File Caching with Rental Cost and Zapping

Monik Khare       Neal E. Young

Abstract

The file caching problem is defined as follows. Given a cache of size $k$ (a positive integer), the goal is to minimize the total retrieval cost for the given sequence of requests to files. A file $f$ has size $\text{size}(f)$ (a positive integer) and retrieval cost $\text{cost}(f)$ (a non-negative number) for bringing the file into the cache. A miss or fault occurs when the requested file is not in the cache and the file has to be retrieved into the cache by paying the retrieval cost, and some other file may have to be removed (evicted) from the cache so that the total size of the files in the cache does not exceed $k$.

We study the following variants of the online file caching problem. **Caching with Rental Cost (or Rental Caching):** There is a rental cost $\lambda$ (a positive number) for each file in the cache at each time unit. The goal is to minimize the sum of the retrieval costs and the rental costs. **Caching with Zapping:** A file can be zapped by paying a zapping cost $N \geq 1$. Once a file is zapped, all future requests of the file don’t incur any cost. The goal is to minimize the sum of the retrieval costs and the zapping costs.

We study these two variants and also the variant which combines these two (rental caching with zapping). We present deterministic lower and upper bounds in the competitive-analysis framework. We study and extend the online covering algorithm from [19] to give deterministic online algorithms. We also present randomized lower and upper bounds for some of these problems.

1 Introduction

1.1 Background

The file caching (or generalized caching) problem is defined as follows. Given a cache of size $k$ (a positive integer), the goal is to minimize the total retrieval cost for the given sequence of requests to files. A file $f$ has size $\text{size}(f)$ (a positive integer) and retrieval cost $\text{cost}(f)$ (a non-negative number) for bringing the file into the cache. A miss or fault occurs when the requested file is not in the cache and the file has to be brought into the cache by paying the retrieval cost. When a file is retrieved into the cache, some other file may have to be removed (evicted) from the cache so that the total size of the files in the cache does not exceed $k$. Weighted caching (or weighted paging) is the special case when each file has size 1. Paging is the special case when each file has size 1 and the retrieval cost for each file is 1.

An algorithm is online if its response for each request is independent of all future requests. Let $\text{ALG}(\sigma)$ be the cost of an algorithm ALG on request sequence $\sigma$, and let $\text{OPT}(\sigma)$ be the corresponding optimal offline cost. ALG is $\alpha$-competitive if, for every request sequence $\sigma$, $\text{ALG}(\sigma) \leq \alpha \cdot \text{OPT}(\sigma) + c$, where $c$ is a constant independent of the request sequence.

In this paper, we study the following variants of the file caching problem in the online setting using the competitive-analysis framework [13].
Definition 1. Caching with Rental Cost (or Rental Caching): There is a rental cost $\lambda$ (a positive number) for each file in the cache at each time step. The goal is to minimize the sum of the retrieval costs and the rental costs. In our mode, we allow time steps with no requests.

Chrobak [7] proposes the rental caching problem and also presents some preliminary results. Weighted rental caching (or, weighted rental paging) is a special case of the rental caching problem where each file has size 1. Rental paging is a special case where each file has size 1 and the retrieval cost for each file is 1.

The rental caching problem is motivated by the idea of energy efficient caching. Caching systems can save power by turning off the memory block that are not being used to store any files. Rental Caching models this by charging a rental cost for keeping each file in the cache. See [15] for specific applications.

In section 3.2 we show that the variant of rental caching where the cache has infinite size, is closely related to the ski-rental problem. The ski-rental problem is the following. A pair of skis can be rented by paying $\lambda$ per day, or can be bought for the remainder of the ski season by paying $B$. It is not known when the season is going to end and the goal is to minimize the total money spent for the entire season [12].

Definition 2. Caching with Zapping: There is an additional cache of infinite size and any file can be added to this cache by paying a cost $N$ (a positive number greater than or equal to 1) at any time step. When a file is placed into this additional cache, we say the file has been zapped. A miss or fault occurs only when the requested file is not present in either cache. Thus, any future requests to a file in the additional cache do not incur any cost. The goal is to minimize the sum of the retrieval costs and the zapping costs.

Weighted caching with zapping (or, weighted paging with zapping) is a special case of the caching with rental cost problem where each file has size 1. Paging with zapping is a special case where each file has size 1 and the retrieval cost for each file is 1.

These variants generalize the file caching problem. File caching is a special case of rental caching where the rental cost is 0. Similarly, caching is a special case of caching with zapping where the cost of zapping is arbitrarily large. We also study the variant which combines these two variants: rental caching with zapping. In our model, there is no rental cost for files in the additional cache. Only the files in the size $k$ primary cache have to pay the rental cost.

