Online file caching with rejection penalties*

Leah Epstein[†] Csanád Imreh[‡] Asaf Levin[§] Judit Nagy-György[¶]

Abstract

In the file caching problem, the input is a sequence of requests for files out of a slow memory. A file has two attributes, a positive retrieval cost and an integer size. An algorithm is required to maintain a cache of size k such that the total size of files stored in the cache never exceeds k. Given a request for a file that is not present in the cache at the time of request, the file must be brought from the slow memory into the cache, possibly evicting other files from the cache. This incurs a cost equal to the retrieval cost of the requested file. Well-known special cases include paging (all costs and sizes are equal to 1), the cost model, which is also known as weighted paging, (all sizes are equal to 1), the fault model (all costs are equal to 1), and the bit model (the cost of a file is equal to its size). If bypassing is allowed, a miss for a file still results in an access to this file in the slow memory, but its subsequent insertion into the cache is optional.

We study a new online variant of caching, called *caching with rejection*. In this variant, each request for a file has a rejection penalty associated with the request. The penalty of a request is given to the algorithm together with the request. When a file that is not present in the cache is requested, the algorithm must either bring the file into the cache, paying the retrieval cost of the file, or reject the file, paying the rejection penalty of the request. The objective function is the sum of total rejection penalty and the total retrieval cost. This problem generalizes both caching and caching with bypassing.

We design deterministic and randomized algorithms for this problem. The competitive ratio of the randomized algorithm is $O(\log k)$, and this is optimal up to a constant factor. In the deterministic case, a k-competitive algorithm for caching, and a $(k+1)$-competitive algorithm for caching with bypassing are known. Moreover, these are the best possible competitive ratios. In contrast, we present a lower bound of $2k+1$ on the competitive ratio of any deterministic algorithm for the variant with rejection. The lower bound is valid already for paging. We design a $(2k+2)$-competitive algorithm for caching with rejection. We also design a different $(2k+1)$-competitive algorithm, that can be used for paging and for caching in the bit and fault models.

1 Introduction

Paging and caching problems [35, 16, 26, 37] are fundamental optimization problems, with multiple applications such as operating systems and the Internet. Such problems deal with page replacement policies in two-level memory systems, that consist of a small and fast memory (often referred to as *cache*) and a large and slow main memory. Both memory levels are partitioned into slots of size 1, and they can accommodate memory units. The cache consists of k such slots.

*An extended abstract of this paper appears in Proc. of ICALP 2011 as "On variants of file caching". This research was partially supported by the TÁMOP-4.2.2/08/1/2008-0008 program of the Hungarian National Development Agency.

[†]Department of Mathematics, University of Haifa, 31905 Haifa, Israel. lea@math.haifa.ac.il

[‡]Department of Informatics, University of Szeged, 6720 Szeged, Hungary. cimreh@inf.u-szeged.hu. Supported by the Bolyai Scholarship of the Hungarian Academy of Sciences.

[§]Faculty of Industrial Engineering and Management, The Technion, 32000 Haifa, Israel. levinas@ie.technion.ac.il

[¶]Department of Mathematics, University of Szeged, Aradi Vértanúk tere 1, H-6720 Szeged, Hungary Nagy-Gyorgy@math.u-szeged.hu

The general problem, called *file caching* (or caching), is defined as follows. Files are bundles of memory units (we use the term pages for these units). A file f has two attributes, that are its (integer) size $size(f)$ (number of pages), and its cost $cost(f) > 0$ (the cost of accessing it in the main memory)[1]. In most applications, this cost is proportional to the time that is required to access it, while the cost of accessing a file residing in the cache is seen as zero, since this time is negligible compared to the time required to access the main memory.

The input is a sequence of requests for files. If an algorithm is given a request for a file that it has stored in the cache, then the algorithm does nothing (in terms of modifying the contents of the cache, it may still do some calculations to be used in the future). However, a request for a file f that is not present in the cache requires an access to the main memory, resulting in a cost of $cost(f)$ for the algorithm. This situation is called a *miss*. In the standard model (also called the *forced* model), the algorithm must insert the file into the cache, possibly evicting other files to create sufficient space for the requested file. An additional model was studied. This model is called caching with bypassing [3, 13, 32], and it is also known as the *optional* model [26, 18] (or the optional-fetch model [17]). In this model, the algorithm must read the requested file f from the main memory for a cost of $cost(f)$, but it does not necessarily need to insert it into the cache, though it may do so.

A number of important special cases were studied. The first study was the *paging* variant [11, 35]. In this case a file is also referred to as a *page*. For every page f, $size(f) = cost(f) = 1$. That is, the sequence consists of single pages, and the system is uniform in the sense that reading any page from the main memory results in the same cost. Three common models that generalize paging and are special cases of caching are the *bit model*, the *fault model* and *cost model* (also called weighted paging, and sometimes weighted caching) [4]. In the first two models, files may have arbitrary sizes, while in the third model, all files have size 1 (but arbitrary costs). In the bit model, each file f has $cost(f) = size(f)$, while in the fault model, each file f satisfies $cost(f) = 1$. That is, the fault model assumes that the cost of accessing the main memory multiple times is determined by the number of accesses, while the bit model assumes that what determines the cost is the total size of files read from the main memory.

We study online variants of caching and use competitive analysis. In the online problem the algorithm is presented with one request at each time, and it must respond to the current request before seeing the next one. For caching problems, it is common to use the asymptotic competitive ratio, and we adopt this approach [35]. For an online problem, we let $\textsc{opt}$ denote a specific optimal offline algorithm that knows the entire input. We always assume (without loss of generality) that $\textsc{opt}$ is lazy, that is, it does not do any action unless they are necessary for processing the input. For caching problems, a lazy algorithm only inserts a file into the cache if a miss for this file occurs, and it only evicts a file in order to make room for another file. For an algorithm $\mathcal{A}$, we let $\mathcal{A}(I)$ (or $\mathcal{A}$, if I is clear from the context) denote the cost of $\mathcal{A}$ on an input I. An online algorithm $\textsc{alg}$ is $\mathcal{R}$-competitive, if there exists a constant c (independent of I) that satisfies $\textsc{alg}(I) \leq \mathcal{R} \cdot \textsc{opt}(I) + c$ for any input I. The infimum value $\mathcal{R}$ for which this inequality can be satisfied for all inputs and some value c is called the competitive ratio of $\textsc{alg}$. For a randomized algorithm $\textsc{alg}$, the definition is similar, but $\textsc{alg}(I)$ is replaced with its expected value $E(\textsc{alg}(I))$.

In this paper, we initiate the study of caching with rejection. In this natural variant of caching each miss can be treated in two ways. The first way is the traditional way, where the file is brought from the main memory. The second one is by declining the request, notifying the user that the request cannot be carried out, and paying a positive rejection penalty[2]. In this variant, if the i-th request is for file f, then this request

[1]It is possible to relax this condition and allow non-negative costs instead. This does not change the results as explained in the Appendix, see Claim 14.

[2]Requests of zero rejection penalty can obviously be always rejected without incurring any cost, and thus we assume that no such requests exist.

is a pair (f, r_i), where r_i is the rejection penalty of this request. Note that r_i is not a property of f, and the rejection penalty of one file can differ for different requests for this file[3]. The cost of an algorithm is the sum of the total rejection penalty of rejected requests, and the total cost of served requests. Caching with rejection generalizes caching, since the latter is the special case where the rejection penalty is very large (e.g., twice the total cost of all files in the slow memory, see Appendix, Claim 13). Caching with rejection also generalizes caching with bypassing as the later problem is the special case where the rejection penalty for every request (f, r_i) satisfies $r_i = cost(f)$. Similarly, for each one of the specific models, we can define caching (or paging) with rejection in this model, generalizing caching in this model with or without bypassing.

The offline variant of paging problem is polynomially solvable [11], and caching in the cost model can be solved using network flow methods [16]. We note that these last techniques can be extended for the cost model with bypassing or with rejection. The other variants are strongly NP-hard [18] with or without bypassing, implying NP-hardness for the variants with rejection. Though caching problems come from real-time applications, offline approximation algorithms are of interest [26, 4, 8].

Notation. Throughout the paper, we assume that the cache is empty before any input requests have arrived. For a set of files S, let $size(S) = \sum_{g \in S} size(g)$. Let n denote the number of files in the slow memory. An input for caching with or without bypassing is represented by a sequence of file names (the arrival times of the requests are their positions in the sequence). Similarly, an input for caching with rejection is represented by a sequence of pairs, each consisting of a file name and an additional attribute that is the rejection penalty of the request. For an input I that consists of a list of requests for files (without the option of rejection), we use $\text{OPT}_s(I)$ (or OPT_s) to denote the cost of an optimal offline algorithm that does not use bypassing and similarly $\text{OPT}_b(I)$ (or OPT_b) denotes this cost if bypassing is allowed. In the Appendix we discuss the relation between the two problems. A simple modification of paging algorithms shows that for paging, $\text{OPT}_b(I) \leq \text{OPT}_s(I) \leq 2 \cdot \text{OPT}_b(I)$ (see Appendix and [5]). However, such a relation does not hold for any of the more general variants. In the Appendix we present a reduction, which we will use here, that shows that these problems (caching with and without bypassing) are in fact related in a weaker (but still useful) way.

Previous work. Sleator and Tarjan [35] were the first to consider online paging and showed that the two natural paging algorithms, namely, LRU and FIFO, have competitive ratios of at most k. This bound is the best possible result for any deterministic online algorithm (see [30]). A class of algorithms, called MARKING ALGORITHMS, was later shown by Karlin et al. [27] to achieve the same bound.

The study of randomized algorithms for paging was started by Fiat et al. [24]. An algorithm, called *Randomized Marking* was shown to perform much better than the deterministic algorithms. The competitive ratio of the randomized marking algorithm was shown to be at most $2H_k$, where H_k is the k-th harmonic number (later it was proved [1] that its tight competitive ratio is $2H_k - 1$). A lower bound of H_k on the competitive ratio of any randomized algorithm was given in [24] as well. Thus, the order of growth of the best competitive ratio is logarithmic in k. Several different algorithms of competitive ratio H_k are known [31, 1, 10, 14].

