Delta(-kD_{\min} + K) \leq 0$. Statement (2) now follows because whenever MLL removes documents from C and the cache in line 9, their credit is 0 and hence the potential does not change.

Statements (1) and (2) give the desired bound on MLL's amortized cost. □

The competitiveness of $k+1$ is best possible for deterministic online algorithms that do not have to load requested documents into cache, see [5, 8].

3 Offline algorithms

We develop polynomial time offline algorithms that achieve constant factor approximation ratios. Our algorithms are based on a general technique that transforms the r-reordering problem into one of computing batched service schedules. We first present this technique and then use it to develop approximation algorithms for the various cost models.

3.1 Batched processing

As in the previous section we imagine that an algorithm, processing a request sequence, maintains a sliding window W that always contains r consecutive requests. Requests to the left of W are served and request reordering is feasible within the window. We say that an algorithm A *serves a request sequence in batches* if, for any $i = 0, 1, \ldots, \lfloor m/r \rfloor$, A serves all requests $\sigma(ir+1), \sigma(ir+2), \ldots, \sigma(\min\{ir+r, m\})$ when $\sigma(ir+1)$ is the leftmost request in the sliding window W. Requests $\sigma(ir+1), \ldots, \sigma(\min\{ir+r, m\})$ are also referred to as *batch* i. Thus, when $\sigma(ir+1)$ becomes the leftmost request in W, all requests in batch i are still unserved. The batch is served while the position of W remains unchanged. Then W is shifted to $\sigma((i+1)r+1)$ if $i < \lfloor m/r \rfloor$. For any batch i, let $B(i)$ be the set of documents referenced in that batch.

Lemma 1 *Let A be an algorithm that serves a request sequence σ at cost C in the standard r-reordering model. Then there exists an algorithm A' that serves σ in batches and incurs a cost of at most $2C$.*

Proof: We transform A into an algorithm A' with the desired properties. Let $S_i, 0 \leq i \leq \lfloor m/r \rfloor$, be the set of documents stored in the cache maintained by A when $\sigma(ir+1)$ becomes the leftmost request in W. We assume w.l.o.g. that $S_0 = \emptyset$. By $S_{\lfloor m/r \rfloor + 1}$ we denote the set of documents in the final cache configuration. Let D_i be the set of documents for which A initiates service operations while processing batch i but which are neither in S_i nor in S_{i+1}. A document $d \in D_i$ is either loaded into cache but evicted before the end of the batch is reached or not loaded into cache at all. While processing batch i, algorithm A incurs a cost for serving documents $d \in D_i$ and and $d \in S_{i+1} \setminus S_i$. Thus the total cost of S is at least

$$\sum_{i=0}^{\lfloor m/r \rfloor} \left(\sum_{d \in S_{i+1} \setminus S_i} cost(d) + \sum_{d \in D_i} cost(d) \right).$$

Algorithm A' is now constructed as follows. When $\sigma(ir + 1)$ becomes the leftmost request in W, algorithm A' first serves requests to documents $d \in S_i$ that are referenced in batch i. These service operations incur no cost. Then A' serves requests to documents $d \in S_{i+1} \setminus S_i$ and loads them into cache by evicting documents $d' \in S_i \setminus S_{i+1}$. The cost of these operations is $\sum_{d \in S_{i+1} \setminus S_i} cost(d)$. Additionally A' schedules service operations for documents $d \in D_i$ without bringing them into cache. The service cost is $\sum_{d \in D_i} cost(d)$. If $i > 0$, A' finally schedules service operations for documents $d \in D_{i-1} \cup S_{i-1} \setminus S_i$ that are requested in batch i. Again these documents are not brought into cache and the service cost is $\sum_{d \in D_{i-1}} cost(d) + \sum_{d \in S_{i-1} \setminus S_i} cost(d)$. Since the original algorithm A serves every request in σ and, at any time, can only serve requests that are at most r requests ahead, every document referenced in batch i must be in $S_i \cup S_{i+1} \cup D_i$ or in $S_{i-1} \cup D_{i-1}$ if $i > 0$. Thus A' serves every request in σ.