1.2 Previous work

In 1985 Sleator and Tarjan [17] introduced the competitive-analysis framework. In [17] they show that the well-known paging rules like LEASTRECENTLYUSED (LRU), FIRSTINFIRSTOUT (FIFO), and FLUSHWHENFULL (FWF) are $k$-competitive and that $k$ is the best ratio any deterministic online algorithm can achieve for the paging problem.

Fiat et al. [10] initiate the competitive analysis of paging algorithms in the randomized setting. They show a lower bound of $H_k$, where $H_k$ is the $k$th harmonic number, for any randomized algorithm. They give a $2H_k$-competitive RANDOMIZEDMARKING algorithm. Achlioptas et al. [1] show that the tight competitive ratio of RANDOMIZEDMARKING is $2H_k - 1$. McGeoch and Sleator [16] and Achlioptas et al. [1] give optimal $H_k$-competitive randomized algorithms for paging.

For weighted caching, Chrobak et al. [8] give a tight $k$-competitive deterministic algorithm. For the randomized case, Bansal et al. [3] give a tight $O(\log k)$-competitive primal-dual algorithm.
For file caching, Irani [11] show that the offline problem is NP-hard. For the online case, Irani [11] give results for the bit model (cost(f) = size(f) for each file f) and fault model (cost(f) = 1 for each file f). She shows that LRU is (k + 1)-competitive for both models. Cao and Irani [6] extend the result to file caching. Young [19] independently gives LANDLORD algorithm and shows that it is k-competitive for the file caching problem. Irani [11] gives an $O(\log k)$-competitive randomized algorithm for bit and fault models. Bansal et al. [3] give an $O(\log k)$-competitive randomized algorithm for both the models, and an $O(\log^2 k)$-competitive randomized algorithm for the general case.

Young [18] uses online primal-dual analysis to give a k-competitive deterministic online algorithm for weighted-caching. Bansal et al. [3, 4], Buchbinder and Naor [5] use online primal-dual approach to give randomized algorithms for the paging, weighted caching, and file caching problems. In a recent work, Adamaszek et al. [2] builds on their online primal-dual approach to give an $O(\log k)$-competitive for the general case. In another recent work Epstein et al. [9] show that this online primal-dual approach can be extended to Caching with Rejection. Caching with rejection is a variant of file caching where a request to a file, that is not in the cache, can be declined by paying a rejection penalty. In this variant, each request is specified as a pair $(f, r)$, where $f$ is the file requested and $r$ is the rejection penalty. Note that, caching with rejection is different from caching with zapping. In caching with zapping, a file can be zapped at any time step, while in caching with rejection, a file can be rejected only at the time step when it is requested. Moreover, a rejected file can incur retrieval cost or rejection penalty again in the future, while the zapped file does not incur any cost after it is zapped.

Koufogiannakis and Young [14] present a deterministic greedy $\Delta$-approximation algorithm for any covering problem with a submodular and non-decreasing objective function, and with arbitrary constraints that are closed upwards, such that each constraint has at most $\Delta$ variables. They show that their algorithm is $\Delta$-competitive for the online version of the problem where the constraints are revealed one at a time. Many online caching and paging problems reduce to online covering, and consequently, their algorithm generalizes many classical deterministic algorithms for these problems. These include LRU and FWF for paging, BALANCE and GREEDY DUAL for weighted caching, LANDLORD (a.k.a. GREEDY DUAL SIZE) for file caching, and algorithms for CONNECTION CACHING [14]. We study this approach and extend it to give deterministic online algorithms for the variants of online file caching studied in this paper.

1.3 Our contributions

We study rental caching, caching with zapping, and rental caching with zapping. We present deterministic and randomized lower and upper bounds for these new variants of paging, weighted caching, and caching in the online setting. We use the approach in [14] to give deterministic algorithms for these online problems. While this approach is general, it doesn’t necessarily give optimal online algorithms. The direct application of this approach yields sub-optimal algorithms in some of the cases we study in this paper. We describe these scenarios and also the appropriate modifications to the algorithm to achieve better competitive ratios.

Table 1.3 presents the summary of the results in this paper.

For rental paging and for fault model, the deterministic upper and lower bounds in this paper are tight within constant factors. For the randomized case, the lower and upper bounds are tight within constant factors when $\lambda$ is $O(\frac{1}{k \log k})$ and when $\lambda \geq \frac{1}{k}$. For weighted rental paging and for rental caching, the upper and lower bounds are tight within constant factors when $\lambda < \frac{1}{k}$ for the
deterministic case, and when \( \lambda = O\left(\frac{1}{k^2 H_k}\right) \) or when \( \lambda \geq \frac{1}{k} \) for the randomized case. The bounds for the variants with rental cost are within constant factors of the bounds for the variants without rental cost when \( \lambda \leq \frac{1}{k^2} \) for the deterministic case and when \( \lambda = O\left(\frac{1}{k^2 H_k}\right) \) for the randomized case. For higher values of \( \lambda \geq \frac{1}{k} \) we show constant lower bounds and matching upper bounds.