For the variant with bypassing, the results for paging are similar. In the deterministic case, the best competitive ratio is $\Theta(k)$ and in the randomized case the best competitive ratio is $\Theta(\log k)$. More specifically, it was mentioned in [26] that the best deterministic competitive ratio is $k + 1$, and it is achieved by the same algorithms that give the upper bound of k for paging without bypassing. Thus, a lower bound of $\Omega(\log k)$

[3]Clearly, all upper bounds hold also for the case that every file has a fixed rejection penalty. The lower bound that we will show in this paper has uniform rejection penalties for all requests, and thus it will be valid for this model too. It can be seen that lower bounds following from previous work (since caching with rejection generalizes caching and caching with bypassing) also apply to the model where the rejection penalty is a property of a file.

on the competitive ratio of randomized algorithms for paging with bypassing follows from the lower bound of [24] and the relation stated above between optimal solutions for paging with and without bypassing, and an upper bound of $O(\log k)$ simply follows from the existence of an $O(\log k)$-competitive algorithm for paging without bypassing (and the same relation).

Once these results were known, it was an intriguing question to find which generalizations of paging allow similar results. In the deterministic variant, the tight bound of k on the competitive ratio holds for all generalizations without bypassing. The result for weighted paging was shown by Chrobak et al. [16] (see also [36]). The result for caching was proved independently by Young [37] and by Cao and Irani [15] (the algorithm of Young [37] is in fact a class of algorithms, where the algorithm of [15] is a special case of this class that generalizes the GREEDY DUAL algorithm of [36]). For the variant with bypassing, Irani [26] mentioned that LRU has a competitive ratio of $k + 1$ for the bit and fault models. Cohen and Kaplan [19] adapted the GREEDY DUAL algorithm [36, 15, 37] for the case with bypassing and showed that it is $(k + 1)$-competitive (see also Koufogiannakis and Young [29]). Thus, for both variants, with and without bypassing, the general variant of caching admits the same competitive ratio as the basic paging problem.

On the other hand, for the randomized variant, the progress was slower. Irani [26] designed algorithms of competitive ratio $O(\log^2 k)$ for the bit model, and the fault model, that are valid both with and without bypassing. A few years later, Bansal, Buchbinder, and Naor, applying a novel usage of the online primal-dual method, designed an algorithm of competitive ratio $O(\log k)$ for weighted paging [6] (see also [5]), and algorithms with similar upper bounds on the competitive ratio for the bit model and the fault model [7]. For the most general case (that is, for caching), the upper bound of [7] is $O(\log^2 k)$. Most recently, Adamaszek et al. [2] presented an $O(\log k)$-competitive algorithm for this general case. The papers [6, 7, 2] exploit clever techniques. However, these techniques cannot be applied directly to instances of caching with bypassing.

In *finely competitive paging*, an optimal offline algorithm can either serve a new request for a page by fetching it into the cache, or rent (reject, in our terminology) the page for a cost of $\frac{1}{r}$, where $r \geq 1$ is a given parameter. The online algorithm that is compared to this offline algorithm cannot rent the page and must fetch it into the cache if bypassing is not allowed, and otherwise, if bypassing is allowed, it can rent (bypass) the page for a cost of 1. Blum et al. [12] showed a randomized $O(r + \log k)$-competitive algorithm for these problems (the randomized marking algorithm gives a competitive ratio of $O(r \log k)$). Bansal et al. [5] improved this result and showed a $r + O(\log k)$-competitive algorithm for the case with bypassing (this also implies an improved dependence on r for the case without bypassing). In our model (unlike finely competitive paging), the cost of not serving a request (rejection penalty, or the cost of bypassing) is the same for both the offline algorithm and the online algorithm. Thus, the case $r = 1$ of finely competitive paging is equivalent to paging with bypassing. The results of [12, 5] can also be used as randomized algorithms of competitive ratio $O(\log k)$ for paging with bypassing.

Many online problems were studied with rejection. This includes scheduling [9, 34, 23, 33], bin packing [20, 21], and coloring [22]. Online problems with rejection are typically very different from the same problems without rejection. Problems with rejection are frequently studied in offline scenarios, where they are sometimes called prize-collecting problems. The primal-dual method for obtaining offline approximation algorithms for a problem extends naturally to the corresponding prize-collecting problem (the approximation guarantee may degrade). For example, Goemans and Williamson in their seminal work [25] considered the prize-collecting TSP and the prize collecting Steiner tree problem.

Our results. Recall that the best possible competitive ratio for the deterministic variant of caching with bypassing is $k + 1$ [26, 19]. We prove a lower bound of $2k + 1$ on the competitive ratio of any deterministic algorithm for paging with rejection (in Section 2) and design two deterministic algorithms (in Section 4). The first one is $(2k + 2)$-competitive, and works for the most general setting, i.e., for caching (see Section

4.1). Using methods related to the SKI-RENTAL problem [27, 28], we develop (in Section 3) a general reduction of the any variant of paging or caching with rejection to the corresponding variant of paging or caching with bypassing, showing that an α_k-competitive algorithm for a particular variant of caching with bypassing can be converted (using this algorithm as a black box) into a $(4\alpha_k + 1)$-competitive algorithm for the same variant of caching with rejection. If the α_k-competitive algorithm is deterministic, then the resulting algorithm is deterministic, and if the α_k-competitive algorithm is randomized, then the resulting algorithm is randomized. The reduction is non-trivial as the rejection penalties may be arbitrary and simply running an algorithm for caching with bypassing on an input for caching with rejection (ignoring rejection penalties) leads to poor results. Our deterministic algorithm for the caching problem with rejection uses this reduction. This algorithm LANDLORD WITH REJECTION (LLR) uses a variant of the LANDLORD algorithm for caching with bypassing, that is called LANDLORD WITH BYPASSING (LLB). In order to get an upper bound of $2k + 2$ rather than the bound of $4k + 5$ implied by the reduction, we compare the cost of the algorithm directly to an optimal algorithm for caching with rejection. The second deterministic algorithm (see Section 4.2) is different and it is valid for paging, the bit model and the fault model. This algorithm has an optimal competitive ratio of $2k + 1$, and it generalizes the FLUSH WHEN FULL (FWF) algorithm. The role of bypassing in this work is mainly technical, allowing us to design algorithms for caching with rejection by using algorithms for caching with bypassing and our reduction. We present a second reduction (see Appendix), showing that an instance of caching with bypassing and a cache of size k can be seen as an instance of caching (without bypassing) and a larger cache of size $2k$ (for weighted paging, it is possible to decrease the size of the larger cache to $k+1$). Recall that the ratio between the optimal costs for one instance I, seeing it both as an instance for caching with and without bypassing (the ratio $\frac{\mathrm{OPT}_s(I)}{\mathrm{OPT}_b(I)}$), can be infinite for caching even for large inputs (this gap is smaller for paging, but linear for the bit model and the fault model, see Appendix for details). Thus, in order to employ known algorithms, a reduction where the input is modified must be used. The two reductions allow us to design a randomized $O(\log k)$-competitive algorithm for caching with rejection that uses a randomized algorithm for caching (for example, the one of [2]) as a black box. This is the best possible competitive ratio for this problem (up to a constant multiplicative factor).

2 A lower bound

We prove a lower bound on the competitive ratio of any deterministic algorithm. In particular, this lower bound shows that the best possible competitive ratios of deterministic algorithms for caching and for caching with bypassing cannot be achieved already for paging with rejection.

Theorem 1 *The competitive ratio of any deterministic algorithm for paging with rejection is at least* $2k+1$.

Proof We use an input that contains requests for $k + 1$ distinct pages $1, 2, \ldots, k + 1$ (similarly to other proofs for paging problems and their generalizations [30]). Let N, M be large integers and let $\varepsilon = \frac{1}{N} > 0$. Every request has a rejection penalty of ε.

The sequence is constructed as follows. At each time, the next request is for the unique page that would cause a miss to the algorithm. We define a mega-request to be a maximal subsequence of consecutive requests for one page. Such a mega-request consists of some non-negative number of requests that the algorithm rejects, followed by a request for which the algorithm inserts the page into the cache (the only exception can possibly be the last mega-request, where the last page can either be rejected or inserted into the cache by the algorithm). The length of a mega-request is defined to be the number of requests in it. Requests are generated until there are M mega-requests (that is, the algorithm inserted pages into its cache exactly M times), or if the current mega-request has a length of $M^2N + 1$.

Consider an online algorithm whose competitive ratio is at most $2k+1$ (if no such algorithm exists, then the claim follows). We claim that the resulting input has M mega-requests, each of length at most M^2N, i.e., that the sequence is terminated after M mega-requests, such that in all of them the algorithm inserts the page into the cache when the last request of the mega-request is presented (but not earlier). Assume by contradiction that the t-th mega-request ($t \leq M$) has a length of exactly M^2N+1, so the algorithm rejects the page that is requested in this mega-request at least M^2N times. The cost of the algorithm is therefore at least M^2. The cost of an optimal offline algorithm is clearly at most $t \leq M$, since for each mega-request, it can insert this page requested in it into the cache at the time that the mega-request starts. This results in a competitive ratio strictly above $2k+1$ for a sufficiently large value of M.

We consider $2k+1$ offline algorithms OFF_i for $1 \leq i \leq 2k+1$ as follows. For $1 \leq i \leq k+1$, OFF_i is an algorithm that keeps the set of pages $\{1,2,\ldots,k+1\} \setminus \{i\}$ in its cache, and rejects page i when it is requested. The remaining k offline algorithms keep the invariant that they all have distinct subsets of the pages in their caches, and none of them has the same subset as the algorithm. These last k algorithms always have the next requested page in their caches.

Let s_j be the length of the j-th mega-request. The total cost of the online algorithm is $\sum_{j=1}^{M}(\frac{s_j-1}{N}+1) = \sum_{j=1}^{M}\frac{s_j}{N} + M(1-\frac{1}{N})$.