To prove that the cost of A' is bounded by twice the cost of A, it suffices to show that

$$\sum_{i=1}^{\lfloor m/r \rfloor} \sum_{d \in S_{i-1} \setminus S_i} cost(d) \leq \sum_{i=0}^{\lfloor m/r \rfloor} \sum_{d \in S_{i+1} \setminus S_i} cost(d). \tag{1}$$

Consider a document $d \in S_{i-1} \setminus S_i$. This document is evicted by A during the processing of batch $i-1$. Let $j < i-1$ be the largest index with $d \in S_{j+1} \setminus S_j$. Match the eviction of d during batch $i-1$ with the most recent loading operation of d during batch j. Index j exists because $S_0 = \emptyset$. Since each loading operation is matched with at most one eviction, inequality (1) follows. □

3.2 The Uniform Model

We investigate the basic setting that all documents have the same size and incur a cost of 1 when being served or loaded into cache. We present a batched version of Belady's [4] optimum offline paging algorithm MIN, taking into account that requested documents do not necessarily have to be brought into cache. On a cache replacement, the algorithm evicts a document whose next unserved request occurs in the highest indexed batch possible. We use the following notation. Consider an algorithm that serves a request sequence in batches. At any given time during the processing and for any document d, let $b(d)$ be the index of the batch where the next unserved request to d occurs. If d is not requested again, let $b(d) = \lfloor m/r \rfloor + 1$.

Algorithm BMIN: Serve a request sequence in batches. When the processing of a batch starts, first serve all requests to documents that are currently in cache. While there is still a document d with unserved requests in the batch, execute the following steps. Serve all requests to d and determine $b = \max_{d' \in S} b(d')$, where S is the set of documents that are currently in cache. If $b(d) < b$, load d into cache and evict any document d' with $b(d') = b$.

We will show that BMIN is optimal among algorithms processing request sequences in batches. Lemma 1 then implies that BMIN achieves an approximation ratio of 2. BMIN does not necessarily evict the document from cache whose next unserved request $\sigma(l)$ in farthest in the future but evicts any document whose next unserved reference occurs in the batch of $\sigma(l)$. The following lemma, which we will also need in the optimality proof, implies that this does not cause any problems.

Lemma 2 *Let A be an algorithm that processes a request sequence σ in batches. Consider any time t during the processing. Let S be the set of documents that are currently in cache and let C be the cost incurred by A after time t until the end of σ. Let $d \in S$ and $d' \notin S$ be two documents with $b(d') \leq b(d)$.*

Then there exists an algorithm A' that starts at time t with the same set of served requests as A but cache configuration $S - \{d\} \cup \{d'\}$ and incurs a cost of at most C during the rest of the sequence. Algorithm A' operates in batches, using the same batch partitioning as A.

Proof: At time t algorithms A and A' start with the same set of served requests and the same cache configuration except that the configuration of A' stores d' instead of d. Algorithm A' simulates A until A evicts d or until A serves requests to d or d'. In the first case, when A evicts d, A' simply evicts d'. Algorithm A' is then in the same configuration as A and proceeds in the same way as A on the remaining request sequence. The cost of A' is equal to that of A. In the second case we study two scenarios depending on whether d or d' is served first.

Suppose that A first serves requests to d'. At that time A incurs a cost of 1 while A can serve the requests at 0 cost. If A does not load d' into cache, then A' loads document d by evicting d'. The incurred cost is 1. If A loads d' into cache and evicts a document $\overline{d} \neq d$, then A' load d by evicting $\overline{d}$. Again the incurred cost of A' is equal to 1. Finally, if A loads d' by evicting d, algorithm A' does not perform a cache replacement. In any case A' is in the same configuration as A and executes the same operations on the rest of the sequence. The cost of A' is bounded by that of A.

Next suppose that A first serves requests to d. Since $b(d') \leq b(d)$, these requests must be in the same batch as the next unserved requests to d'. We may assume w.l.o.g. that when A serves the batch, it never loads a document d_1 that is evicted again until the batch's processing ends. For, if A evicts d_0 to load d_1 and evicts d_1 to load d_2, we can modify the algorithm so that it serves the requests to d_1 without bringing the document into cache and evicts d_0 to load d_2.

When A serves the requests to d, algorithm A' serves the request to d' in the current batch. The simulation then proceeds until A serves requests to d'. At that time A incurs a cost of 1. We distinguish two cases depending on whether or not A loads d' into cache.