For paging with zapping, weighted paging with zapping, and caching with zapping, the deterministic lower and upper bounds in this paper are tight with constant factors.

### 1.4 Other work on rental paging

Lopez-Ortiz and Salinger [15], in an independent work, study the rental paging problem. They give a deterministic polynomial time algorithm for the offline problem by reducing it to interval weighted interval scheduling. They show that any conservative or marking algorithm is \( k \)-competitive and that the bound is tight. An algorithm is conservative if it incurs at most \( k \) faults on any consecutive subsequence of requests that contains at most \( k \) distinct pages. A marking algorithm marks each page when it is requested, and when it is required to evict a page, it evicts an unmarked page. If there are no unmarked pages, it first unmarks all the pages and then removes one.

For any online algorithm \( A \) for paging, define the algorithm \( A_d \) for rental paging as follows. \( A_d \) behaves like \( A \) with the modification that any page in the cache that has not been requested for

| Problem                      | Lower Bound | Upper Bound |
|------------------------------|-------------|-------------|
| Rental paging                | \( \lambda \geq \frac{1}{k} \) | \( 2 - \lambda \) | 2 |
| Deterministic                | \( \frac{1}{k^2} \leq \lambda < \frac{1}{k^2} + \frac{k^2 H_k}{2} \) | \( \frac{k^2 H_k}{2} \) | \( k \) |
| Randomized                   | \( \lambda < \frac{1}{k^2} \) | \( \frac{k^2 H_k}{2} \) | \( H_k + \frac{e}{e-1} \) |
| Weighted rental paging       | \( \lambda \geq \frac{1}{k} \) | \( 2 - \lambda \) | \( k \) |
| Deterministic                | \( \lambda < \frac{1}{k} \) | \( \frac{k^2 H_k}{2} \) | \( H_k + \frac{e}{e-1} \) |
| Randomized                   | \( \lambda < \frac{1}{k} \) | \( \frac{k^2 H_k}{2} \) | \( H_k + \frac{e}{e-1} \) |
| Rental caching               | Same as weighted rental paging | Same as rental paging |
| Rental caching: fault model  | Same as rental paging |
| Paging with zapping          | \( \frac{N}{(2k+1)(k+1)} \) | \( \min(N, 2k+1) \) |
| Deterministic                | \( \frac{N}{(2k+1)(k+1)} \) | \( \min(N, 2k+1) \) |
| Weighted Paging with zapping | Same as paging with zapping |
| Deterministic                | \( \frac{N}{(2k+1)(k+1)} \) | \( \min(N, 2k+1) \) |
| Weighted Paging with zapping | Same as paging with zapping |
| Deterministic                | \( \frac{N}{(2k+1)(k+1)} \) | \( \min(N, 2k+1) \) |
| Rental caching with zapping  | Same as weighted rental paging with zapping |
| Deterministic                | \( \frac{N}{(2k+1)(k+1)} \) | \( \min(N, 2k+1) \) |
| Weighted rental paging with zapping | 2k + 1 |
| Deterministic                | \( \frac{N}{(2k+1)(k+1)} \) | \( \min(N, 2k+1) \) |
| Rental caching with zapping  | Same as weighted rental paging with zapping |
| Deterministic                | \( \frac{N}{(2k+1)(k+1)} \) | \( \min(N, 2k+1) \) |
| Rental caching with zapping: fault model | Same as rental paging |