We next compute the total cost of all the $2k+1$ offline algorithms. Each algorithm initializes its cache with a cost of k. Assume that a request arrives, and this request is not the last request of its mega-request. The last k offline algorithms have the request in their caches (since the online algorithm does not have it). Out of the first $k+1$ offline algorithms, exactly one needs to reject the page with a cost of $\frac{1}{N}$. Assume next that the last request of a mega-request arrives. In addition to the offline algorithm that rejects the request, one of the last k algorithms needs to retrieve a page. This page is the page that is requested in the next mega-request, and by this it is ensured that it has a different subset of pages compared to the online algorithm. Thus, the total cost of all offline algorithms is at most $k(2k+1) + \sum_{j=1}^{M}\frac{s_j}{N} + M$. The cost of an optimal offline algorithm is therefore at most $k + \sum_{j=1}^{M}\frac{s_j}{N(2k+1)} + \frac{M}{2k+1}$. For sufficiently large values of M and N, this implies a lower bound of $2k+1$ on the competitive ratio of the online algorithm. □

3 Reducing a caching problem with rejection to a caching problem with bypassing

We present a general reduction from caching with rejection to caching with bypassing. Specifically, we show how to convert an input for caching with rejection into a modified input for caching with bypassing. The reduction has the property that each file keeps its original size and cost in the modified input. Thus, an input for paging with rejection is converted into an input for paging with bypassing, an input for weighted paging with rejection is converted into an input for weighted paging with bypassing, and an input for caching with rejection in the bit model or in the fault model is converted into an input for caching with bypassing in the corresponding model. The reduction can be performed online as the input arrives.

Such a reduction must apply some transformation on the input, since if an algorithm simply ignores the rejection penalties and treats the input for caching with rejection as an input for caching with bypassing (or as an input for caching), then the competitive ratio of the algorithm is unbounded (for example, consider instances in which all rejection penalties are very small, and all requests are for distinct files).

Assume that ALG is a randomized or deterministic α_k-competitive algorithm for caching with bypassing. We present a randomized or deterministic, respectively, $(4\alpha_k+1)$-competitive algorithm for caching with rejection. We define an algorithm F(ALG) that constructs a distribution on possible cache states after each request. For a deterministic algorithm ALG this defines the contents of the cache at each time with probability

1. An algorithm that maintains a distribution on cache states (after each request) immediately defines an algorithm for caching with rejection as follows. After a request was placed, if the requested file is not in the cache for some atom of the distribution, then the request is rejected for this atom by F(ALG) (and if the request is in the cache for some atom, then the request is not rejected for this atom). We define F(ALG) for randomized algorithms, while deterministic algorithms are seen as special cases of randomized algorithms. However, if ALG is deterministic, then F(ALG) is deterministic as well. Step 1 of F(ALG) can be skipped without affecting its correctness proof in this section. It is used, however, in the analysis in Section 4.2. F(ALG) will be an algorithm that can be applied to certain inputs of caching with rejection. In order to obtain an algorithm that can be applied to arbitrary inputs, another transformation is required. Thus, the transformation of an input I has two parts, that can be applied on the input as it arrives. In the first part an input I is modified into an input $\tilde{I}$. The goal of this modification is to ensure that there is no request whose rejection penalty is larger than the cost of the requested file. We will use the property (proved below) that a request for a file f with a rejection penalty p can be replaced with an arbitrary number of requests t, each of rejection penalty $\frac{p}{t}$ (i.e., a sequence of identical requests will appear in the input instead of the original request) without changing the optimal cost for this input. Recall that the cost of a file is strictly positive, thus choosing a sufficiently large t allows us to apply this modification. This transformation will be defined together with an algorithm ϕ. The algorithm ϕ adapts an algorithm $\mathcal{A}$ that assumes that its input (for caching with rejection) has the required properties, while $\phi(\mathcal{A})$ will not assume this. The second part, in which $\tilde{I}$ is modified (online) into I' is motivated by a Ski-Rental type of approach. Informally, we would like to replace a number of requests for one file, whose total rejection penalty is equal to the cost of the file, with one request. Naturally, those requests do not necessarily appear consecutively, and moreover, it could happen that their total rejection penalty is either smaller or larger than the cost. The main idea is to traverse the input online, while a counter for each file f, denoted by $counter(f)$, keeps the total rejection penalty of requests for f since the last time that a request for f was inserted into I' (or since the beginning of the input, if such a request was never inserted). A request of $\tilde{I}$ for f that sets the $counter(f)$ to be at least $cost(f)$ is called a *basic request*, and it is inserted into I' (and the counter of f is set to zero). The only requests that are completely ignored (and no counters are modified) are such that the algorithm surely has the requested file in the cache. Note that when the input $\tilde{I}$ terminates, there may be some files whose counters are positive, but no additional requests for them are inserted into I'.

We start with the modification of I into $\tilde{I}$. $\phi(\mathcal{A})$ is an algorithm for arbitrary inputs for caching with rejection (where no specific relation must hold between the cost of a file and the penalty of a requests for this file), that results from an algorithm $\mathcal{A}$ that assumes that for every request for a file f with the rejection penalty p, $cost(f) \geq p$ must hold.

Algorithm $\phi(\mathcal{A})$

Consider an input I. An input $\tilde{I}$ is created from I as follows. Given a new request for a file f (in I), with the rejection penalty p, create $t(f,p) = \lceil \frac{p}{cost(f)} \rceil \geq 1$ separate requests for f in $\tilde{I}$, each one with the penalty $\frac{p}{t(f,p)}$.
The action of $\phi(\mathcal{A})$ on I is based on the action of $\mathcal{A}$ on $\tilde{I}$. Consider a request for f of penalty p in I, which appears as $t(f,p)$ identical requests in $\tilde{I}$. The algorithm $\phi(\mathcal{A})$ applies exactly the same sequence of changes to the distribution over the cache states that is applied by $\mathcal{A}$ for this subsequence of identical requests in $\tilde{I}$. The resulting distribution will be the distribution at the end of this subsequence. For every atom of the distribution, if there is a step during this sequence of changes such that f is in the cache, then this request for f in I is serviced (by being in the cache), and otherwise, this request is rejected.

Lemma 1 $\phi(\mathcal{A})(I) \leq \mathcal{A}(\tilde{I})$.

Proof Consider each request of I separately. The cost incurred by changes to the contents of the cache is the same for both algorithms. The event that the request is rejected by $\phi(\mathcal{A})$ is contained in the event that $\mathcal{A}$ rejects all the requests in the corresponding subsequence, and the claim follows. $\square$

Next, we show how an algorithm for caching with bypassing is employed on a modified input I'. The input I' results from $\tilde{I}$. We will run ϕ(F(ALG)) on an input I for caching with rejection.

Algorithm F(ALG)

Consider an input $\tilde{I}$.
For every file g in the slow memory, maintain a non-negative value $counter(g)$ (satisfying $counter(g) \leq cost(g)$ at all times, except for possibly the case that g had just been requested and F(ALG) is dealing with this request). For every file g, the variable $counter(g)$ is initialized by zero.

Next, we show how $\tilde{I}$ is transformed into an input I' for caching with bypassing, and define the action of F(ALG). Given a request for a file f in $\tilde{I}$, with a rejection penalty $p \leq cost(f)$, do the following:

1. If the probability of the cache states for which f is in the cache is 1, then do nothing. Otherwise, continue as follows.
2. Let $counter(f) = counter(f) + p$.
3. If $counter(f) < cost(f)$, then do not change the distribution of the cache states.
4. If $counter(f) \geq cost(f)$, then apply the following steps.
   (a) Add f to I', and present f as an input request to ALG.
   (b) Set $counter(f) = 0$.
   (c) If ALG changes the distribution of the cache states, then change the distribution in the same way.

Theorem 2 *Given an online (deterministic or randomized) algorithm* ALG *for caching with bypassing, with a competitive ratio of at most α_k, the online algorithm ϕ(F(*ALG*)) for caching with rejection has a competitive ratio of at most $4\alpha_k + 1$.*

Proof We see deterministic algorithms as special cases of randomized algorithms, and therefore we can assume without loss of generality that ALG is a randomized algorithm. Consider a specific request sequence I. We first analyze the effect of the modification of the input sequence I into $\tilde{I}$, i.e., splitting some requests into a sequence of consecutive identical requests. The input $\tilde{I}$ has the property that for each requested file f, $cost(f)$ is no smaller than the rejection penalty of this request. We prove that $\text{OPT}(\tilde{I}) = \text{OPT}(I)$. Given an optimal solution for I, it is possible to construct a solution of the same cost for $\tilde{I}$ as follows. Given a request for f in I, if the request is rejected, then the subsequence of corresponding requests of $\tilde{I}$ are rejected (incurring the same cost in total), and if the request is not rejected, then the contents of the cache are modified for $\tilde{I}$ just before the first request of the subsequence corresponding to the single request for f in I. Next, given an optimal solution for $\tilde{I}$, it induces a solution for I as follows. Consider a subsequence of requests in $\tilde{I}$ corresponding to one request in I. If all these requests are rejected, then the single request in I is rejected too. Otherwise, since OPT is lazy, the contents of the cache of the solution for I are modified to be as they are for $\tilde{I}$ after the last request of the subsequence. The cost of the resulting solution for I does not exceed the cost of the optimal solution for $\tilde{I}$ that we considered. An algorithm F(ALG) for $\tilde{I}$ defines the algorithm for I, called ϕ(F(ALG)). By Lemma 1, we have $\phi(F(\text{ALG}))(I) \leq F(\text{ALG})(\tilde{I})$.

The sequence I', which is an input for caching with bypassing, was created as follows. All requests on which F(ALG) does not call ALG were removed. Any request for a file f (with some penalty) on which F(ALG) calls ALG is given as a request for the file f. Any request in $\tilde{I}$ that was transformed into a request in I' is called a *basic request*. We will now apply another transformation on $\tilde{I}$, resulting in an input I'' for caching with rejection. This transformation is defined for the sake of the proof, and it is not used by any of our algorithms.

Let H be the set of files requested in $\tilde{I}$. For a file $g \in H$, let $cost'(g)$ be the total penalty of all requests for g in $\tilde{I}$ that appear after the last basic request for g. If there are no basic requests for g, then $cost'(g)$ is defined to be the total penalty of all requests for g. If the last request for some file g in $\tilde{I}$ is a basic request, then we let $cost'(g) = 0$. By the definition of basic requests, $cost'(g) < cost(g)$ for any $g \in H$.

Claim 1 $\text{OPT}(\tilde{I}) \geq \sum_{f \in H} cost'(f)$.

