Online Compression Caching

C. Greg Plaxton[1], Yu Sun[2], Mitul Tiwari[2], and Harrick Vin[2]

*Department of Computer Science
University of Texas at Austin
plaxton@cs.utexas.edu, asun@vmware.com, mitul@kosmix.com, vin@cs.utexas.edu

Abstract. Motivated by the possibility of storing a file in a compressed format, we formulate the following class of compression caching problems. We are given a cache with a specified capacity, a certain number of compression/uncompression algorithms, and a set of files, each of which can be cached as it is or by applying one of the compression algorithms. Each compressed format of a file is specified by three parameters: encode cost, decode cost, and size. The miss penalty of a file is the cost of accessing the file if the file or any compressed format of the file is not present in the cache. The goal of a compression caching algorithm is to minimize the total cost of executing a given sequence of requests for files. We say an online algorithm is resource competitive if the algorithm is constant competitive with a constant factor resource advantage. A well-known result in the framework of competitive analysis states that the least-recently used (LRU) algorithm is resource competitive for the traditional paging problem. Since compression caching generalizes the traditional paging problem, it is natural to ask whether a resource competitive online algorithm exists or not for compression caching. In this work, we address three problems in the class of compression caching. The first problem assumes that the encode cost and decode cost associated with any format of a file are equal. For this problem we present a resource competitive online algorithm. To explore the existence of resource competitive online algorithms for compression caching with arbitrary encode costs and decode costs, we address two other natural problems in the aforementioned class, and for each of these problems, we show that there exists a non-constant lower bound on the competitive ratio of any online algorithm, even if the algorithm is given an arbitrary factor capacity blowup. Thus, we establish that there is no resource competitive algorithm for compression caching in its full generality.

1 Introduction

Recently we have seen an explosion in the amount of data distributed over hand-held devices, personal computers, and local and wide area networks. There is a

[1]Supported by NSF Grants CNS–0326001 and CCF–0635203.
[2]Supported by NSF Grant CNS–0326001 and Texas Advanced Technology Program 003658-0608-2003.
*Most of the work was done while Yu Sun and Mitul Tiwari were Ph.D. students at UT Austin.

growing need for self-tuning data management techniques that can operate under a wide range of conditions, and optimize various resources such as storage space, processing, and network bandwidth. There is a large body of work addressing different aspects of this domain of self-tuning data management.

An important aspect of this domain that merits further attention is that data can be stored in different formats. For example, one can compress a text file using different traditional compression techniques such as gzip and bzip. Various studies [1, 2, 6] have experimentally demonstrated the advantages of compression in caching. A compressed file takes up less space, effectively increasing the size of the memory. However, this increase in size comes at the cost of extra processing needed for compression and uncompression. Consequently, it may be desirable to keep frequently accessed files uncompressed in the memory.

As another example, consider the option of storing only a TEX file or the corresponding pdf file along with the TEX file. One can save space by storing only the TEX file, but one has to run a utility (such as pdflatex) to generate the pdf file when needed. On the other hand, storing the pdf file may require an order of magnitude more storage space than the TEX file, but the pdf file is readily accessible when needed. In general, many files are automatically generated using some utility such as a compiler or other translator. If the utility generates a large output compared to the input, then by storing only the input one achieves a form of "compression", not in the traditional sense, but with analogous consequences. In this paper, when we refer to compression, we have in mind this broader notion of compression where one can have a wide separation between storage space and processing costs associated with different formats of a file.

In this work, we address the notion of compression and uncompression of files, while contemplating the possibility of a richer variety of separation between the sizes and processing costs associated with the different formats of a file. We focus primarily on the single machine setting, however one of our upper bound results (see Section 3.2) is applicable to a simple, but well-motivated, special case of a distributed storage problem.

Problem Formulation. We define a class of compression caching problems in which a file can be cached in multiple compressed formats with varying sizes, and costs for compression and uncompression (see Section 2 for a formal description). We are given a cache with a specified capacity. Also assume that for each file, there are multiple associated formats. Each format is specified by three parameters: encode cost, decode cost, and size. The encode cost of a particular format is defined as the cost of creating that format from the uncompressed format of the file. The decode cost of a format is defined as the cost of creating the uncompressed format. The miss penalty of a file is defined as the cost of accessing the file if no format of the file is present in the cache. To execute a request for a file, the file is required to be loaded into the cache in the uncompressed format. The goal of a compression caching algorithm is to minimize the total cost of executing a given request sequence.