### Table 1: Competitive ratios in this paper

| Problem                      | Lower Bound | Upper Bound |
|------------------------------|-------------|-------------|
| Rental paging                | \( \lambda \geq \frac{1}{k} \) | \( 2 - \lambda \) | 2 |
| Deterministic                | \( \frac{1}{k^2} \leq \lambda < \frac{1}{k^2} + \frac{k^2 H_k}{2} \) | \( \frac{k^2 H_k}{2} \) | \( k \) |
| Randomized                   | \( \lambda < \frac{1}{k^2} \) | \( \frac{k^2 H_k}{2} \) | \( H_k + \frac{e}{e-1} \) |
| Weighted rental paging       | \( \lambda \geq \frac{1}{k} \) | \( 2 - \lambda \) | \( k \) |
| Deterministic                | \( \lambda < \frac{1}{k} \) | \( \frac{k^2 H_k}{2} \) | \( H_k + \frac{e}{e-1} \) |
| Randomized                   | \( \lambda < \frac{1}{k} \) | \( \frac{k^2 H_k}{2} \) | \( H_k + \frac{e}{e-1} \) |
| Rental caching               | Same as weighted rental paging | Same as rental paging |
| Rental caching: fault model  | Same as rental paging |
| Paging with zapping          | \( \frac{N}{(2k+1)(k+1)} \) | \( \min(N, 2k+1) \) |
| Deterministic                | \( \frac{N}{(2k+1)(k+1)} \) | \( \min(N, 2k+1) \) |
| Weighted Paging with zapping | Same as paging with zapping |
| Deterministic                | \( \frac{N}{(2k+1)(k+1)} \) | \( \min(N, 2k+1) \) |
| Weighted Paging with zapping | Same as paging with zapping |
| Deterministic                | \( \frac{N}{(2k+1)(k+1)} \) | \( \min(N, 2k+1) \) |
| Rental caching with zapping  | Same as weighted rental paging with zapping |
| Deterministic                | \( \frac{N}{(2k+1)(k+1)} \) | \( \min(N, 2k+1) \) |
| Weighted rental paging with zapping | 2k + 1 |
| Deterministic                | \( \frac{N}{(2k+1)(k+1)} \) | \( \min(N, 2k+1) \) |
| Rental caching with zapping  | Same as weighted rental paging with zapping |
| Deterministic                | \( \frac{N}{(2k+1)(k+1)} \) | \( \min(N, 2k+1) \) |
| Rental caching with zapping: fault model | Same as rental paging |
$d$ steps is evicted. They define a class of online algorithms $M_d$, where $M$ is any conservative or marking algorithm. They show an upper bound of 2 on the competitive ratio of $M_{\frac{1}{k}}$ when $\lambda > \frac{1}{k}$, which matches the upper bound in this paper. They show an upper bound of $\max(k, \frac{(k+1)}{1+\lambda(k+1)})$ on the competitive ratio of $M_{\frac{1}{k}}$ when $\lambda \leq \frac{1}{k}$. This upper bound is weaker than the upper bound we present in this paper when $\frac{1}{\lambda} < \lambda < \frac{1}{k}$.

Their deterministic lower bound on the competitive ratio for rental paging matches the lower bound in this paper.

They also present experimental results for the performance of various LRU, LRU$_{\frac{1}{k}}$, FWF, FWF$_{\frac{1}{k}}$, FIFO, FIFO$_{\frac{1}{k}}$, and the optimal offline algorithm. The experimental results agree with the upper bounds shown in the paper.

They present results only for rental paging and not for weighted rental paging or rental caching. They do not study the rental paging problem in the randomized setting.

2 Online covering approach

In this section, we give a brief overview of the online covering approach from [14]. We use this approach, with modifications in some cases, to give deterministic algorithms for the variants of paging and caching problems in this paper. The idea is to reduce the given problem to online covering and then use the online covering algorithm from [14] as follows. In online covering the constraints are revealed one at a time in any order. Whenever the algorithm gets a constraint that is not yet satisfied, it raises each variable in the constraint, at the rate inversely proportional to the coefficient of the variable in the cost function, until the constraint is satisfied. This algorithm is $\Delta$-competitive, where $\Delta$ is the maximum number of variables in any constraint.

Now we illustrate this approach for the case of paging. To formulate paging as a Covering Integer Linear Program (CILP), we define the following notation and continue using it in the remainder of the paper.

- $f_t$: file requested at time $t$
- $t'$: time of next request to the file requested at time $t$
- $x_t$: indicator variable for the event that the file requested at time $t$ was evicted before $t'$
- $R(t)$: set of times of the most recent request to each file until and including time $t$
- $Q(t) = \{Q \subseteq R(t) - \{t\} : |Q| = k\}$. That is, $Q(t)$ represents all possible ways that the cache can be full when $f_t$ is requested at time $t$.
- $T$: time of last request

We formulate paging as follows (LP-Paging):

$$\min \sum_{t=1}^{T} x_t$$

s.t. $\forall t, \forall Q \in Q(t) : \sum_{s \in Q} \lfloor x_s \rfloor \geq 1$

We formulate paging as follows (LP-Paging):
Each constraint represents the following. At time \( t \), when \( f_t \) is requested, for any subset \( Q \) of \( Q(t) \), it must be true that at least of the files, corresponding to the times in \( Q \), must be evicted to make space for \( f_t \). Clearly, any feasible solution to the paging problem, is a feasible solution to Paging-LP. In particular, any optimal solution to the paging problem, is a feasible solution to Paging-LP. For any variable \( x \), \( x^* \) denotes the value of \( x \) in the optimal solution.

Now we describe the CILP based algorithm for paging. Note that, at each time step the algorithm may get multiple constraints. The algorithm considers the constraints in arbitrary order. When it gets a constraint that is not yet satisfied, it raises each variable on the constraint at unit rate until the constraint is satisfied. Whenever a variable reaches 1, the algorithm evicts the corresponding file from the cache. If the algorithm gets a constraint that is already satisfied, the algorithm does not do anything. We say, the algorithm does work on a constraint, if it wasn’t already satisfied and the algorithm raises the variables in the constraint, as described above, to satisfy it.