Proof For every $f \in H$, consider the subsequence of $\tilde{I}$ corresponding to requests for f. If f is ever inserted into the cache of the optimal solution for $\tilde{I}$, then at least $cost(f) \geq cost'(f)$ is paid for these requests. Otherwise, the total rejection penalty for f is at least $cost'(f)$, and since all requests for f must be rejected by this optimal solution, then it pays at least $cost'(f)$ for these requests. □

Denote by I'' the input sequence that results from $\tilde{I}$ by removing for every file $g \in H$ all requests that appear after the last basic request for g (or all requests for g, if none of these requests is basic). Note that the algorithm F(ALG) does not change the distribution of the cache states when requests of $\tilde{I} \setminus I''$ arrive. Since $\tilde{I}$ contains all requests of I'', we have $\text{OPT}(I'') \leq \text{OPT}(\tilde{I})$. Additionally, we can establish the following claim.

Claim 2 *For any realization of the random bits during the execution of* ALG *on I,*

$$\phi(F(\text{ALG}))(I) \leq F(\text{ALG})(I'') + \sum_{f \in H} cost'(f) \leq F(\text{ALG})(I'') + \text{OPT}(\tilde{I}) .$$

Proof The second inequality holds by Claim 1. Using $\phi(F(\text{ALG})(I)) \leq F(\text{ALG})(\tilde{I})$, it is sufficient to prove $F(\text{ALG})(\tilde{I}) \leq F(\text{ALG})(I'') + \sum_{f \in H} cost'(f)$. To show this inequality, note that the cost of $F(\text{ALG})(\tilde{I})$ paid for requests for a file f differs from the cost that $F(\text{ALG})(I'')$ paid (for requests of the same file f) by at most $cost(f)$. The claim follows since for the requests of $\tilde{I} \setminus I''$ the distribution of the cache state does not change and hence such requests do not affect the cost paid for requests for other files. □

Given the input I'', and a file g, $t(g) \geq 0$ be the number of requests for g in I', that is, the number of basic requests for g in I''. If $t(g) > 0$, then the time interval between the time just before the very first request for g in I'' and until the time just after the last request for g in I'' (which is a basic request) is partitioned into $t(g)$ intervals. Each interval ends just after a basic request (i.e., it contains a non-negative number of requests for g that are not basic, followed by a basic request).

Claim 3 *For any realization of the random bits during the execution of* ALG *on I,* $F(\text{ALG})(I'') \leq 2\text{ALG}(I')$.

Proof Fix a realization of the random bits. Then, both ALG and F(ALG) can be seen as deterministic algorithms. For a file g, it suffices to show that the cost of F(ALG) on the subsequence of requests of one interval is at most twice the cost of ALG on this subsequence. First, consider such a subsequence where there is a point in time where g is not in the cache of ALG. The cost of ALG, running it on I', for such a subsequence (which in fact contains only the last request of this subsequence) is at least $cost(g)$, since ALG must either bypass it or it inserts it into the cache. The cost of F(ALG) on such a subsequence is at most the value of the counter just before the basic request, which is smaller than $cost(g)$, plus the cost of the basic request, which is at most $cost(g)$, no matter if the file is rejected or inserted into the cache by F(ALG). Finally, consider a subsequence where g is in the cache throughout the time interval since the last basic request for g. Then, both the costs of ALG and F(ALG) on this subsequence are 0. □

Claim 4 $\text{OPT}(I') \leq 2\text{OPT}(\tilde{I})$.

Proof Since $\text{OPT}(I'') \leq \text{OPT}(\tilde{I})$, we will prove $\text{OPT}(I') \leq 2\text{OPT}(I'')$. Consider an optimal solution for I'' and convert it into a solution SOL for I' by maintaining the same contents in the cache after the arrival of each request, exactly as in I''. Recall that I' is an input for caching with bypassing, while I'' is an input for caching with rejection. The cost of SOL may be larger for I' (compared to the cost for I'') due to basic requests for which the cache does not have the requested file, and so this file must be bypassed (while the rejection penalty in I'' is smaller).

Let $t_1(g) \leq t(g)$ be the number of basic requests where g is not in the cache just after the request (that is, the number of requests for g that are bypassed in SOL). We claim that out of the total value $\text{OPT}(I'')$, the total cost paid for the rejection of requests for g, and for insertion of g into the cache, is at least $t_1(g) \cdot cost(g)$. Therefore, it will follows that $\text{OPT}(I'') \geq \sum\limits_{g \in H} t_1(g)cost(g)$, while $\text{SOL}(I') \leq \text{OPT}(I'') + \sum\limits_{g \in H} t_1(g)cost(g) \leq 2\text{OPT}(I'')$.

To show this, consider the optimal solution OPT for the sequence I'' for a fixed interval with respect to a file g. Specifically, consider an interval for g in which g is not in the cache in the beginning of the interval. OPT could insert g into the cache during the interval, and in this case its cost for g during this time interval is at least $cost(g)$. Otherwise, if this is not the case, then it needs to reject all requests for g of this interval. These requests have a total penalty of at least $cost(g)$, showing that the cost for g is at least $cost(g)$ in both cases. We show that the number of such intervals (where g is not in the cache in the beginning) is at least $t_1(g)$. The first interval has this property. Additionally, there are $t_1(g)$ basic requests after which g is not in the cache, out of which at least $t_1(g) - 1$ basic requests are not the last basic requests, and each is followed by such an interval. We find that the total cost for g is at least $t_1(g)cost(g)$ as claimed. □

By the last three claims and the fact that ALG is α_k-competitive, we conclude that the following hold.

$$\begin{aligned} E[\phi(F(\text{ALG})(I))] &\leq E[F(\text{ALG}(I''))] + \text{OPT}(\tilde{I}) \leq 2E[\text{ALG}(I')] + \text{OPT}(\tilde{I}) \\ &\leq 2\alpha_k\text{OPT}(I') + \text{OPT}(\tilde{I}) \leq (4\alpha_k + 1)\text{OPT}(\tilde{I}) \leq (4\alpha_k + 1)\text{OPT}(I) \ . \end{aligned}$$

□

4 Algorithms

4.1 Caching with bypassing

We use an algorithm that is similar to a special case of LANDLORD [37], with some modifications. In particular, it is allowed to bypass a file in some cases. Recall that Cohen and Kaplan [19] analyzed a modified version of GREEDY DUAL with bypassing. Our algorithm is similar, but we use a different special case of LANDLORD. We also use a direct analysis. This allows us to perform a better analysis of the variant of this algorithm for caching with rejection (which results from the reduction between the two problems).

The algorithm maintains a cache (called an actual cache in what follows), where each file in the actual cache has a credit assigned and maintained by the algorithm. Normally, files that are not in the actual cache do not have credits (or alternatively, their credits are defined to be zeroes), except for the case that a file was just requested, and the algorithm is in the process of deciding on its further actions.

On a miss for a file f (that is, a request for a file f that is currently absent from the actual cache), the algorithm considers a *virtual cache*, denoted by G. The virtual cache G contains all files that are present in the actual cache (each having a credit) and f is inserted into the virtual cache and it is assigned a credit too. This last credit equals its cost. The size of the virtual cache becomes at most $k + size(f)$. Then all credits of files in G are decreased repeatedly by small amounts (keeping all credits non-negative), and files of credit zero are removed from the virtual cache, until the size of the virtual cache is at most k. Each file that is in the actual cache and its credit became zero is removed from the actual cache. If at some time f is no longer present in the virtual cache, then f is bypassed by the algorithm, and the algorithm proceeds to the next request (in this case G has files of a total size at most k). Otherwise, if G reaches a total size of at most k while the credit of f is strictly positive, f is inserted into the actual cache, keeping its current credit (which may be smaller than its cost, at this time, but it must be positive).

Algorithm LANDLORD WITH BYPASSING (LLB)
Maintain a non-negative value $credit(g)$ (which satisfies $credit(g) \leq cost(g)$) for every file g, that is initialized by zero.
Given a request for a file f, if f is not in the actual cache, then do the following:

```
1. Let $credit(f) = cost(f)$.
2. Let $G$ be a set of files consisting of all files in the actual cache and $f$.
3. While $size(G) > k$
   (a) Let $\Delta = \min_{g \in G} \frac{credit(g)}{size(g)}$.
   (b) For each $g \in G$, let $credit(g) = credit(g) - \Delta \cdot size(g)$.
   (c) For each $g \in G$ such that $credit(g) = 0$, remove $g$ from $G$.
4. Remove every file $g$ in the actual cache that satisfies $g \notin G$.
5. If $f \in G$, insert $f$ into the actual cache, and otherwise bypass $f$.
```

Note that the initial credit of files outside the actual cache is zero, and this holds for every file outside the actual cache after each request was serviced. The only time that a file gets a non-zero credit without being inserted into the actual cache is when it is requested, and it is inserted into the virtual cache G.

The result of Young [37] is proved using a potential function that can be easily adapted to show that LLB is $(k+1)$-competitive. We give a different analysis that builds on the analysis of [37, 16], with the addition of virtual caches. As the result of this section is not new, its main purpose is to provide a foundation for the analysis in the next section. We let LLB and OPT denote the sets of files that LLB and OPT have in their caches, respectively (for LLB this is the actual cache).

Theorem 3 *The competitive ratio of* LLB *is at most* $k+1$.

Proof We give an alternative proof that will be helpful for the analysis of the algorithm for caching with rejection that we will present later (which is not the proof of [19]). We apply the following reductions on the input. All requests to files that are not misses are removed. Due to the definition of LLB, such requests do not affect the behavior of the algorithm. For the analysis, as it is done in previous proofs, we assume that given a request, OPT deals with it first, and only then the algorithm deals with it. Recall also that OPT is lazy in the sense that it does not change the contents of the cache unless it services a request. We assume that the actual cache of LLB and the cache of OPT are initially empty.

Let a *bad interval* for a file f be a maximum time interval such that LLB has f in its virtual cache, but OPT does not have it during the entire interval and does not bypass it. Let $\nu(f)$ be the number of bad intervals for f. We have an additional binary variable for every file f, denoted by $b(f)$. We let $b(f) = 1$ if f is requested at least once after the last bad interval ends, and $b(f) = 0$ otherwise.

Let H denote the set of all files ever requested.

Claim 5 OPT $\geq \sum\limits_{f \in H} (\nu(f) + b(f)) cost(f)$.