The main challenge is to design algorithms that determine — in an online manner — which files to keep in the fast memory, and of these, which to keep in compressed form. The problem is further complicated by the multiple compression formats for a file, with varying sizes and encode/decode costs. Since compression caching has the potential to be useful in many different scenarios, a desirable property of an online algorithm is to provide a good competitive ratio, which is defined as the maximum ratio of the cost of the online algorithm and that of the offline algorithm over any request sequence (see [4] for more details). We refer to an online algorithm that achieves a constant competitive ratio when given a constant factor resource advantage as a resource competitive algorithm.

In a seminal work, Sleator and Tarjan [7] show that the competitive ratio of any deterministic online paging algorithm without any capacity blowup is the size of the cache, and they also show that LRU is resource competitive for the disk paging problem. Since compression caching generalizes the disk paging problem, it is natural to ask whether similar resource competitive results can be obtained for compression caching.

Contributions. In this paper, we address three problems in the class of the compression caching. Our contributions for each of these problems are as follows.

- The first problem assumes that the encode cost and decode cost associated with any format of a file are equal. For this problem we generalize the Landlord algorithm [9] to obtain an online algorithm that is resource competitive. We find that this problem also corresponds to a special case of the distributed storage problem, and hence, our algorithm is applicable to this special case.
- The second problem assumes that the decode costs associated with different formats of a file are the same. For this problem, we show that any deterministic online algorithm (even with an arbitrary factor capacity blowup) is $\Omega(m)$-competitive, where m is the number of possible formats of a file. The proof of this lower bound result is the most technically challenging part of the paper. Further, we give an online algorithm for this problem that is $O(m)$-competitive with $O(m)$ factor capacity blowup. Thus, we tightly characterize the competitive ratio achievable for this problem.
- The third problem assumes that the encode costs associated with different formats of a file are the same. For this problem we show that any deterministic online algorithm (even with an arbitrary factor capacity blowup) has competitive ratio $\Omega(\log m)$. We also present an online algorithm for this problem that is $O(m)$-competitive with $O(m)$ factor capacity blowup.

Related Work. The competitive analysis framework was pioneered by Sleator and Tarjan [7]. For the disk paging problem, it has been shown that LRU is $\frac{k}{k-h+1}$-competitive, where k is the cache capacity of LRU and h is the cache capacity of any offline algorithm [7]. In the same paper, it has been shown that $\frac{k}{k-h+1}$ is the best possible competitive ratio for any deterministic online paging algorithm. For the variable size file caching problem, which is useful in the context of web-caching, Young [9] proposes the Landlord algorithm, and shows that Landlord is $\frac{k}{k-h+1}$-competitive. For the variable size file caching problem, Cao and Irani [5] independently propose the greedy-dual-size algorithm and show that it is k-competitive against any offline algorithm, where k is cache capacity of both greedy-dual-size and the offline algorithm. For the distributed paging problem, Awerbuch et al. [3] give an algorithm that is polylog(n, Δ)-competitive with polylog(n, Δ) factor capacity blowup, where n is the number of nodes and Δ is the diameter of the network.

Various studies [1, 2, 6] have shown experimentally that compression effectively increases on-chip and off-chip chip cache capacity, as well as off-chip bandwidth, since the compressed data is smaller in size. Further, these studies show that compression in caching increases the overall performance of the system.

Outline. The rest of this paper is organized as follows. In Section 2 we provide some definitions. In Section 3 we present our results for the compression caching problem with equal encode and decode costs. In Section 4 we describe our results for the compression caching problem with varying encode costs and uniform decode costs. In Section 5 we discuss our results for the compression caching problem with uniform encode costs and varying decode costs.