Each constraint in LP-Paging has exactly \( k \) variables. Now we show that this algorithm is \( k \)-competitive using the following potential function.

\[
\phi = \sum_t \max(x_t^* - x_t)
\]

Initially, \( \phi = \text{OPT} \) and \( \text{ALG} = 0 \). When the algorithm gets a constraint that is not satisfied, it raises each variable in the constraint at rate 1. So, the cost of the algorithm increases at the rate \( k \). Also, \( \phi \) decreases at unit rate because there is at least one variable \( x_s \) in the constraint such that \( x_s < x_s^* \) (otherwise the constraint would already be satisfied). Thus, the algorithm maintains the invariant \( \text{ALG}/k + \phi \leq \text{OPT} \). Since, \( \phi \geq 0 \), \( \text{ALG} \leq k \cdot \text{OPT} \).

For the variants in this paper, we use the approach outlined above, but with modifications in some cases. When we use the algorithm without any modifications, we omit the proofs for the competitive ratio. For these cases, the competitive ratio is the maximum number of variables in any constraint on which the algorithm does some work. If we apply any modifications, we present complete proofs.

3 Rental caching

3.1 Deterministic algorithms using CILP

In this section, we present a deterministic algorithm, RentalPagingCILP, for rental paging, and then extend the algorithm to rental caching. Our algorithm is based on the greedy online covering algorithm outlined in Section 2. We use the notation defined in Section 2. In addition, we define the following indicator variable to account for renting files.

- \( y_{t,s} \) : indicator variable for the event that the file requested at time \( t \) pays the rental cost at time \( s < t' \)
The following is the formulation for rental paging (LP-Rental-Paging):

$$\min \sum_{t=1}^{T} (x_t + \lambda \sum_{t \leq s < t'} y_{t,s})$$

s.t. \(\forall t, \forall Q \in Q(t) : \sum_{s \in Q} \lfloor x_s \rfloor \geq 1\) \hspace{1cm} (I)

\(\forall t, t \leq s < t' : \lfloor y_{t,s} \rfloor + \lfloor x_t \rfloor \geq 1\) \hspace{1cm} (II)

The first set of constraints (I) enforce the cache size at time \(t\) (same as the constraints in LP-Paging), and the second set of constraints (II) say that either a file has been evicted or it is being rented at time \(s\). We denote them by cache-size constraints and rent-evict constraints, respectively.

For each request, RentalPagingCILP gets some cache-size constraints and some rent-evict constraints. It considers the rent-evict constraints before the cache-size constraints. Whenever it gets a constraint that is not satisfied, it raises each variable in the constraint at the rate inversely proportional to its cost in the objective, until the constraint is satisfied. So, the algorithm raises \(x_s\) at unit rate and \(y_{t,s}\) at rate \(\frac{1}{\lambda}\).

For some \(\gamma > 0\), RentalPagingCILP\(\gamma\) is the algorithm that behaves like RentalPagingCILP with the following modification. RentalPagingCILP\(\gamma\) raises \(y_{t,s}\) at the modified rate of \(\gamma \lambda\). Note that, RentalPagingCILP\(1\) and RentalPagingCILP are the same algorithm.

**Theorem 3.1.** For rental paging, (a) RentalPagingCILP is 2-competitive when \(\lambda \geq \frac{1}{k}\), (b) RentalPagingCILP\(k\lambda\) is \((1 + \frac{1}{k^2})\)-competitive when \(\frac{1}{k^2} < \lambda < \frac{1}{k^2}\), and (c) RentalPagingCILP is \(k\)-competitive when \(\lambda \leq \frac{1}{k^2}\).

**Proof.** (a) \(\frac{1}{k} \leq \lambda\): We claim that, at any given time, if all the rent-evict constraints are satisfied, the cache-size constraints are satisfied too. We prove this by showing that each file is evicted within \(k\) steps from its latest request, by considering just the rent-evict constraints. At any given time, the algorithm considers the rent-evict constraint corresponding to each file in the cache. In the rent-evict constraint at time \(t\), when \(y_{t,s}\) goes from 0 to 1, \(x_s\) increases by \(\lambda\). So, if the file has been in the cache for \(t\) time steps since its latest request, \(x_s = t\lambda\), which is at least 1 for \(t \geq \frac{1}{k}\). Since, \(\frac{1}{k} \leq k\), \(x_s\) will be 1 in at most \(k\) steps. Thus, the algorithm does work only on rent-evict constraints, each of which has exactly 2 variables. So, RentalPagingCILP is 2-competitive.