Assume that d' is not loaded into A's cache. If d is still in A's cache configuration when A serves the requests to d', then A' serves its unserved requests to d at a cost of 1 and loads d into cache by evicting d'. Algorithms A and A' have incurred the same cost and are again in the same configuration. On the rest of the sequence A' works the same way as A. If d is not anymore in A's cache configuration and hence has been evicted in the meantime, then in that eviction A' drops d'. Algorithm A' serves the requests to d without loading the document into cache. Again A and A' incurred the same cost and are in identical configurations so that they work the same for the rest of the request sequence.

Finally assume that d' is loaded into A's cache. Let $\overline{d}$ be the document evicted. If $\overline{d} = d$, then A' simply serves the requests to d without loading the document into cache. If $\overline{d} \neq d$ and if d is still in A's cache configuration, then A' serves the requests to d and loads it into cache by evicting $\overline{d}$. If $\overline{d} \neq d$ and d has been evicted in the meantime, then during that eviction A' first serves requests to $\overline{d}$ in the current batch at cost 0 and then evicts $\overline{d}$. The requests to d are served without bringing the document into cache. In all cases the cost of A' is not higher than that of A. Both algorithms are in identical configurations and work the same on the remaining sequence. □

Theorem 2 *For any request sequence σ, BMIN incurs the minimum cost among algorithms processing request sequences in batches.*

Proof: Consider an arbitrary sequence σ and let A be an algorithm that processes σ in batches and incurs minimum cost among such algorithms. In the following, by a *service operation* refer to an operation where an algorithm serves requests to a document in the current batch. The operation may or may not involve a cache replacement. Suppose that the first l service operations of A and BMIN are identical but that the $(l + 1)$-st operation is different in both algorithms. Initially l may be 0. We show that there exists an algorithm A' for which the first $l + 1$ service operations are identical to that of BMIN and whose total cost is bounded by that of A. Repetition of this step, for increasing l, yields the lemma.

Suppose that the $(l+1)$-st service operation is part of the processing of batch i. Immediately before the $(l+1)$-st operation A has the same cache configuration and the same set of served requests as BMIN. If during the $(l+1)$-st operation BMIN serves requests to a document d stored in cache, we are easily done. Up to and including operation l algorithm A' works in the same way as BMIN. Then it serves requests to document d, which does not generated any cost, and proceeds with all the remaining operation of A.

So suppose that during operation $l+1$ algorithm BMIN serves a request to a document d not in cache. At that time A and BMIN have served all requests in the current batch that are to documents stored in cache. As in the proof of Lemma 2 we may assume that in the remaining processing steps of the batch, algorithm A does not load a document into cache that is evicted again during these steps. Thus the documents being loaded are different from those being evicted. Since there are no dependencies, it does not matter in which order the documents to be served are actually being served. Hence we may assume w.l.o.g. that during the $(l+1)$-st service operation A also serves document d. We have to consider three cases.

(1) Both BMIN and A both load d into cache but the algorithms evict different documents from cache.

(2) Only BMIN loads d into cache.

(3) Only A loads d into cache.

Algorithm A' works as follows. On the first l service operations it executes the same steps as A and BMIN. The $(l+1)$-st operation is performed in the same way as BMIN. At that point the cache configuration of A and A' differ in one document. Algorithm A has a document d_1 while A' has a document d'_1. We will show that $b(d'_1) \leq b(d_1)$. Lemma 2 then yields the statement to be proven. In case (1) BMIN and A' evict a document d_1 while A evicts a document d'_1. By the eviction rule of BMIN, $b(d_1) \geq b(d'_1)$. In case (2), BMIN and A' drop d_1 while A does not load d and keeps d_1. Set $d'_1 = d$. Then, the definition of BMIN implies $b(d_1) > b(d'_1)$. Finally, in case (3), BMIN and A' do not load d while A evicts a document d'_1 to load d. Again, by the definition of BMIN we have $b(d_1) > b(d'_1)$ where $d_1 = d$. □

Lemma 1 and Theorem 2 yield the following result.