2 Preliminaries

Assume that we are given a cache with a specified capacity and m different functions for encoding and decoding any file, denoted h_i and h_i^{-1}, where $0 \leq i < m$. Without loss of generality, we assume that h_0 and h_0^{-1} are the identity functions. We define index i as an integer i such that $0 \leq i < m$. For any index i, we obtain the i-encoding of any file f by evaluating $h_i(f)$, and we obtain the file f from the i-encoding μ of f by evaluating $h_i^{-1}(\mu)$. For any file f, we refer to the 0-encoding of f as the *trivial* encoding, and for $i > 0$, we refer to the i-encoding of f as a *nontrivial* encoding. For any file f and index i, the i-encoding of f is also referred to as an encoding of f, and we say f is present in the cache if any encoding of f is present in the cache. For any file f and index i, the i-encoding of f is characterized by three parameters: encode cost, denoted $\mathit{encode}(i, f)$; decode cost, denoted $\mathit{decode}(i, f)$; and size, denoted $\mathit{size}(i, f)$. The encode cost $\mathit{encode}(i, f)$ is defined as the cost of evaluating $h_i(f)$, and the decode cost $\mathit{decode}(i, f)$ is defined as the cost of evaluating $h_i^{-1}(\mu)$, where μ is the i-encoding of f. For any file f, $\mathit{encode}(0, f)$ and $\mathit{decode}(0, f)$ are 0.

For any file f, the *access cost* of f is defined as follows: if for some index i, the i-encoding of f is present in the cache (break ties by picking minimum such i), then the access cost is $\mathit{decode}(i, f)$; if none of the encodings of f is present in the cache, then the access cost is defined as the miss penalty $p(f)$. Without loss of generality, we assume that the miss penalty for any file f is at least the decode cost of any of the encodings of f. The cost of deleting any encoding of any file from the cache is 0. For any file f and index i, the i-encoding of f can be added to the cache if there is enough free space to store the i-encoding of f. For any file f and index i, the cost of adding the i-encoding of f to the cache is the sum of the access cost of f and $\mathit{encode}(i, f)$.

To execute a request for a file f, an algorithm A is allowed to modify its cache content by adding/deleting encodings of files, and then incurs the access cost for f. The goal of the compression caching problem is to minimize the total cost of executing a given request sequence. An online compression caching algorithm A

is c-competitive if for all request sequences τ and compression caching algorithms B, the cost of executing τ by A is at most c times that of executing τ by B.

Any instance I of the compression caching problem is represented by a triple (σ, m, k), where σ is the sequence of request for the instance I, m is the number of possible encodings for files in σ, and k is the cache capacity. For any instance $I = (\sigma, m, k)$, we define $\mathit{reqseq}(I) = \sigma$, $\mathit{numindex}(I) = m$, and $\mathit{space}(I) = k$.

We define a *configuration* as a set of encodings of files. For any configuration S, we define the size of S as the sum, over all encodings μ in S, of size of μ. We define a *trace* as a sequence of pairs, where the first element of the pair is a request for a file and the second element of the pair is a configuration. For any configuration S and any integer k, S is *k-feasible* if the size of S is at most k. For any trace T and integer k, T is k-feasible if and only if any configuration in T is k-feasible. For any two sequences τ and τ', we define $\tau \circ \tau'$ as the sequence obtained by appending τ' to τ. For any trace T, we define $\mathit{requests}(T)$ as the sequence of requests present in T, in the same order as in T.

For any file f, any trace T, and any configuration S, we define $\mathit{cost}_f(T, S)$ inductively as follows. If T is empty, then $\mathit{cost}_f(T, S)$ is zero. If T is equal to $\langle (f', S') \rangle \circ T'$, then $\mathit{cost}_f(T, S)$ is $\mathit{cost}_f(T', S')$ plus the sum, over all i-encodings μ of f such that μ is present in S' and μ is not present in S, of $\mathit{encode}(i, f)$, plus the access cost of f in S if $f = f'$. For any file f and any trace T, we define $\mathit{cost}_f(T)$ as $\mathit{cost}_f(T, \emptyset)$. For any trace T and any configuration S, we define $\mathit{cost}(T, S)$ as the sum, over all files f, of $\mathit{cost}_f(T, S)$. For any trace T, we define $\mathit{cost}(T)$ as $\mathit{cost}(T, \emptyset)$.