(b) \(\frac{1}{k^2} < \lambda < \frac{1}{k^2}\): When the algorithm considers a rent-evict constraints, it raises \(x_s\) at unit rate, but raises \(y_{t,s}\) at rate \(\frac{1}{k}\), where \(\gamma = k\lambda\). The increment in \(x_s\) is \(\frac{1}{k}\) at each time step. So, for \(\gamma \leq k\lambda\), within \(k\) steps \(x_s \geq 1\) and hence the corresponding file is evicted. Thus, like in the previous case, the algorithm never does any work on the cache-size constraints. Now we show that this algorithm is \((1 + \frac{1}{k^2})\)-competitive. The proof is similar to the proof in Section 2. We use the following potential function for our proof:

\[\phi = \sum_{t=1}^{T} \left( \max (x^*_t - x_t, 0) + \sum_{t \leq s < t'} \lambda \max (y^*_{t,s} - y_{t,s}, 0) \right)\]

Consider the rent-evict constraint at time \(s\) for the file whose most recent request was at time \(t\). When the algorithm raises the variables in the constraint, the cost of the algorithm increases at the rate \((1 + \gamma)\). Also, \(\phi\) decreases at the rate \(\min(1, \gamma)\). Thus, the algorithm maintains the invariant
\[ \text{ALG}/(1 + \gamma) + \phi/(\min(1, \gamma)) \leq \text{OPT}. \] It is true initially, because ALG = 0 and \( \phi = \text{OPT} \). Since, \( \phi \geq 0 \), this implies that ALG \( \leq \frac{1 + \gamma}{\min(1, \gamma)} \) OPT. Also, \( \gamma = k\lambda \leq 1 \). So, ALG \( \leq (1 + \frac{1}{k\lambda}) \) OPT.

(c) \( \lambda \leq \frac{1}{k^2} \): In this case, RentalPagingCILP does work on both cache-size constraints and rent-evict constraints, and thus, the algorithm is \( k \)-competitive.

Now we extend the results to rental caching and present the algorithm RentalCachingCILP. For rental caching, the linear program is similar to the linear program for rental paging, with appropriate changes to take into account the cost and the size of each file. We define \( Q(t) \) to take into account the file sizes as follows. \( Q(t) = \{ Q \subseteq R(t) - \{ t \} : k - \text{size}(f) < \text{size}(Q) \leq k \} \), where \( \text{size}(Q) = \sum_{t \in Q} \text{size}(f_t) \). We modify the objective to take into account the cost of files. The following is the formulation for rental caching (LP-Rental-Caching):

\[
\begin{aligned}
\min & \sum_{t=1}^{T} (\text{cost}(f_t) \cdot x_t + \lambda \sum_{t \leq s < t'} y_{t,s}) \\
\text{s.t.} & \forall t, \forall Q \in Q(t) : \sum_{s \in Q} \text{size}(f_s) \cdot |x_s| \geq \text{size}(f_t) \\
& \forall t, t \leq s < t' : |y_{t,s}| + |x_t| \geq 1
\end{aligned}
\]

When RentalCachingCILP gets a rent-evict constraint that is not yet satisfied, it raises \( x_t \) at rate \( \frac{1}{\text{cost}(f_t)} \) and \( y_{t,s} \) at rate \( \frac{1}{\lambda} \). When it gets a cache-size constraint, it raises \( x_s \) at rate \( \frac{1}{\text{cost}(f_s)} \).

**Theorem 3.2.** RentalCachingCILP is \( k \)-competitive for rental caching.

**Proof.** Since each file has size at least 1, each constraint has at most \( k \) variables. So, for the general case of rental caching, the algorithm is \( k \)-competitive.

**Corollary 3.1.** For rental caching for the case of fault model, (a) RentalCachingCILP is 2-competitive when \( \lambda \geq \frac{1}{k} \), (b) RentalCachingCILP \( k\lambda \) is \( (1 + \frac{1}{k\lambda}) \)-competitive when \( \frac{1}{k^2} < \lambda < \frac{1}{k} \), and (c) RentalCachingCILP is \( k \)-competitive when \( \lambda \leq \frac{1}{k^2} \).

**Proof.** For the fault model, \( \text{cost}(f) \) is 1 for each file \( f \). So, the cost function and the rent-evict constraints are the same as in case of rental paging with zapping. Thus, the three cases of Theorem 3.1 still hold.

### 3.2 Rental caching with infinite cache

Consider the special case of the rental paging (or caching) problem where the cache has infinite size. This is equivalent to the rental caching problem without any cache size constraint. Even though there is no cache size constraint, this problem is still interesting because there is a rental cost for keeping files in the cache.

**Theorem 3.3.** If there is an \( \alpha \)-competitive algorithm ALG\(_{SR}\) for ski-rental, then there is a \( (\frac{\lambda + \alpha}{\lambda + 1}) \)-competitive algorithm for rental caching with infinite cache.

**Proof.** Consider any file \( f \). We define a phase as follows. A phase starts with a request to \( f \) and ends at the time step just before the next request to \( f \). When a \( f \) is requested, it is either already in the cache or it is retrieved and added to the cache. Thus, once a phase starts, the file must be present in the cache and the earliest this file can be evicted from the cache is at the next time step.
Such a phase, excluding the first step, reduces to ski-rental as follows. The cost of renting is $\lambda$ and cost of buying is the cost of eviction, which is $cost(f)$. The algorithm doesn’t know when the phase ends and at each time step it has to decide if it keep renting the file or if it should pay for the eviction cost to buy it.