Corollary 1 *BMIN achieves an approximation ratio of 2.*

3.3 The Bit Model

In this section we consider the Bit Model, i.e. $cost(d) = size(d)$, for any document d. Again we design an approximation algorithm that processes a request sequences in batches. The algorithm proceeds in two steps. First it constructs a *fractional solution*, where documents are allowed to be fractionally in cache. We say that a document d *is fractionally in cache* if $0 < c(d) < size(d)$, where $c(d)$ denotes the number of bits of d present in cache. In this case the cost of serving requests to d or loading the remainder of d into cache is equal to $cost(d) - c(d)$. In a second step the algorithm rounds the fractional solution to a feasible integral solution. The strategy for designing fractional solutions is a bitwise implementation of BMIN. For any document d and any batch i, let $b(d, i)$ be the smallest index j, $j > i$, such that batch j contains requests to d. If d is not requested again after batch i, then $b(d, i) = \lfloor m/r \rfloor + 1$, where m is the length of the request sequence.

Algorithm BBMIN: Serve the request sequence in batches. For any batch i, first serve the requests to documents that are fully or fractionally in cache; consider those documents d in non-increasing order of $b(d, i)$ values. Then serve the requests to documents not present in cache. For any requested document d that is not fully in cache, execute the following steps after the service operation. Determine $b = \max_{d' \in S} b(d', i)$, where S is the set of documents that are fully or fractionally in cache. While $b(d, i) < b$ and $c(d) < size(d)$, perform two instructions: (1) Evict $\beta = \min\{size(d) - c(d), c(d')\}$ bits from any document d' with $b(d', i) = b$. (2) Load β missing bits of d and recompute $b = \max_{d' \in S} b(d', i)$.

We first argue that, for any request sequence, the cost incurred by BBMIN in the fractional cost model is a lower bound on the optimum cost in the standard integral cost model when restricting ourselves to algorithms that process sequences in batches. Let $\mathrm{BBMIN}(\sigma)$ be the cost incurred by BBMIN on σ in the fractional cost model. Furthermore, let $\mathrm{OPT}^B(\sigma)$ be the minimum cost achieved by an algorithm that processes σ in batches and produces an integral solution with $c(d) \in \{0, size(d)\}$ always, for all d.

Lemma 3 *For any σ,* $\mathrm{BBMIN}(\sigma) \leq \mathrm{OPT}^B(\sigma)$.

Proof: Let $\mathrm{OPT}^B_F(\sigma)$ be the minimum cost of an algorithm that processes σ in batches and is allowed to generate a fractional solution. Obviously, $\mathrm{OPT}^B_F(\sigma) \leq \mathrm{OPT}^B(\sigma)$. We show that the cost of BBMIN satisfies $\mathrm{BBMIN}(\sigma) = \mathrm{OPT}^B_F(\sigma)$. To this end consider a *Full Bit Model*, where the bits of the documents are viewed as indivisible mini-pages and a request to a document d is viewed as $size(d)$ requests to the corresponding mini-pages/bits. Let $\mathrm{OPT}^B_B(\sigma)$ be the minimum cost that can be achieved in this model using an algorithm that serves σ in batches. We have $\mathrm{OPT}^B_B(\sigma) \leq \mathrm{OPT}^B_F(\sigma)$ because any fractional solution can be viewed as a solution for the Full Bit Model with the same cost: If the solution serves requests to document d, then we assume that it, one after the other, serves requests to the corresponding mini-pages. The service operation to a mini-page may be accompanied by a cache replacement if the extent to which d is in cache increases.