Proof We will show that when the input reaches the ith bad interval for file f, OPT already had a cost of at least $i \cdot cost(f)$ for file f (that is, the number of times that OPT either inserted f into its cache or bypassed f is at least i). We use induction. For the base case, since the first bad interval starts after LLB has f in its virtual cache, this means that there was a request for f before the bad interval started, and OPT either bypassed f or inserted it into its cache (and removed it later). Let $i > 1$, and consider the time between the $(i-1)$th bad interval and the ith bad interval. If OPT bypassed f or inserted it into the cache between the two bad intervals, then we are done. Otherwise, since when the $(i-1)$th bad interval ends OPT does not have f in its cache, and it does not deal with a request for it till the ith bad interval starts, we find that there were no requests for f. Therefore, LLB also does not insert it into its virtual cache. However, it must be the case that when the $(i-1)$th bad interval ends, it is because LLB removes f from its virtual cache, and the ith bad interval can start only if LLB inserts it again into its virtual cache, contradicting the fact that there is no request for f at that time. If $b(f) = 0$, then we are done. Otherwise, since every request for f incurs a miss to LLB, and OPT is lazy, we find that at the time that the last bad interval ends, OPT does not have f in its cache. Since f is requested again afterwards, OPT has an additional cost of $cost(f)$ for the first such request. □

Let Δ_i be the i-th value of Δ, for $i = 1, \ldots, p$, where p is the number of times that Δ is computed. We assign weights to files as follows. The weight of f, $w(f)$ is initialized by zero. Consider the value Δ_i. Recall that at the time that it is computed, we assume that OPT has already serviced the current request. The algorithm inserted the requested file into the set G, but the set G still has a total size that exceeds k. After the value Δ_i is determined, the weight of every file $g \in G$ that is not present in the cache of OPT increases by $size(g)\Delta_i$.

For each file, let $w(f)$ and $credit(f)$ denote these values at the end of the input.

Claim 6 *At the time of termination, for each file g it holds that* $w(g) + credit(g) \leq (\nu(g) + b(g)) \cdot cost(g)$.

Proof The claim holds for every file g that satisfies $w(g) = 0$, thus we only consider files for which the value is positive. The value $w(g)$ can only be modified during a bad interval for g. During such an interval, LLB has g in its virtual cache continuously, so $credit(g)$ is only modified by decreasing it by the same value that $w(g)$ is increased. Thus, during the ith bad interval such that $i \leq \nu(g)$, the value $w(g)$ increases by at most $cost(g)$, and after $\nu(g) - 1$ bad intervals, we have $w(g) \leq (\nu(g) - 1)cost(g)$. When the last bad interval ends, $w(g) + credit(g) \leq \nu(g) \cdot cost(g)$ holds. If $b(g) = 0$, then there are no further requests for g, so $credit(g)$ cannot increase until termination. Otherwise, $b(g) = 1$, and at the time of termination $credit(g) \leq b(g)cost(g)$ while $w(g) \leq \nu(g)cost(g)$, and we are done. □

Let G_i denote the set G at the time that Δ_i is computed, and let $|G_i|$ denote the total size of files in G_i. Let C_F denote the remaining total credit of files in the actual cache at termination. The following holds by definition (since a file is removed from G only if its credit is zero).

Claim 7 LLB $\leq \sum\limits_{i=1}^{p} \Delta_i \cdot |G_i| + C_F$.

Claim 8 $\sum_{i=1}^{p} \Delta_i \cdot |G_i| \leq (k+1) \sum_{f \in H} w(f)$.

Proof At the time that Δ_i is defined, the files in the cache of OPT have a total size of at most k. Therefore, the total size of files for which the weight increases is at least $|G_i| - k$, for a total increase of at least $(|G_i| - k)\Delta_i$. The function $\frac{\Delta_i \cdot |G_i|}{(|G_i|-k)\Delta_i}$ is maximized (over positive integers $|G_i|$ such that $|G_i| \geq k+1$) for $|G_i| = k+1$. We get that the two expressions are initialized with zero, and each time that the left hand side increases, the right hand side increases by at least the same amount. □

We finally get:

LLB $\leq \sum_{i=1}^{p} \Delta_i \cdot |G_i| + C_F \leq (k+1) \sum_{f \in H} w(f) + C_F \leq (k+1) \sum_{f \in H} w(f) + (k+1)C_F \leq (k+1) \sum_{f \in H} (\nu(f) + b(f)) \cdot cost(f) \leq (k+1)$OPT.

□

4.2 Caching with rejection

We design an algorithm for caching with rejection. Our algorithm LANDLORD WITH REJECTION (LLR) is exactly ϕ(F(LLB)). Using Theorem 2, we get an upper bound of $4k+5$ on the competitive ratio of LLR. We present a more careful analysis of LLR to show the following.

Theorem 4 *The competitive ratio of* LLR *is at most* $2k+2$.

Proof Recall that we assume that OPT (both for caching with rejection and for caching with bypassing) is lazy, that is, it never inserts a file into the cache unless a miss occurs, and it only evicts a file in order to make room for another file. We also assume that given a new request, OPT deals with it first, and then ϕ(F(LLB)) deals with it. This will be used in the proof. Consider a specific request sequence I. We use the notations of the proof of Theorem 2. We will analyze $F(\text{LLB})(\tilde{I})$, since $\phi(F(\text{LLB}))(I) \leq F(\text{LLB})(\tilde{I})$, and $\text{OPT}(I) = \text{OPT}(\tilde{I})$. As F(LLB) ignores all requests of $\tilde{I}$ that did not cause a miss (these requests do not increase the counter, and are not presented to LLB), we remove these requests from the sequence, and analyze the resulting sequence (without loss of generality we assume that $\tilde{I}$ is such a sequence). Another assumption on $\tilde{I}$ that we make is that every file that is requested at least once also has at least one basic request. To show this, consider such a file g. The counter $counter(g)$ for F(LLB) applied on $\tilde{I}$ never reaches $cost(g)$, so g is never given as an input to ALG and all requests for g are rejected by F(LLB). OPT for $\tilde{I}$ also rejects all the requests for g, as the total rejection penalty is strictly below $cost(g)$. Thus, removing such requests from $\tilde{I}$ decreases the costs of F(LLB) and OPT by the same amount. Without loss of generality we assume that $\tilde{I}$ does not contain such requests.

We will show that without loss of generality we can assume $I'' = \tilde{I}$. Recall that the sequence I'' is an input for caching with rejection where for each file its last request is a basic request (which is a request in I').

Lemma 2 *Without loss of generality, we assume that the last request for any file f appearing in $\tilde{I}$ is a basic request.*

Proof First, we show that we can assume without loss of generality that every request for file f that appears after the last basic request for f is not rejected by OPT. Consider such a request for f that was rejected by OPT. Delete this request from $\tilde{I}$. The cost of the algorithm and of OPT are reduced by the same quantity (the rejection penalty of the deleted request), since the algorithm rejects this request as well (since this request as any request is a miss, and it is not a basic request). This is applied to every file.

Given the above assumption, we show that it can be assumed without loss of generality that the last request for file f in $\tilde{I}$ is basic. Assume that there is a request for a file f (in $\tilde{I}$) that is the last request for f and it is not a basic request. This request appears after the last basic request, and by the above OPT does not reject this request, so OPT serves it by having it in its cache. As this request is a miss for F(LLB), and the request is not basic, F(LLB) rejects this request (and does not have f in its cache both before it is requested and after the request). Denote by c the value of $counter(f)$ just after this request. Modify the input $\tilde{I}$ by adding two new requests for f. The first such request for f is added just after the currently last request for f, and its rejection penalty is $(1-\varepsilon)\cdot(cost(f)-c)$ (for a small enough value of ε satisfying $\frac{1-\varepsilon}{\varepsilon} > 4k$ or alternatively $\varepsilon < \frac{1}{1+4k}$). This new request has a rejection penalty below $cost(f)$, and $counter(f)$ becomes $(1-\varepsilon)cost(f) + \varepsilon \cdot c < (1-\varepsilon)cost(f) + \varepsilon \cdot cost(f) = cost(f)$, so this new request will not be basic, and hence the algorithm will reject it, and will not change the contents of its cache, while OPT has f in its cache at this time. Thus, adding it does not change the behavior of OPT, the cost of F(LLB) increases, but its further behavior does not change.

The second new request for f is added at the end of the sequence and its rejection penalty is $\varepsilon\cdot(cost(f)-c)$. This last request is a basic request. The cost of the algorithm does not decrease by this last request (and it will not affect further behavior, as it will be the last request), whereas the cost of OPT increases by at most $\varepsilon \cdot (cost(f) - c)$. Thus the ratio between the increase of the cost of the algorithm and the increase of the cost of OPT is at least $\frac{1-\varepsilon}{\varepsilon} > 4k$. Since we intend to prove that the competitive ratio of the algorithm is at most $2k + 2$, we can assume that our instance $\tilde{I}$ satisfies the conditions of the lemma.

Note that the new requests do not contradict our assumptions on $\tilde{I}$; the penalty of a request never exceeds its cost, and for file f we added two requests such that both of them are misses. Moreover, we did not add requests for files that will not have basic requests. □

To prove an upper bound of $2k + O(1)$, instead of $4k + O(1)$ (using the reduction), we will compare the cost of F(LLB) for I'' directly to the cost of LLB for I', and compare this last cost to the cost of OPT for I''. The action of F(LLB)=LLR on sequences of this type (where the last request for each requested file is basic) is as follows. For every file f, the counter is zero at termination. Since every request is a miss, and the contents of the cache are never modified for requests that are not basic, every request of I'' that is not basic is a miss, and F(LLB) rejects it. Since the rejection penalty of a single request for a file f does not exceed $cost(f)$, $counter(f)$ never reaches $cost(f)$ as long as the current request is not basic. Since the total rejection penalty for f since the last time that $counter(f)$ was zero is exactly $counter(f)$, the cost of rejecting f between two times that its counter is zero is at most $cost(f)$. The cost of LLB for a basic request for f is exactly $cost(f)$ (no matter whether it inserts f into the cache of bypasses it). Thus, the resulting cost of F(LLB) applied on I'' is at most twice the cost of LLB applied on I'. This is summarized as follows.

Let H be the set of files requested in I'' (and therefore also in I'). The following claim holds by the discussion above.

Claim 9 $F(\text{LLB})(I'') \leq 2\text{LLB}(I')$.