3 Equal Encode and Decode Costs

In this section, we consider a symmetric instance of the compression caching problem which assumes that the encode cost and decode cost associated with any encoding of a file are equal. We present an online algorithm for this problem, and show that the algorithm is resource competitive. Interestingly, this problem also corresponds to a multilevel storage scenario, as discussed in Section 3.2.

For the restricted version of the compression caching problem considered in this section, we have $\mathit{encode}(i, f) = \mathit{decode}(i, f)$ for any file f and index i. At the expense of a small constant factor in the competitive ratio, we can assume that, for any file f, the miss penalty $p(f)$ is at least $q \cdot \mathit{encode}(m-1, f)$, where $q > 1$; and by preprocessing, we can arrange encodings of files in geometrically decreasing sizes and geometrically increasing encode-decode costs. The basic idea behind the preprocessing phase is as follows. First, consider any two encodings with sizes (resp., similar encode-decode costs) within a constant factor. Second, from these two encodings, select the one with smaller encode-decode cost (resp., smaller size), and eliminate the other. While an encoding can be eliminated by one of the above preprocessing steps, we do so. After the above preprocessing phase, we can arrange the encodings of files in geometrically decreasing sizes and geometrically increasing encode-decode costs.

For ease of presentation, we assume that m encodings are selected for each file in the preprocessing phase. More precisely, after the preprocessing phase, for any file f and index $i < m-1$, we have $\mathit{size}(i+1, f) \leq \frac{1}{r} \cdot \mathit{size}(i, f)$ and $\mathit{encode}(i+1, f) \geq q \cdot \mathit{encode}(i, f)$, where $r > 1$. Also, we assume that the capacity of the cache given to an online algorithm is b times that given to an offline algorithm.

3.1 Algorithm

In Figure 1, we present our online algorithm ON. At a high level, ON is a credit-rental algorithm. Algorithm ON maintains a containment property on the encodings in the cache, defined as follows: If ON has the i-encoding of some file f in the cache, then ON also has all the j-encodings of f for any index $j \geq i$ in the cache. A credit is associated with each encoding present in the cache. For any file f and index i, the i-encoding of f is created with an initial credit $\mathit{decode}(i+1, f)$, for $i < m-1$, and credit $p(f)$, for $i = m-1$. On a request for a file f, if the 0-encoding of f is not present in the cache, then ON creates space for the 0-encoding of f, and for other i-encodings of f that are necessary to maintain the containment property. Then, ON creates the 0-encoding of f, and any other i-encodings of f that are necessary to maintain the containment property, with an initial credit as described above. In order to create space, for each file present in the cache, ON charges rent from the credit of the largest encoding of the file, where rent charged is proportional to the size of the encoding, and deletes any encoding with 0 credit. The credit-rental algorithm described here can be viewed as a generalization of Young's Landlord algorithm [9].

We use a potential function argument similar to that of Young to show that ON is resource competitive. See [8, Section 3.3.2] for the complete proof of the following theorem.

Theorem 1. *Algorithm* ON *is resource competitive for any symmetric instance of the compression caching problem.*

3.2 Multilevel Storage

Consider an outsourced storage service scenario (for simplicity, here we describe the problem for a single user) where we have multiple levels of storage. Each storage space is specified by two parameters: storage cost and access latency to the user. The user specifies a fixed overall budget to buy storage space at the various levels, and generates requests for files. The goal is to manage the user budget and minimize the total latency incurred in processing a given request sequence. Our credit-rental algorithm for the compression caching problem with equal encode and decode costs can be easily generalized to this scenario, and we can show (using a similar analysis as above) that the generalized algorithm is constant competitive with a constant factor advantage in the budget for the aforementioned multilevel storage problem.

```
{Initially, for any encoding μ of any file, credit(μ) = 0}
On a request for a file f
if f is not present in the cache then
  createspace(f, m − 1)
  for all indices i, add the i-encoding μ of f, with credit(μ) := decode(i + 1, f), if i < m − 1,
    and with credit(μ) := p(f), if i = m − 1
else if the i-encoding μ of f is present in the cache (break ties by picking the minimum such i) then
  evaluate h_i^{-1}(μ)
  credit(μ) := decode(i + 1, f)
  if i > 0 then
    createspace(f, i − 1)
    for all indices j < i, add the j-encoding ν of f, with credit(ν) := decode(j + 1, f)
  fi
fi

createspace(f, i)
  sz := Σ_{j=0}^{i} size(f, j)
  while free space in the cache < sz do
    δ := min_{μ∈X} credit(μ)/size(j, f′), where μ is the j-encoding of f′
    for each file f′ such that there is an encoding of f′ in the cache do
      let μ be the largest (in size) encoding of f′ in the cache
      let j be the index of μ
      credit(μ) := credit(μ) − δ · size(j, f′)
      if credit(μ) = 0 then
        delete μ
      fi
    od
  od
```