Lemma 2 implies that BMIN achieves a cost of $\mathrm{OPT}^B_B(\sigma)$. We show that BBMIN constructs the same solution as an implementation of BMIN, i.e. at the end of each batch they have the same bits in cache. Thus the cost of BBMIN and BMIN is the same and this gives the desired equation $\mathrm{BBMIN}(\sigma) = \mathrm{OPT}^B_F(\sigma)$. Consider any batch i. BMIN does not specify in which order requests to mini-pages in cache or to mini-pages not in cache are served. Thus, with respect to the mini-pages in cache, we may assume that BMIN serves the requests in non-increasing order of $b(d, i)$-values, d being the document a mini-pages belongs to; service operations to mini-pages of the same document are grouped together. We refer to these service steps as *part 1 of batch* i. With respect to the mini-pages not in cache we assume that BMIN first serves requests to those belonging to documents d that were already addressed in part 1. The processing proceeds again in non-increasing order of $b(d, i)$-values. Then the service operations to the remaining mini-pages follow. In these service steps after part 1, which we refer to as *part 2 of batch* i, operations to mini-pages of the same documents are again grouped together. Now consider the documents d that occur both in part 1 and part 2. These documents are fractionally in cache at the beginning of the batch. Note that BMIN never evicts bits of a document d' to load bits of d if $b(d', i) < b(d, i)$. Thus, when serving requests to mini-pages of d in part 1, BMIN can as well serve all the mini-pages of d at that time since mini-pages evicted in d's processing of part 2 will not be served in the rest of part 1. Since the eviction rules of BMIN and BBMIN are the same, we may assume that their cache configuration at the end of each batch is indeed the same. □

Next we present our rounding algorithm, generating an integral solution from the fractional solution produced by BBMIN. We extend the notation. For any document d and any batch i, let $c(d, i)$ be the number of bits of d present in cache when the processing of batch i ends. We first modify the solution of BBMIN such that the extent to which a document is in cache does not change between two consecutive batches in which the document is requested. Suppose that d is referenced in batch i. Document d's presence in cache may decrease during the processing of batches $j = i + 1, \ldots, b(d, i) - 1$. The cost of the next references to d is equal to the number of bits of d not present in cache when batch $b(d, i) - 1$ ends, which is equal to $size(d) - c(d, b(d, i) - 1)$. Therefore, for any document d and batch i such that $d \in B(i)$, we set $c(d, j) = c(d, b(d, i) - 1)$, for $j = i, \ldots, b(d, i) - 2$. This does not change the cost of the solution and only frees up space in cache. The solution by BBMIN has an important property that is crucial for the actual rounding procedure: Consider two batches i and j with $i \leq j$ as well as two documents $d \in B(i)$ and $d' \in B(j)$. If $b(d', j) < b(d, i)$ and $c(d, l) > 0$ for $l = i, \ldots, b(d, i) - 1$, then $c(d', l) = size(d')$, for $l = j, \ldots, b(d', j) - 1$. This is because BBMIN prefers d over d' when evicting bits as d's next request is farther in the future.

The rounding procedure produces a solution that may need up to $K + \delta D_{\max}$ memory in cache, where $D_{\max}$ is the size of the largest document ever requested and $0 < \delta \leq 1$. Fix a δ with $0 < \delta \leq 1$. For batches

$i = 0, \ldots, \lfloor m/r \rfloor$ the procedure considers the documents $d \in B(i)$ with $(1-\delta)\mathit{size}(d) < c(d,i) < \mathit{size}(d)$. Document d is rounded up if, after the rounding, the rounded-up documents occupy extra memory of no more than $\delta D_{\max}$. Otherwise the document is rounded down to $(1-\delta)\mathit{size}(d)$. Finally, all values $c(d,i)$ with $0 < c(d,i) \leq (1-\delta)\mathit{size}(d)$ are rounded down to 0. The pseudo-code is given in Figure 2 below.

Algorithm Rounding

1. $\mathit{Extra} \leftarrow 0$;
2. **for** $i \leftarrow 0$ to $\lfloor m/r \rfloor$ **do**
3.   **for all** $d \in B(i)$ with $(1-\delta)\mathit{size}(d) < c(d,i) < \mathit{size}(d)$ **do**
4.   **if** $\mathit{Extra} + \mathit{size}(d) - c(d,i) \leq \delta D_{\max}$ **then**
5. $\mathit{Extra} \leftarrow \mathit{Extra} + \mathit{size}(d) - c(d,i)$;
6. $c(d,i) \leftarrow \mathit{size}(d)$, for $j = i. \ldots, b(d,i) - 1$;
7.   **else**
8. $\mathit{Extra} \leftarrow \mathit{Extra} - (c(d,i) - (1-\delta)\mathit{size}(d))$;
9. $c(d,i) \leftarrow (1-\delta)\mathit{size}(d)$, for $j = i. \ldots, b(d,i) - 1$;
10. **for all** $c(d,i)$ with $0 < c(d,i) \leq (1-\delta)\mathit{size}(d)$ **do**
11. $c(d,i) \leftarrow 0$;

Figure 2: The Rounding algorithm.