We define bad phases for I'' and a file f. The definition is different from that of bad intervals in the previous section, but serves similar purposes. A bad phase for f is a maximal time interval such that LLB has f in G, while OPT does not have it in its cache for at least a part of the time. Recall that the algorithm can only insert f into G on a basic request for f. The next basic request occurs only after f was evicted from G. Such a time interval can be very short if the file is not inserted into the cache of F(LLB). We observe that (since OPT is lazy) each bad phase contains exactly one bad interval (as defined in the previous section) that ends when the bad phase ends, however it will be easier for us to refer to the starting point of a bad phase rather than to that of a bad interval. Thus, we use $\nu(f)$ to denote the number of bad phases (or bad intervals) for f, and $b(f) = 1$ if and only if there is at least one (basic) request for f after the last bad interval ends.

Claim 10 $\text{OPT}(I'') \geq \sum_{f \in H} (\nu(f) + b(f))cost(f)$.

Proof We will show that when the input reaches the beginning of the ith bad phase for file f, OPT already had a cost of at least $i \cdot cost(f)$ for file f (including the cost for inserting f into its cache and the total cost of rejecting requests for f). We use induction. For the base case, the first bad phase starts with a basic request for f. If OPT inserted f into its cache before the first bad phase starts, then we are done. Otherwise, OPT rejects all requests for f till that time, and its cost is at least $cost(f)$ since the total rejection penalty for f of all requests including this basic request is at least $cost(f)$. Let $i > 1$. If OPT inserts f into the cache after the previous bad phase started, then we are done. Otherwise, we claim that f is not in the cache of OPT at all times that the following requests for f arrive: requests arriving strictly after the basic request such that the $(i-1)$th bad phase starts right after it, and before (and including) the next basic request. During this time, OPT does not insert f into the cache. Since it is a bad phase, there is an overlap between the time that LLB has f in G but OPT does not, that is, OPT no longer has f in the cache when LLB removes f from G at the end of the $(i-1)$th bas phase, and since all requests are missed, only then additional requests for f can start arriving. Thus, OPT must reject requests for f of a total rejection penalty of at least $cost(f)$ until (and including) the following basic request. If $b(f) = 0$, then we are done. Otherwise, if OPT inserts f into its cache after the last bad phase started, then we are done, and if it does not, then since there will be another basic request, the previous proof shows that OPT has an additional cost of at least $cost(f)$, no matter whether the last time interval that LLB has f in G is a bad phase or not. □

We will compare an optimal solution for I'' to LLB. The difference between a solution for I' and a solution for I'' is that I'' has additional requests that are not basic. Claim 6 still holds since in every bad phase for f, $counter(f)$ is set to $cost(f)$ and $w(f)$ increases by at most the amount that $counter(f)$ decreases, and this can only happen during bad phases. As in the previous section, we find the following claim.

Claim 11 $\text{LLB}(I') \leq (k+1)\text{OPT}(I'')$

We thus have

$$F(\text{LLB})(I'') \leq 2\text{LLB}(I') \leq (2k+2)\text{OPT}(I'') \,.$$

□

4.3 An improved algorithm for several cases

We design a phase-based algorithm. We define the algorithm for arbitrarily sized files since we use this algorithm not only for paging, but also for caching in the bit and fault models. The algorithm creates phases such that in each phase, the cache is initially empty, and the phase ends when according to the rules of the phase, another file should be inserted into the cache. This last file will possible be inserted into the cache in the next phase, but this decision is postponed. During a phase, a file f that is not in the cache of the algorithm is rejected as long as its total rejection penalty is at most 1. Once this value (called the counter of f and denoted by $counter(f)$) will exceed 1 due to a new request for f, f is inserted into the cache. However, if there is no space for a file f, then a new phase begins, but the rejection penalty that was paid for f by the algorithm in the current phase (at most 1) still counts towards the cost of the current phase. The algorithm actually sees this cost as exactly 1 (even if it paid a small amount), and thus before starting the next phase, it reduces the counter of f by 1, while all other files do not keep a record of their status in the current phase; at the end of the phase all files are removed from the cache, and all counters are set to zero (excluding $counter(f)$).

Algorithm FLUSH WHEN FULL WITH REJECTION (FWFR)

Maintain a non-negative value $counter(g)$ for every file g in the slow memory. Let λ be the total size of the files in the cache. Initialize all these values with zeroes.
Given a request (f, r), if f is not in the cache, then let $counter(f) = counter(f) + r$, and do the following:

1. If $counter(f) \leq 1$ then reject f.
2. If $counter(f) > 1$, and $size(f) + \lambda \leq k$, then insert f into the cache and let $\lambda = \lambda + size(f)$.
3. If $counter(f) > 1$, and $size(f) + \lambda > k$, then empty the cache and set $\lambda = 0$, for each $g \neq f$ let $counter(g) = 0$, and let $counter(f) = counter(f) - 1$. Go to step 1.

Claim 12 *Consider a time that the algorithm just finished dealing with a request. For every file g, $counter(g) > 1$ if and only if g is in the cache of* FWFR.

Proof The claim is proved using induction. Initially all counters are equal to zero, and the cache is empty. The contents of the cache change only if a new request for a file f is a miss. If step 1 is applied on the new request, then the contents of the cache do not change, and the only counter that is modified is $counter(f)$ that remains at most 1. If step 2 is applied, then the only counter that changes is that of f. This counter increases above 1 and f is inserted into the cache. If step 3 is applied, then all counters except for that of f will be zero, and the cache is emptied. Then, either step 1 or step 2 are applied on f, and if $counter(f)$ remains above 1 after it was reduced by 1 in step 3, then it will be inserted into the cache, and otherwise it will not be inserted. □

Theorem 5 FWFR *is $(2k + 1)$-competitive for paging with rejection, and for caching with rejection in the bit model and in the fault model.*

Proof FWFR partitions the input into phases, between consecutive times that the cache is empty (it is initially empty, and it can be emptied in step 3). The last phase may be such that the cache is not empty when it ends, and we call this phase *incomplete*. We modify the input by removing all requests that are not misses. This cannot increase the optimal cost and does not change the behavior of FWFR.

Consider a request for a file f with penalty r that FWFR processes it in several steps (out of steps 1,2,3). The only case that this can happen is that the request is first processed in step 3, and then it is considered again. Since after step 3 we must have $\lambda = 0$, the request will be processed again only once, in step 1 or step 2. For each such request, we modify the sequence by splitting it into two requests for the analysis. Let C be the value of the counter of f just before the request (f, r) arrives. Since this request is a miss, by Claim 12, f is not in the cache prior to this request, and $C \leq 1$. Since step 3 was applied, we find that $C + r > 1$. Replace the request (f, r) with the requests $(f, 1 - C)$ and $(f, C + r - 1)$. As a result of this modification the cost of an optimal solution does not change. The first request out of the two will be the last request of a phase, while the second one will be the first request of the following phase. FWFR will consider the first request of these two requests in step 1, and the second request will be considered both in step 3 and afterwards in step 1 or 2. If it is considered again in step 1, then the cost of FWFR on (f, p) does not change, as it rejects both $(f, 1-C)$ and $(f, C+r-1)$. Otherwise, its cost increases by $1-C$, as it first rejects the first request of the two, and then inserts f into the cache. The behavior on other requests does not change, as the values of counters and contents of the cache will be the same, no matter whether FWFR received one request for (f, r) or the the two consecutive requests $(f, 1 - C)$, $(f, C + r - 1)$. We analyze the modified sequence in what follows. Note that if $C = 1$, then the rejection penalty of the last request of the phase is zero, we keep this request for technical reasons (and assume that FWFR rejects it). Using this modification we can assume that every file f considered in step 3 has $counter(f) = 1$ prior to the current request (f, r), and after applying this step, $counter(f) = C + r - 1$. If $C + r - 1 > 1$, then f is inserted into the cache immediately, and otherwise this rejection penalty is paid. Thus, we find that in the beginning of a phase, all counters except for that of f are equal to zero, and therefore the rejection penalty paid by any file during a phase is no larger than 1 (by Claim 12, a file is rejected as long as its counter does not exceed 1).

We consider a specific complete phase (a phase that is not incomplete, i.e., a phase that is followed by another phase), and partition the files requested in this phase into four types. The first type consists of all files that are inserted into the cache of FWFR during the phase, and the file that is requested last in this phase (this is the same file that is requested first in the next phase). The three other types consist of files that are requested in the phase, but not inserted into the cache, none of which is the last file requested in the phase. These files belong to three categories: files that OPT has in its cache in the beginning of the phase, files that OPT inserts into the cache during the phase and were not present in the cache of OPT in the beginning of the phase, and files that OPT also rejects all their requests in this phase.

Consider next a request for a file f' of the fourth type. Removing all requests for f' in this phase from the input does not change the behavior of FWFR on the other requests, and it reduces the cost of FWFR and of OPT by the same amount. Thus all such requests are removed from the input. We therefore assume that every file requested in a phase belongs to one of the first three types. Let a_i denote the number of files of type i (for $i = 1, 2, 3$). Clearly, $a_2 \leq k$, since the cache of OPT in the beginning of the phase cannot have more than k files. We next argue that in all models, the cost of FWFR for this phase is at most $2k + 1 + a_2 + a_3 \leq 3k + 1 + a_3$. The total retrieval cost is at most k, since no files are evicted until the phase ends. There are at most $k + 1$ files of the first type (k that are inserted into the cache, and possibly the file of the last request of the phase), and since for any file, the total rejection penalty of the requests in this phase is at most 1, the total rejection penalty is at most $(k + 1) + a_2 + a_3$.

The total size of the files of the first type exceeds k, so no matter if OPT inserts such a file into the cache or not, it has a cost of at least 1 on such a file in this phase (since the rejection penalty of each such file is at least 1). Therefore, the optimal cost is at least $a_3 + 1$ (since it additionally inserts at least a_3 files into the cache). If $a_3 \geq 1$, then the ratio of the costs is smaller than $2k + 1$. Otherwise, $a_3 = 0$, so any file that is requested in this phase is of the first type or the second type. Moreover, if the cost of OPT in this phase is at least 2 then we are done, and if $a_2 = 0$ we are done as well. So we assume $a_2 > 0$ and a cost of less than 2 for OPT in this phase. Therefore, OPT has a_2 files of the second type in its cache when the phase begins, and can have at a total size of at most $k - a_2 \leq k - 1$ of files of the first type in its cache before the phase begins. However, the total size of files of the first type denoted by j is at least $k + 1$.

The remainder of the proof depends on the specific model. We first consider the bit model and afterwards the fault model. Since paging is a special case of both, the result for paging will follow. We show that in all cases, the cost of FWFR on this phase is at most $2k + 1$.