Fig. 1. The online algorithm ON for any symmetric instance of the compression caching problem. Here, X is the cache content of ON.

4 Varying Encode Costs and Uniform Decode Costs

We say that an instance $I = (\sigma, m, k)$ of the compression caching problem is a uniform-decode instance if any file in σ satisfies the following properties. First, we consider that the decode cost associated with different encodings of any file in σ are the same; for any file f and any index $i > 0$, we abbreviate $decode(i, f)$ to $decode$. Second, we consider that for any index i, any file f and f' in σ, $size(i, f) = size(i, f')$, $p(f) = p(f')$, and $encode(i, f) = encode(i, f')$. For the sake of brevity, we write $encode(i, f)$ as $encode(i)$.

We formulate this problem to explore the existence of resource competitive algorithms for the problems in the class of compression caching. This problem is also motivated by the existence of multiple formats of a multimedia file with varying sizes and encode costs, and with roughly similar decode costs.

One might hope to generalize existing algorithms like Landlord for this problem, and to achieve resource competitiveness. However, in this section we show that any deterministic online algorithm (even with an arbitrary factor capacity blowup) for this problem is $\Omega(m)$-competitive, where m is the number of possible encodings of each file. We also give an online algorithm for this problem that is $O(m)$-competitive with $O(m)$ factor capacity blowup.

4.1 The Lower Bound

In this section we show that any deterministic online algorithm (even with an arbitrary factor capacity blowup) for any uniform-decode instance of the compression caching problem is $\Omega(m)$-competitive.

For any algorithm A, any request sequence σ, and any real number k, we define $config(A, \sigma, k)$ as the configuration of A after executing σ with a cache of size k, starting with an empty configuration.

Informal overview At a high level, the adversarial request generating algorithm *Adversary* works recursively as follows. For a given online algorithm ON, a given number of encodings for a file m, a given cache capacity of the offline algorithm k, and a blowup b, the algorithm $Adversary(\mathrm{ON}, m, k, b)$ picks a set of files X such that any file in X is not in ON's cache, and invokes a recursive request generating procedure $AdversaryHelper(X, i, \sigma, \mathrm{ON}, k, b)$, where initially $|X|$ is the number of $(m-1)$-encodings that can fit in a cache of size k, $i = m-1$, and σ is empty. This procedure returns a trace of the offline algorithm OFF. (See Section 4.1 for formal definitions and a description of the algorithm.)

Consider an invocation of procedure $AdversaryHelper(X, i, \sigma, \mathrm{ON}, k, b)$. The adversary picks a subset of the files Y from X such that any file f in Y satisfies certain conditions. For $i > 1$, if Y contains sufficiently many files, then the adversary invokes $AdversaryHelper(Y, i-1, \sigma', \mathrm{ON}, k, b)$, where σ' is the request sequence generated; otherwise, $AdversaryHelper(X, i, \sigma, \mathrm{ON}, k, b)$ is terminated. For $i = 1$, the adversary picks a file f in Y, and repeatedly generates requests for f until either ON adds an encoding of f to its cache, or a certain number of requests for f are generated. Finally, $AdversaryHelper(X, 1, \sigma, \mathrm{ON}, k, b)$ terminates when Y is empty.