The algorithm $ALG_\infty$ for rental caching with infinite cache does the following. For a request to file $f_t$ at time $t$, $ALG_\infty$ brings the file into the cache. Starting at the next time step, it simulates $ALG_{SR}$ on $f_t$ to decide for how long it keeps the file in the current phase. If $ALG_{SR}$ buys $f$ at any time step during the phase, $ALG_\infty$ evicts it from the cache at that step. The total rental cost of $ALG_\infty$ is same as the total rental cost of $ALG_{SR}$ and the total eviction cost of $ALG_\infty$ is equal to the total cost of buying for $ALG_{SR}$.

Let $OPT_{SR}$ be the optimal cost of the ski-rental problem. In a phase, $ALG_\infty$ cost is at most $\lambda + \alpha \cdot OPT_{SR}$ and the optimal cost is $OPT_\infty = \lambda + OPT_{SR}$. The the competitive ratio of this algorithm is at most $\frac{\lambda + \alpha \cdot OPT_{SR}}{\lambda + OPT_{SR}}$. Since $\alpha > 1$, the competitive ratio is at most $\alpha$.

**Corollary 3.2.** $ALG_\infty$ is a 2-competitive deterministic algorithm for rental caching with infinite cache.

**Proof.** The 2-competitive deterministic algorithm for ski-rental [12] and Theorem 3.3 together imply that $ALG_\infty$ is 2-competitive.

**Corollary 3.3.** There is a $(\frac{e}{e-1})$-competitive randomized algorithm for rental caching with infinite cache.

**Proof.** The $(\frac{e}{e-1})$-competitive randomized algorithm for ski-rental [12] and Theorem 3.3 together imply that $ALG_\infty$ is $(\frac{e}{e-1})$-competitive.

**Theorem 3.4.** If there is an $\alpha$-competitive algorithm $ALG_{SR}$ for ski-rental, then there is an $\alpha$-competitive algorithm for rental paging when $\lambda \geq \frac{1}{k}$.

**Corollary 3.4.** When $\lambda \geq \frac{1}{k}$, there is a 2-competitive deterministic algorithm for rental caching.

**Corollary 3.5.** When $\lambda \geq \frac{1}{k}$, there is a $(\frac{e}{e-1})$-competitive randomized algorithm for rental caching.

### 3.3 RentalCachingMeta

**Theorem 3.5.** If there is an $\alpha$-competitive algorithm $ALG_{SR}$ for ski-rental, and a $\beta$-competitive algorithm for caching (no rental cost) $ALG_C$, then there is $(\alpha + \beta)$-competitive algorithm for rental caching.

We present the RentalCachingMeta algorithm. Our algorithm uses $ALG_{SR}$ and $ALG_C$ to generate a solution for rental caching. On an input sequence $\sigma$ and cache size $k$, RentalCachingMeta does the following. It simulates $ALG_C$ on the input sequence $\sigma$ and cache $C_1$ of size $k$. In parallel, it simulates $ALG_\infty$ on the request sequence $\sigma$ and cache $C_2$ of infinite size. $ALG_\infty$ in turn simulates $ALG_{SR}$ on each request. At any time, the cache of RentalCachingMeta contains the intersection of the files present in caches $C_1$ and $C_2$.

**Claim 3.1.** The total size of the items in the cache of RentalCachingMeta never exceeds $k$.  

9
Proof. Total size of all items in the cache of ALGC is at least the total size of all items in the cache of RentalCachingMeta. This proves our claim, because ALGC maintains the invariant that the total size of items in the cache is at most \( k \).

Claim 3.2. \( E[RentalCachingMeta] \leq E[ALG_{SR}] + E[ALG_{C}] \)

Proof. RentalCachingMeta evicts a file, when at least one of ALG_{SR} and ALG_{C} evicted the file. For each eviction, we charge the cost of eviction for RentalCachingMeta to the algorithm that evicted the file, breaking ties arbitrarily. We charge the rental cost of RentalCachingMeta to the rental cost of ALG_{SR}. This proves our claim.

Also, \( E[ALG_{SR}] \leq \alpha \cdot \text{OPT}_{SR} \leq \alpha \cdot \text{OPT} \), and \( E[ALG_{C}] \leq \beta \cdot \text{OPT}_{C} \leq \beta \cdot \text{OPT} \), where \( \text{OPT}_{SR} \) denotes the optimal cost for rental caching with infinite cache, \( \text{OPT}_{C} \) denotes the optimal cost for caching, and \( \text{OPT} \) denotes the optimal cost for rental caching. So, \( E[RentalCachingMeta] \leq (\alpha + \beta)\text{OPT} \), and hence, RentalCachingMeta is \((\alpha + \beta)-\)competitive algorithm for rental caching.