Consider the bit model. As $j \geq 2$, but the cost of OPT is strictly below 2 for this phase, it must reject all the requests for at least one file of the first type. Since the total rejection penalty of requests for a given file of the first type is at least 1, and inserting a file into the cache incurs a cost of at least 1 as well, it must be the case that there is exactly one file of the first type that OPT does not have in its cache when the phase begins, and all the requests for this file are rejected by OPT. The total size of this file is at least $j - (k - a_2) = j - k + a_2$. Let g denote this file. In this case, we can find a smaller upper bound on the cost of FWFR. If g is the file requested last in the phase, then the total size of files of the first type excluding g is at most $j - (j - k + a_2) = k - a_2$. The total cost of file retrieval done by the algorithm is at most $k - a_2$. There are at most $k - a_2 + 1$ files of the first type, and thus the total rejection penalty paid by the algorithm is at most $k+1$, and its total cost for the phase is at most $2k - a_2 + 1$. Otherwise, the number of files in the first type is at most $k + 1 - (j - k + a_2 - 1)$ (since one file has size of at least $j - k + a_2$), so there are at most $2k - j + 1$ files for which it pays a rejection penalty of at most 1 per file, and the retrieval cost is at most k. The cost of FWFR is at most $3k - j + 2$. Using $j \geq k + 1$ this is at most $2k + 1$.

In the fault model, if OPT has a cost smaller than 2 on the phase, then there must also be exactly one file that is not in the cache of OPT in the beginning of the phase, as every file incurs a cost of at least 1 in this model too. In this case OPT either rejects all the requests for this file, or inserts it into the cache, but once again there is a file of size at least $j - k + a_2$. Similarly to the bit model, if this file is the last file of the phase, then there are at most $k - a_2$ other files of the first type, and at most $k + 1$ files in total, so the cost of FWFR is at most $2k - a_2 + 1$. Otherwise, the algorithm inserts at most $k - (j - k + a_2 - 1) = 2k - j - a_2 + 1$ files into the cache, and there are at most $2k - j + 2$ files in total, so it has a cost of at most $4k - 2j - a_2 + 3 \leq 4k - 2(k + 1) - a_2 + 3 = 2k - a_2 + 1$ (since $j \geq k + 1$).

Note that for an incomplete phase, the cost of OPT is at least a_3, while the cost of FWFR is at most $3k + 1 + a_3$, so the cost of this phase contributes an additive constant of at most $3k + 1$. $\square$

A Some properties and a randomized algorithm

We will use a reduction between caching with bypassing and caching to obtain a randomized algorithm. In this reduction, given an input for caching with bypassing and a cache of size k, the cache size for the resulting instance is different, and moreover, the instance is modified. We briefly discuss the relation between the models with and without bypassing for a fixed input. Thus, we show that the most straightforward reduction between the case with and without bypassing for the bit model and the fault model fails, while for paging the difference between the optimal costs is much smaller. Moreover, in the case of weighted paging, (and therefore, also for caching), the ratio between the optimal costs of the two variants is unbounded.

Proposition 1 *For paging,* $\sup_I \frac{\text{OPT}_s(I)}{\text{OPT}_b(I)} = 2$. *In the bit model and in the fault model,* $\sup_I \frac{\text{OPT}_s(I)}{\text{OPT}_b(I)} = k+1$. *For weighted paging and file caching* $\frac{\text{OPT}_s(I)}{\text{OPT}_b(I)}$ *can be arbitrarily large.*

Proof We first show the upper bounds (the upper bound for paging is folklore and was previously mentioned in [5]). Consider an optimal offline algorithm OPT_b with bypassing. We convert it into an algorithm OFF_s that never uses bypassing. Each time that OPT_b bypasses a file f, OFF_s inserts this file into the cache, temporarily evicting arbitrary files until there is sufficient room for it, and then it removes the file f again, inserting the temporarily evicted files back into the cache. For paging, there is at most one evicted page for every request page. The cost incurred by OFF_s for one request page is at most 2, i.e., at most twice the cost of OPT_b for a page. In the bit model, the total size of evicted files is at most k, so the cost of OFF_s for f is at most $\frac{size(f)+k}{size(f)} \leq k + 1$ times the cost of OPT_b for it. Similarly, in the fault model, at most k files are evicted, so the cost of OFF_s for f is at most $k + 1$ while the cost of OPT_b for it is 1.

Next, we show the lower bounds. For the bit model, consider an input that repeats M times a sequence of two files f and g, where $size(f) = k$, and $size(g) = 1$. We get $\text{OPT}_b = k + M$, by keeping f in the cache and bypassing g each time it is requested, while $\text{OPT}_s = (k + 1)M$, since a miss occurs on every file.

For the fault model, consider an input that repeats M times a sequence of $k + 1$ files: f_i, for $1 \leq i \leq k$, and g, where $size(f_i) = 1$, and $size(g) = k$. We get $\text{OPT}_b = k+M$, by keeping the k files f_i for $1 \leq i \leq k$ in the cache and bypassing g each time it is requested, while $\text{OPT}_s = (k + 1)M$, since a miss occurs on every file.

In the case of weighted paging, consider an input that repeats M times a sequence of $k + 1$ files f_i, for $1 \leq i \leq k$, and g, where $cost(f_i) = N$, and $cost(g) = 1$. We get $\text{OPT}_b = Nk + M$, by keeping the first

k files in the cache and bypassing g each time it is requested, while $\text{OPT}_s \geq Nk + (M-1)N$, since every subsequence of the form $g, f_1, f_2, \ldots, f_k$ results in at least one miss on a file of cost N, since at least one of the last k files in the subsequence is not in the cache after g is requested.

In the last case of paging, consider a sequence that consists of M phases, where in phase j, there are $k+1$ requests for pages $1, 2, \ldots, k, k+j$. An offline algorithm with bypassing keeps the pages $1, 2, \ldots, k$ in the cache, and bypasses all requests for other pages, that is for any page $k+j$ such that $1 \leq j \leq M$. The cost of this algorithm is $k + M$. As for an optimal solution without bypassing, we use the optimal offline algorithm of Belady [11] that on a miss, evicts the page that will be requested again furthest in the future. Each time that a page $k+j$ for some $j \geq 1$ is requested, the cache would contain pages $1, 2, \ldots, k$, so page k would be evicted. Then when page k is requested, since page $k+j$ would never be requested again, it is evicted. Therefore, for each phase, the cost of the algorithm is at least 2 (in the first phase this holds for a different reason, since the cache is initially empty). We conclude that the total cost of an optimal solution is at least $2M$. This gives a ratio of 2 between the optimal costs. $\square$

Note that the bounds above hold for arbitrarily large values of OPT_b.

Next, we discuss some relations between the different models. The next claim shows that caching with rejection generalizes both caching and caching with bypassing.

Claim 13 *For any $\mathcal{R}$-competitive algorithm* ALG *for caching with rejection, there exists an $\mathcal{R}$-competitive algorithm $\mathcal{G}($*ALG*$)$ for caching, and an $\mathcal{R}$-competitive algorithm $\mathcal{G}'($*ALG*$)$ for caching with bypassing. If* ALG *is deterministic, then so are $\mathcal{G}($*ALG*$)$ and $\mathcal{G}'($*ALG*$)$.*

Proof First, we show the existence of $\mathcal{G}'$. Given an input for caching with bypassing, we define an input for caching with rejection that contains the same requests, and the rejection penalty of a request is its cost. The resulting caching problem with rejection is equivalent to the original caching problem with bypassing, and thus the algorithm $\mathcal{G}'(\text{ALG})$ only needs to apply this transformation on the input while running ALG.

Let λ denote the total cost of all files in the slow memory. Given an input I for caching, the input I' is defined by the same sequence of file requests, where every request has a rejection penalty of 2λ. This transformation can be computed online. For an algorithm ALG, $\mathcal{G}(\text{ALG})$ applied on I maintains the same cache contents as ALG applied on I' (for every realization of the random bits). Moreover, $\mathcal{G}(\text{ALG})$ acts exactly as ALG does in cases that ALG does not reject the request. In a case that ALG rejects a file f, $\mathcal{G}(\text{ALG})$ empties the cache, inserts f, removes it, and re-inserts the previous contents of the cache. The cost for serving such a request is less than 2λ. Thus, the cost of $\mathcal{G}(\text{ALG})$ for I (for every realization of the random bits) is no larger than the cost of ALG for I'. Next, we show $\text{OPT}(I) = \text{OPT}(I')$, where $\text{OPT}(I)$ is an optimal solution for the caching problem (and the input I), and $\text{OPT}(I')$ is an optimal solution for caching with rejection (and the input I'). Obviously, $\text{OPT}(I') \leq \text{OPT}(I)$, as any solution for I defines a solution for I' (it never rejects any file, and performs the same sequence of actions). We can also define a solution for I that is based on $\text{OPT}(I')$, where this solution is simply $\mathcal{G}(\text{OPT}(I'))$. Thus, $\text{OPT}(I) \leq \text{OPT}(I')$. $\square$

Next, we prove that our assumption that all file costs are positive is not restrictive.

Claim 14 *For any $\mathcal{R}$-competitive algorithm* ALG *for caching (caching with rejection) that does not allow zero costs, there exists an $\mathcal{R}$-competitive algorithm $\mathcal{H}($*ALG*$)$ for caching (caching with rejection, respectively) that allows zero costs. If* ALG *is deterministic, then so is $\mathcal{H}($*ALG*$)$.*

Proof Given an input I, possibly with zero file costs, we define an input $\tilde{I}$, where no file of cost zero is requested more than once. For that, for every file f of cost zero, we create copies $f_1, f_2, \ldots$, all of cost zero. The process of creating $\tilde{I}$ from I (online) is as follows. If the next request of I is a file with a positive cost, then the next request of $\tilde{I}$ is the same as in I (including the rejection penalty, if I is an input for caching with rejection). Otherwise, if the next request of I is a zero cost file g, and this is the ith request for g in I, then the next request of $\tilde{I}$ is g_i (and if I is an input for caching with rejection, then it has the same rejection penalty as the current request for g in I).