Theorem 3 *For any $\epsilon \geq 0$, we can construct a solution that incurs a cost of at most $(1+\epsilon)\mathrm{BBMIN}(\sigma)$ and uses an additional memory of at most $D_{\max}/(1+\epsilon)$.*

Proof: We first observe that $0 \leq \mathit{Extra} \leq \delta D_{\max}$ always. The second inequality follows from line 4 of the Rounding algorithm. If a document is rounded down, then $\mathit{Extra} + \mathit{size}(d) - c(d,i) > \delta D_{\max}$, which implies $\mathit{Extra} - (c(d,i) - (1-\delta)\mathit{size}(d)) > \delta D_{\max} - \delta \mathit{size}(d) \geq 0$.

We can show that at any time the total extra memory used by the rounded solution is at most the value of Extra. Details are omitted in this submission. Thus the extra space used by the rounded solution is at most $\delta D_{\max}$.

It remains to analyze the cost. During the executions of lines 1 to 9, the value of Extra is always equal to the cost savings relative to $\mathrm{BBMIN}(\sigma)$. If a document d is rounded up, then the cost savings on the next requests to d is $\mathit{size}(d) - c(d,i)$. If a document d is rounded down, the additional cost is $c(d,i) - (1-\delta)\mathit{size}(d)$. Since Extra is always non-negative, immediately before the execution of line 11, the cost of the rounded solution is bounded by $\mathrm{BBMIN}(\sigma)$. In line 11 the cost may increase by a factor of $1/\delta$. Replacing $1/\delta$ by $1+\epsilon$, the theorem follows. □

Corollary 2 *For any request sequence σ and any $\epsilon \geq 0$, we can construct a solution that incurs a cost of at most $2+\epsilon$ times that of an optimal solution. The extra space required is bounded by $D_{\max}/(1+\epsilon/2)$.*

3.4 The Fault Model

We investigate the Fault Model where $\mathit{cost}(d) = 1$, for all documents d, and design an approximation algorithm that processes a given request sequence in batches. As in the previous section, for any document d and any batch i, let $b(d,i)$ be the smallest index $j > i$ such that batch j contains requests to d and let $c(d,i)$ be the number of bits of documents d present in cache when the processing of batch i ends. The number of bits of d that are in cache initially is denoted by $c(d,-1)$. Unfortunately, we know of no simple combinatorial algorithm for constructing a good fractional solution operating in batches. Therefore, we formulate the caching problem as a linear program.

W.l.o.g. we may restrict ourselves to solutions in which the extent to which a document d is in cache does not change between two consecutive batches referencing d. Thus if $d \in B(i)$, then $c(d,j) = c(d, b(d,i)-1)$, for $j = i, \ldots, b(d,i) - 2$. The cost for serving requests to $d \in B(i)$ is equal to the fraction to which d is not in cache at the end of the previous batch, which is $1 - c(d, i-1)/size(d)$. The only additional constraint we have to impose is that the total size of documents in cache may not exceed K. Thus the linear program is as follows. Let D be the set of documents ever requested.

$$
\begin{array}{llcll}
\text{Minimize} & \displaystyle\sum_{i=0}^{\lfloor m/r \rfloor} \sum_{d \in B(i)} (1 - c(d,i-1)/size(d)) & & & \\
\text{subject to} & c(d,j) & = & c(d, b(d,i)-1)) & \forall d,i,j \text{ such that } d \in B(i) \\
 & & & & \text{and } i \leq j < b(d,i) - 1 \\
 & \displaystyle\sum_{d \in D} c(d,i) & \leq & K & \forall i \\
 & c(d,i) & \in & \{0, size(d)\} & \forall i, d
\end{array}
$$

We replace the constraint $c(d,i) \in \{0, size(d)\}$ by $0 \leq c(d,i) \leq size(d)$ so that the linear programming relaxation can be solved in polynomial time. Let $\mathrm{OPT}_{\mathrm{LP}}^{B}(\sigma)$ be the cost incurred by the LP relaxation on input σ. Clearly, this is a lower bound on the minimum cost of any integral solution processing σ in batches.