At a high level, the offline algorithm OFF works as follows. Algorithm OFF decides the encodings for the files in X when $AdversaryHelper(X, i, \sigma, \mathrm{ON}, k, b)$ terminates. For any index $j \geq i$, if ON adds the j-encodings of less than a certain fraction of files in X any time during the execution of the request sequence generated by $AdversaryHelper(X, i, \sigma, \mathrm{ON}, k, b)$, then OFF adds the i-encodings of all the files in X, and incurs no miss penalties in executing the request sequence generated by $AdversaryHelper(X, i, \sigma, \mathrm{ON}, k, b)$. Otherwise, OFF returns the concatenation of the traces generated during the execution of $AdversaryHelper(X, i, \sigma, \mathrm{ON}, k, b)$. By adding the j-encodings of a certain fraction of files in X, ON incurs much higher cost than OFF in executing the request sequence generated by $AdversaryHelper(X, i, \sigma, \mathrm{ON}, k, b)$.

Using an inductive argument, we show that ON is $\Omega(m)$-competitive for the compression caching problem with varying encode and uniform decode costs.

Adversarial request generating algorithm Some key notations used in the adversarial request generating algorithm are as follows.

For any file f and any real number b, $eligible(f, m, b)$ holds if the following conditions hold: (1) for any index i, $size(i, f) = r^{m-i-1}$, where $r = 8b$; (2) $p(f) =$

p; (3) for any index i, $encode(i, f) = p \cdot q^i$, where $q = \frac{m}{20}$; and (4) $decode = 0$. The number of i-encodings of files that can fit in a cache of size k is denoted $num(k, i)$. Note that, for eligible files, $num(k, i)$ is equal to $r \cdot num(k, i - 1)$.

For any algorithm A, any request sequence σ, any real number k, any file f, and any index i, we define a predicate $aggressive(A, \sigma, k, f, i)$ as follows. If σ is empty, then $aggressive(A, \sigma, k, f, i)$ does not hold. If σ is equal to $\sigma' \circ \langle f' \rangle$, then $aggressive(A, \sigma, k, f, i)$ holds if either $aggressive(A, \sigma', k, f, i)$ holds or, for some index $j \geq i$, the j-encoding of f is present in $config(\text{ON}, \sigma, k)$.

For any request sequence σ, and any index i, any set of files X, we define $trace(\sigma, i, X)$ as follows. If σ is empty, then $trace(\sigma, i, X)$ is empty. If σ is equal to $\sigma' \circ \langle f \rangle$, then $trace(\sigma, i, X)$ is $trace(\sigma', i, X) \circ \langle (f, Y) \rangle$, where Y is the set of the i-encodings of files in X.

In Figure 2 we describe the adversarial request generating algorithm $Adversary$. The caching decisions of the offline algorithm OFF are given by the trace T gen-

```
Adversary(ON, m, k, b)
T := ∅
while |T| < N do
  X := set of num(k, m − 1) files f such that (1) eligible(f, m, b) holds; and
     (2) f is not present in config(ON, requests(T), bk)
  T := T ∘ AdversaryHelper(X, m − 1, requests(T), ON, k, b)
od
return T

AdversaryHelper(X, i, σ, ON, k, b)
T, σ' := ∅, ∅
Y := X
repeat
  if i = 1 then
     Let f be an arbitrary file in Y
     count := 0
     repeat
       σ' := σ' ∘ ⟨f⟩
       S := config(ON, σ ∘ requests(T) ∘ σ', bk)
       count := count + 1
     until f is not present in S or count ≥ 8
     T := T ∘ trace(σ', 0, {f})
     σ' := ∅
  else
     X' := arbitrary subset of num(k, i − 1) files in Y
     T' := AdversaryHelper(X', i − 1, σ ∘ requests(T), ON, k, b)
     T := T ∘ T'
  fi
  reassign Y as follows: for any file f, f is in Y if and only if (1) f is in X
     (2) f is not present in config(ON, σ ∘ requests(T), bk);
     (3) cost_f(T) < (8 · e_i − 8 · e_{i−1}); and
     (4) aggressive(ON, requests(T), bk, f, i) does not hold
until (i = 1 and |Y| = ∅) or (|Y| < num(k, i − 1))
if |{f ∈ X | aggressive(ON, requests(T), bk, f, i)}| < 2b · num(k, i − 1) then
  T := trace(requests(T), i, X)
fi
return T
```

Fig. 2. The adversarial request generating algorithm for the compression caching problem with varying encode and uniform decode costs. Here, N is the number of requests to be generated.

erated during the execution of *Adversary* (Figure 2). See [8, Section 3.4.1.3] for the proof of the following theorem.