We prove $\text{OPT}(I) = \text{OPT}(\tilde{I})$. Given an optimal solution for $\tilde{I}$, this solution immediately implies a solution for I, where any action on a file f_i of $\tilde{I}$ is applied on f for I. Given an optimal solution for I, the solution is adapted such that the modified solution will always have the same cache contents as $\text{OPT}(I)$, where for each zero cost file it has some instance of this file. Every request of positive cost is served as before (it is either in the cache or not in the cache of both solutions before it is served). Given a request f_i of zero cost, if $\text{OPT}(I)$ does nothing, that is, it already has f in the cache, then the modified solution must have f_j in the cache for some $j < i$, and it replaces f_j with f_i. Otherwise, the modified solution acts as the original solution (if it inserts f into the cache, then the modified solution inserts f_i). Thus, the required property of the cache contents is maintained. As f_i has cost zero, the cost of the modified solution equals $\text{OPT}(I)$.

Next, we modify $\tilde{I}$ into an input I' without zero costs as follows. Given a request for a file f_j with zero cost, such that this request is the ℓth request in the input, its cost is defined to be $\frac{1}{2^\ell}$ (the other properties of $\tilde{I}$ are unchanged).

We define $\mathcal{H}(\text{ALG})$ for I based on the action of ALG. Specifically, it constructs I' in an online fashion, and maintains the same cache contents as ALG applied on I' (for every realization of the random bits), where in cases that ALG has a file f_i in the cache, $\mathcal{H}(\text{ALG})$ simply has f. We have $\mathcal{H}(\text{ALG})(I) \leq \text{ALG}(I')$, as applying all the actions of ALG on I' results in serving all the requests of I, and the cost of a file f_i in I is no larger than the costs of f in I'.

Consider now an optimal solution for $\tilde{I}$. Using it as a solution for I', we obtain a feasible solution whose cost is larger by at most the total increase of all file costs in I' compared to $\tilde{I}$, that is by at most 1, and we find $\text{OPT}(I') \leq \text{OPT}(\tilde{I}) + 1$.

We have (for a given constant C) $\mathcal{H}(\text{ALG})(I) \leq \text{ALG}(I') \leq \mathcal{R} \cdot \text{OPT}(I') + C \leq \mathcal{R} \cdot \text{OPT}(\tilde{I}) + \mathcal{R} + C = \mathcal{R} \cdot \text{OPT}(I) + \mathcal{R} + C$. □

Finally, we show a reduction that allows us to obtain a randomized algorithm.

Theorem 6 *For any $\mathcal{R}$-competitive algorithm* ALG *for caching with cache size $2k$, there exists an $\mathcal{R}$-competitive algorithm Ψ(*ALG*) for caching with bypassing with cache size k. If* ALG *is deterministic, then so is Ψ(*ALG*).*

Proof By Claim 14, we assume that ALG acts on inputs that may contain files of cost zero.

The algorithm $\Psi(\text{ALG})$ applies a modification on the input I while running ALG on the modified input I'. Given the jth request of I for a file f, insert a request for f followed by a request for a new file x_j of size k and cost zero. Such requests are called additional requests. After ALG deals with x_j, it must have x_j in the cache (for every realization of the random bits), and the total size of other files in its cache is at most k. The state of its cache is defined to be the set of files that it has in the cache, excluding x_j. To deal with this request for f, $\Psi(\text{ALG})$ moves to the cache state of ALG after it has dealt with x_j (this is done for every realization of the random bits), and $\Psi(\text{ALG})$ bypasses f if it is not in the cache. It is left to show that the two algorithms have the same cost (for their respective inputs). For every request f of I, the contents of the cache of $\Psi(\text{ALG})$ and the state of the cache of ALG (after the additional request) is the same (for every realization of the random bits). Thus, their cost for f must be the same. Since the cost of ALG on the additional requests is zero, the total costs are the same as well.

Moreover, we can show that the costs of optimal solutions (an optimal solution for caching with bypassing, the input I, and a cache of size k, and an optimal solution for caching, the augmented input, and a cache of size $2k$) are equal. By the same arguments $\Psi(\text{OPT})(I) = \text{OPT}(I')$, and so $\text{OPT}(I) \leq \text{OPT}(I')$.

Consider an optimal solution for I, and create a solution for I' (and a cache of size $2k$) as follows. The first k slots of the cache are called reserved and are never used for the additional requests. The other k slots are called additional slots, and they are used for all the additional requests. The solution for I' imitates the solution for I, and always has the files of the solution for I in the reserved slots. Whenever the solution for I bypasses a file, the solution for I' inserts it into the additional slots (the number of additional slots may be larger than the size of the file). Thus, after an additional request, the caches have the same files (neglecting an additional request that occupies all the additional slots of the larger cache). We described a solution for I' of cost $\text{OPT}(I)$, and thus $\text{OPT}(I') \leq \text{OPT}(I)$. $\square$

Corollary 1 *There is a randomized $O(\log k)$-competitive algorithm for caching with rejection.*

Proof Given the randomized $O(\log k)$-competitive algorithm of [2] for caching and the reduction of Theorem 6, we find that there exists a randomized $O(\log 2k) = O(\log k)$-competitive algorithm for caching with bypassing. Using the first reduction, Theorem 2, we find that there exists a randomized $O(\log k)$-competitive algorithm for caching with rejection. $\square$

Acknowledgments. The authors would like to thank an anonymous reviewer who suggested the reduction presented in Theorem 6, and thus allowed us to simplify the randomized result significantly, making the paper much more elegant.

References

[1] D. Achlioptas, M. Chrobak, and J. Noga. Competitive analysis of randomized paging algorithms. *Theoretical Computer Science*, 234(1-2):203–218, 2000.

[2] A. Adamaszek, A. Czumaj, M. Englert, and H. Räcke. An O(log k)-competitive algorithm for generalized caching. In *Proc. of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA2012)*, pages 1681–1689, 2012.

[3] S. Albers. New results on web caching with request reordering. *Algorithmica* 58(2):461–477, 2010.

[4] S. Albers, S. Arora, and S. Khanna. Page replacement for general caching problems. In *Proc. of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA1999)*, pages 31–40, 1999.

[5] N. Bansal, N. Buchbinder, and J. Naor. Towards the randomized k-server conjecture: A primal-dual approach. In *Proc. of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA2010)*, pages 40–55, 2010.

[6] N. Bansal, N. Buchbinder, and S. Naor. A primal-dual randomized algorithm for weighted paging. *Journal of the ACM*, 59(4): article 19 (24 pages), 2012.

[7] N. Bansal, N. Buchbinder, and S. Naor. Randomized competitive algorithms for generalized caching. *SIAM Journal on Computing*, 41(2): 391–414, 2012.

[8] 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(5):1069–1090, 2001.

[9] Y. Bartal, S. Leonardi, A. Marchetti-Spaccamela, J. Sgall, and L. Stougie. Multiprocessor scheduling with rejection. *SIAM Journal on Discrete Mathematics*, 13(1):64–78, 2000.

[10] W. W. Bein, L. L. Larmore, and J. Noga. Equitable revisited. In *Proc. of the 15th Annual European Symposium on Algorithms (ESA2007)*, pages 419–426, 2007.

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

[12] A. Blum, C. Burch, and A. Kalai. Finely-competitive paging. In *Proc. of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS1999)*, pages 450–458, 1999.

[13] M. Brehob, R. J. Enbody, E. Torng, and S. Wagner. On-line restricted caching. *Journal of Scheduling*, 6(2):149–166, 2003.

[14] G. S. Brodal, G. Moruz, and A. Negoescu. OnlineMin: A fast strongly competitive randomized paging algorithm. In *Proc. of the 9th International Workshop on Approximation and Online Algorithms (WAOA2011)*, pages 164–175, 2011.

[15] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *Proc. of the USENIX Symposium on Internet Technologies and Systems*, pages 193–206, 1997.

[16] M. Chrobak, H. J. Karloff, T. H. Payne, and S. Vishwanathan. New results on server problems. *SIAM Journal on Discrete Mathematics*, 4(2):172–181, 1991.

[17] M. Chrobak and J. Noga. Competitive algorithms for relaxed list update and multilevel caching. *Journal of Algorithms*, 34(2):282–308, 2000.

[18] M. Chrobak, G. J. Woeginger, K. Makino, and H. Xu. Caching is hard - even in the fault model. *Algorithmica*, 63(4):781–794, 2012.

[19] E. Cohen and H. Kaplan. Caching documents with variable sizes and fetching costs: An LP-based approach. *Algorithmica*, 32(3):459–466, 2002.

[20] G. Dósa and Y. He. Bin packing problems with rejection penalties and their dual problems. *Information and Computation*, 204(5):795–815, 2006.

[21] L. Epstein. Bin packing with rejection revisited. *Algorithmica*, 56(4):505–528, 2010.

[22] L. Epstein, A. Levin, and G. J. Woeginger. Graph coloring with rejection. *Journal of Computer and System Sciences*, 77(2):439–447, 2011.

[23] L. Epstein, J. Noga, and G. J. Woeginger. On-line scheduling of unit time jobs with rejection: minimizing the total completion time. *Operetions Research Letters*, 30(6):415–420, 2002.

[24] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.

[25] M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 24(2):296–317, 1995.

[26] S. Irani. Page replacement with multi-size pages and applications to web caching. *Algorithmica*, 33(3):384–409, 2002.

[27] A. Karlin, M. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1–4):79–119, 1988.

[28] R. M. Karp. On-line algorithms versus off-line algorithms: How much is it worth to know the future? In J. van Leeuwen, editor, *IFIP Congress (1)*, volume A-12 of *IFIP Transactions*, pages 416–429. North-Holland, 1992.

[29] C. Koufogiannakis and N. E. Young. Greedy Δ-approximation algorithm for covering with arbitrary constraints and submodular cost. *Algorithmica*, 66(1):113–152, 2013.

[30] M. S. Manasse, L. A. McGeoch, and D. D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11(2):208–230, 1990.

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

[32] M. Mendel and S. S. Seiden. Online companion caching. *Theoretical Computer Science*, 324(2-3): 183–200, 2004.

[33] J. Nagy-György and Cs. Imreh. Online scheduling with machine cost and rejection. *Discrete Applied Mathematics*, 155(18):2546–2554, 2007.

[34] S. S. Seiden. Preemptive multiprocessor scheduling with rejection. *Theoretical Computer Science*, 262(1):437–458, 2001.

[35] D. D. Sleator and R. E. Tarjan. Amoritzed efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[36] N. E. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, 1994.

[37] N. E. Young. On-line file caching. *Algorithmica*, 33(3):371–383, 2002.