Fix an $\epsilon > 0$. We partition the documents into $\lfloor \log_{1+\epsilon} D_{\max} \rfloor + 1$ classes such that class $C_k, 0 \leq k \leq \lfloor \log_{1+\epsilon} D_{\max} \rfloor + 1$ contains documents d with $D_{\max}(1+\epsilon)^{-k} \geq size(d) > D_{\max}(1+\epsilon)^{-(k+1)}$. We round the optimal LP relaxation solution. We first modify the optimal LP relaxation solution such that for any two batches i and j with $i \leq j$ and documents $d \in B(i)$ and $d' \in B(j)$ the following property holds. If $b(d',j) < b(d,i)$ and $c(d,l) > 0$ for $l = i, \ldots, b(i,d) - 1$ then $c(d',l) = size(d')$, for $l = j, \ldots, b(d',j) - 1$. For any class C_k we consider the values $c(d,i)$ of documents $d \in C_k$. While there exist $c(d,i)$ and $c(d',j)$ with $0 < c(d,i) < size(d)$ and $0 < c(d',j) < size(d')$ such that $d \in B(i)$, $d' \in B(j)$, $i \leq j$ and $b(d',j) < b(d,i)$, we modify the values. We increase $c(d',l)$, for $l = j, \ldots, b(d',j) - 1$, and decrease $c(d,l)$, for $l = i, \ldots, b(d,i) - 1$, until the former ones are equal to $size(d')$ or the latter ones are 0. More precisely, let $\gamma = \min\{c(d,i), size(d') - c(d',j)\}$. We increase $c(d',l)$ by γ, for $l = j, \ldots, b(d',j) - 1$, and decrease $c(d,l)$ by γ, for $1 = i, \ldots, b(d,i) - 1$. The modification does not need extra space since $i \leq j$ and $b(d',j) < b(d,i)$. Let OPT^* be the solution obtained after all these modifications. Given OPT^*, we apply the Rounding algorithm described in Figure 2 separately for each class C_k. We use parameter $\delta = 1$ and replace $D_{\max}$ by $D_{\max}^k$, where $D_{\max}^k$ is the size of the largest document in C_k.

Theorem 4 *For any request sequence σ and for any $\epsilon > 0$, we can construct a solution that processes σ in batches and incurs a cost of at most $(1+\epsilon)\mathrm{OPT}_{\mathrm{LP}}^{B}(\sigma)$. The additional memory used by the solution is at most $D_{\max}(1 + 1/\epsilon)$.*

Proof: Solution OPT^* does not use extra space, while the Rounding algorithm uses extra space of at most $D_{\max}^k$, for each class C_k. The total amount of additional memory is bounded by $\sum_{k \geq 0} D_{\max}(1+\epsilon)^{-k} < D_{\max}(1+\epsilon)/\epsilon$.

To analyze the cost we compare the final solution to the optimal LP relaxation solution. If in the final solution a value $c(d,i)$ with $d \in B(i)$ is by γ bits higher than in the optimal LP relaxation solution, we say that d *was rounded up by an amount of* γ. This corresponds to a cost saving of $\gamma/size(d)$ when the next requests to d are served. If the value is by γ bits smaller, we say that d *was rounded down by an amount of* γ and this results in an extra cost of $\gamma/size(d)$ on the next requests to d. For any class C_k, let R_k^u be the total amount by which documents from C_k are rounded up and let R_k^d be the total amount by which documents from C_k are rounded down. The solution OPT^* and the Rounding algorithm (described in Figure 2) imply $R_k^u \geq R_k^d$. The optimal LP relaxation solution incurs a cost of $\mathrm{OPT}_{\mathrm{LP}}^{B}(\sigma) \geq \sum_{k \geq 0} R_k^u / D_{\max}^k$. Our final solution saves a cost of at least $\sum_{k \geq 0} R_k^u / D_{\max}^k$ and incurs an extra cost of at most $\sum_{k \geq 0} R_k^d (1+\epsilon)/D_{\max}^k$ since the document sizes in a class differ by a factor of at most $1+\epsilon$. This gives at most extra cost of $\epsilon \mathrm{OPT}_{\mathrm{LP}}^{B}(\delta)$. □

Corollary 3 *For any request sequence σ and any $\epsilon > 0$ we can construct a solution that incurs a cost of at most $2 + \epsilon$ times the optimum cost and uses an extra space of no more than $(1 + 2/\epsilon)D_{\max}$.*

3.5 The General Model

Due to space limitations we describe the following results in the appendix.