Theorem 2. *Any deterministic online algorithm with an arbitrary factor capacity blowup is $\Omega(m)$-competitive for any uniform-decode instance I of the compression caching problem, where $m = numindex(I)$.*

4.2 An Upper Bound

In this section we present an online algorithm that is $O(m)$-competitive with $O(m)$ factor capacity blowup for any uniform-decode instance I of the compression caching problem, where $m = numindex(I)$. As in Section 3, by preprocessing, we can arrange the encodings of files in decreasing sizes and increasing encode costs; that is, after preprocessing, for any file f and any index $i < m-1$, $size(i+1, f) \leq \frac{1}{r} size(i, f)$, and $encode(i+1, f) \geq q \cdot encode(i, f)$, where $r = 1+\epsilon$, $q = 1 + \epsilon'$, $\epsilon > 0$, and $\epsilon' > 0$.

Algorithm ON divides its cache into m blocks. For any index i, block i keeps only the i-encodings of files. For any integer k and index i, let $num(k, i)$ be the maximum number of i-encodings of files that can fit in any block of size k.

Roughly speaking, ON works as follows. For any index i, ON adds the i-encoding of a file f after the miss penalties incurred by ON on f sum to at least $encode(i, f)$. We use a standard marking algorithm as an eviction procedure for each block. The complete description of ON is presented in [8, Figure 3.3].

See [8, Section 3.4.2.2] for the proof of the following theorem.

Theorem 3. *For any uniform-decode instance I of the compression caching problem, there exists an online algorithm that is is $O(m)$-competitive with $O(m)$ factor capacity blowup, where $m = numindex(I)$.*

5 Uniform Encode Costs and Varying Decode Costs

We say that an instance $I(\sigma, m, k)$ of the compression caching problem is a uniform-encode instance if any file in σ satisfies the following properties. First, we consider that the encode costs of all the nontrivial encodings of any file f in σ are the same; for any index $i > 0$, we abbreviate $encode(i, f)$ to $encode$. Second, we consider that for any index i, any file f and f' in σ, $size(i, f) = size(i, f')$, $p(f) = p(f')$, and $decode(i, f) = decode(i, f')$. For the sake of brevity, for any file f in σ, we write $decode(i, f)$ as $decode(i)$.

In this section we show that any deterministic online algorithm (even with an arbitrary factor capacity blowup) for any uniform-encode instance of the compression caching problem is $\Omega(\log m)$-competitive, where m is the number of possible encodings for each file. Further, we present an online algorithm for this problem that is $O(m)$-competitive with $O(m)$ factor capacity blowup.

5.1 The Lower Bound

In this section, we show that any deterministic online algorithm (even with an arbitrary capacity blowup) for any uniform-encode instance of the compression caching problem is $\Omega(\log m)$-competitive.

For any given online algorithm ON with a capacity blowup b, we construct a uniform-encode instance of the compression caching problem. For any file f and index $i < m - 1$, we consider that $\mathit{size}(i+1, f) \leq \frac{1}{r} \cdot \mathit{size}(i, f)$, where $r > b$. For any file f and index i such that $0 < i < m - 1$, we consider that $\mathit{decode}(i+1, f) \geq \mathit{decode}(i, f) \cdot \log m$. We also set the miss penalty $p(f)$ to be encode, and $\mathit{encode} \geq \mathit{decode}(m-1, f) \cdot \log m$.

Adversarial request generating algorithm Our adversarial request generating algorithm ADV takes ON as input, and generates a request sequence σ and an offline algorithm OFF such that ON incurs at least $\log m$ times the cost incurred by OFF in executing σ. For any file f, ADV maintains two indices denoted $w_u(f)$ and $w_\ell(f)$; initially, $w_u(f) = m$ and $w_\ell(f) = 0$. The complete description of ADV is presented in [8, Figure 3.5].

Roughly, ADV operates as follows. Algorithm ADV forces ON to do a search over the encodings of a file to find the encoding that OFF has chosen for that file. Before any request is generated, ADV ensures that for any f, there is no i-encoding of f in ON's cache such that $w_\ell(f) \leq i < w_u(f)$. On a request for any file f, if ON adds the i-encoding of f such that $w_\ell(f) \leq i < w_u(f)$, then ADV readjusts $w_\ell(f)$ and $w_u(f)$ to ensure that the above condition is satisfied. If ON does not keep the i-encoding of f such that $i < w_u(f)$, then ADV continues to generate requests for f. Finally, when $w_u(f) = w_\ell(f)$, OFF claims that OFF has kept the i-encoding of f, where $i = w_\ell(f)$, throughout this process, and has executed the requests for f. Then, ADV resets the variables $w_u(f)$ and $w_\ell(f)$ to m and 0, respectively, and OFF deletes the encoding of f from its cache. In this process, OFF incurs encoding cost of adding only one encoding of file f. On the other hand ON incurs much higher cost than OFF because of adding multiple encodings of f. See [8, Section 3.5.1.2] for the proof of the following theorem.

Theorem 4. *Any deterministic online algorithm with an arbitrary factor capacity blowup is $\Omega(\log m)$-competitive for any uniform-encode instance of the compression caching problem.*

5.2 An Upper Bound

In this section we present an $O(m)$-competitive online algorithm ON with $O(m)$ factor capacity blowup for any uniform-encode instance I of the compression caching problem, where $m = \mathit{numindex}(I)$.

As in Section 3.1, by preprocessing, we can arrange the encodings of the files in such a way that sizes are decreasing and decode costs are increasing. In other words, after preprocessing, for any file f and index $i < m - 1$, $\mathit{size}(i +$

$1, f) < size(i, f)$, and $decode(i+1, f) > decode(i, f)$. Recall that for any file f, $decode(m-1, f) < p(f)$.

For any uniform-encode instance $I = (\sigma, m, k)$, the online algorithm ON is given a $2bm$ factor capacity blowup, where b is at least $1+\epsilon$ for some constant $\epsilon > 0$. We divide ON's cache into $2m$ blocks, denoted i-*left* and i-*right*, $0 \leq i < m$, such that the capacity of each block is bk. For any index i, i-*left* keeps only the i-encodings of files, and i-*right* keeps only the 0-encodings of files. For any file f and index i, we maintain an associated value $charge(f, i)$. Roughly, whenever the cost incurred in miss penalties or decode costs on a file f exceeds *encode*, then ON adds an encoding of the file that is cheaper in terms of the access cost than the current encoding (if any) of f. The complete description of algorithm ON is given in [8, Figure 3.6].

See [8, Section 3.5.2.2] for the proof of the following theorem.

Theorem 5. *For any uniform-encode instance I of the compression caching problem, there exists an online algorithm that is $O(m)$-competitive with $O(m)$ factor capacity blowup, where $m = numindex(I)$.*

References

1. B. Abali, M. Banikazemi, X. Shen, H. Franke, D. E. Poff, and T. B. Smith. Hardware compressed main memory: Operating system support and performance evaluation. *IEEE Transactions on Computers*, 50:1219–1233, 2001.
2. A. R. Alameldeen and D. A. Wood. Adaptive cache compression for high-performance processors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 212–223, June 2004.
3. B. Awerbuch, Y. Bartal, and A. Fiat. Distributed paging for general networks. *Journal of Algorithms*, 28:67–104, 1998.
4. A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, Cambridge, 1998.
5. P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1st Usenix Symposium on Internet Technologies and Systems*, pages 193–206, December 1997.
6. E. G. Hallnor and S. K. Reinhardt. A unified compressed memory hierarchy. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 201–212, February 2005.
7. D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.
8. M. Tiwari. Algorithms for distributed caching and aggregation. `http://www.cs.utexas.edu/users/plaxton/pubs/dissertations/mitul.pdf`, 2007.
9. N. E. Young. On-line file caching. *Algorithmica*, 33:371–383, 2002.