Theorem 5 *For any σ, we can construct a solution that processes σ in batches and incurs a cost of at most 4 times that of an optimal solution that serves σ in batches. No extra space is required.*

Corollary 4 *For any σ, we can construct a solution that incurs a cost of at most 8 times that of an optimal solution. No extra space is required.*

References

[1] D. Achlioptas, M. Chrobak and J. Noga. Competitive analysis of randomized paging algorithms. *Theoretical Computer Science*, 234:203–218, 2000.

[2] S. Albers, S. Arora and S. Khanna. Page replacement for general caching problems. *Proc. 10th Annual ACM-SIAM Symposium Discrete Algorithms*, 31–40, 1999.

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

[4] L.A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78-101, 1966.

[5] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. *Proc. USENIX Symposium on Internet Technology and Systems*, 193–206, 1997.

[6] T. Feder, R. Motwani, R. Panigrahy and A. Zhu. Web caching with request reordering. *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms*, 2002.

[7] T. Feder, R. Motwani, R. Panigrahy, S. Seiden, R. van Stee and A. Zhu. Combining request scheduling with web caching. Extended version of [6], to appear in *Theoretical Computer Science*.

[8] A. Feldmann, A. Karlin, S. Irani and S. Phillips. Private communication, transmitted through [11], 1996.

[9] A. Fiat, R.M. Karp, L.A. McGeoch, D.D. Sleator and N.E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12:685–699, 1991.

[10] P. Gopalan, H.J. Karloff, A. Mehta, M. Mihail and N. Vishnoi. Caching with expiration times. *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms*, 540-547, 2002

[11] S. Irani. Page replacement with multi-size pages and applications to Web caching. *Proc. 29th Annual ACM Symposium on Theory of Computing*, 701–710, 1997.

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

[13] D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.

[14] N.E. Young. Online file caching. *Proc. 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 82–86, 1998.

Appendix

In the General Model the service/loading cost $cost(d)$ can be an arbitrary value, for any document d. Our solution for the General Model is based on a approximation algorithm by Bar-Noy et al. [3]. They considered the following *loss minimization problem.* Let $\mathcal{I}$ be a set of intervals $I = [t_1, t_2]$, the scheduling of which requires a resource of bounded availability. Let $width(t)$ be the amount of resource available at time t. Each interval $I \in \mathcal{I}$ has a width $w(I)$ as well as a penalty $p(I)$. The width reflects the amount of resource required at any time $t \in I$ if I is scheduled. The penalty represents the cost if I is not scheduled. The goal is to find a set $\mathcal{S} \in \mathcal{I}$ of intervals being scheduled such that $\sum_{I \in \mathcal{S}} w(I) \leq width(t)$, for any time t, and the total penalty $\sum_{I \in \mathcal{I} \setminus \mathcal{S}} p(I)$ is as small as possible. Bar-Noy et al. showed that the standard web caching problem, where request reordering is not allowed and documents must be in cache at the time of the reference, can be formulated as a loss minimization problem. We show here that the problem of constructing a schedule that processes a given request sequence in batches and does not need to have referenced documents in cache can also be formulated in the above framework. As usual, for any document d and any batch i, let $b(d, i)$ be the smallest index $j > i$ such that batch j contains requests to d. For any d and i such that document d is requested in batch i, we introduce an interval $i = [i, b(d, i) - 1]$ representing the case that d resides in cache between the end of batch i and the end of batch $b(d, i) - 1$. Document d is brought into cache during the processing of batch i if it was not already present. If such an interval I is scheduled, then the requests to d in batch $b(d, i)$ can be served at 0 cost. Otherwise the cost is $cost(d)$ and we set the penalty to $p(I) = cost(d)$. The width is $w(I) = size(d)$, representing d's space requirements in cache. We have $width(t) = K$. Thus we can apply the approximation algorithm of Bar-Noy et al. and obtain the results stated in Theorem 5 and Corollary 4.