If both ALGC and ALG_{SR} are deterministic, RentalCachingMeta is also deterministic, otherwise it is randomized. Theorem 3.5 implies the following two corollaries.

Corollary 3.6. The 2-competitive deterministic online algorithm for ski-rental [12] and the \( k \)-competitive deterministic online algorithm for caching [17], give a \((k + 2)\)-competitive deterministic online algorithm for rental caching.

Corollary 3.7. The \((e - \frac{1}{e})\)-competitive randomized online algorithm for ski-rental [12] and the \( H_k \)-competitive randomized online algorithm for caching [1, 16], give a \((H_k + e - \frac{1}{e})\)-competitive randomized online algorithm for rental caching.

3.4 Lower bounds

Theorem 3.6. The competitive ratio of any deterministic algorithm for rental paging is at least (a) \( \frac{e}{e-1} \) when \( \lambda > \frac{1}{k} \), and (b) \( \frac{k + k\lambda}{1 + k\lambda} \) when \( \lambda \leq \frac{1}{k} \).

Proof. (a) \( \lambda > \frac{1}{k} \): Corollary 3.3 implies a deterministic lower bound of \( \frac{e}{e-1} \).

(b) \( \frac{1}{k} \geq \lambda \): The adversary requests files from the set \{1, 2, 3, \ldots, k + 1\}. At each step, the adversary requests a file that is not present in the cache of the algorithm. The algorithm faults at each time step and pays at least \( \lambda \) at each step. OPT pays the rental cost to keep \( k \) items in the cache at each time step and faults once in \( k \) steps. So, the ratio is at least \( \frac{k + k\lambda}{1 + k\lambda} \). For sufficiently small \( \lambda \), the ratio tends to \( k \).

Theorem 3.7. The competitive ratio of any randomized algorithm for rental paging is at least (a) \( \frac{e}{e-1} \) when \( \lambda > \frac{1}{k} \) and (b) \( \frac{H_k + k\lambda H_k}{1 + k\lambda} \) when \( \lambda \leq \frac{1}{k} \).

Proof. (a) \( \lambda > \frac{1}{k} \): Corollary 3.3 implies a randomized lower bound of \( \frac{e}{e-1} \).

(b) \( \lambda \leq \frac{1}{k} \): The adversary requests files from a set of \( k + 1 \). At each step, the adversary requests a file with uniform probability over all files except the file requested at the previous step. We split the request sequence into phases as follows. A phase is the longest request sequence with at most \( k \) distinct requests, and starts immediately after the previous phase ends.

We now show that the expected length of each phase is \( kH_k \). When \( i \) files have been requested, the probability of requesting a file that has not been requested in the phase is \( \frac{k-i}{k} \). Thus the expected length of a phase is \( \left( \sum_{i=1}^{k} \frac{k}{k-i+1} \right) = H_k \).
Next, we show that the algorithm keeps its cache full to minimize the expected cost in a phase. Assume that \( i \) distinct files have been requested in the phase and the algorithm has \( p \leq k \) files in the cache. At the next time step, the algorithm faults with a probability \( \frac{k-c+1}{k} \) and pays a rental cost \( c\lambda \). So, the expected cost of the algorithm is \( 1 + \frac{1}{k} - p\left(\frac{1}{k} - \lambda\right) \). Since, \( \frac{1}{k} \geq \lambda \), the cost is minimized when \( p = k \). So, it pays \( \frac{1}{k} \) eviction cost and \( k\lambda \) rental cost at each step. OPT pays the same rental cost, but faults once in each phase. So, for each phase, the ratio is at least \( \frac{H_k + k^2 H_k \lambda}{1 + k^2 H_k \lambda} \). For sufficiently small \( \lambda \), the ratio tends to \( H_k \).

4 Caching with zapping

4.1 Deterministic algorithms using CILP

In this section, we present a deterministic algorithm, ZAPPINGPAGINGCILP, for paging with zapping, and then extend the algorithm to caching with zapping. We introduce another indicator variable for zapping of files.

- \( z_f \): indicator variable for the event that the file \( f \) has been zapped

We formulate paging with zapping as follows (LP-Paging-Zapping):

\[
\min \sum_{t=1}^{T} x_t + N \sum_{f \in F} z_f
\]

s.t. \( \forall t, \forall Q \in Q(t) : \left( \sum_{s \in Q} \min(|x_s| + |z_{f_s}|, 1) \cdot \text{size}(f_s) \right) + \lfloor z_{f_t} \rfloor \cdot \text{size}(f_t) \geq \text{size}(f_t) \)

The constraints say that either \( f_t \) is zapped, or at least one file in the cache is either zapped or evicted. Whenever ZAPPINGPAGINGCILP gets a constraint that is not satisfied, raises \( x_s \) at unit rate and \( z_{f_s} \) at rate \( \frac{1}{N} \).

**Theorem 4.1.** ZAPPINGPAGINGCILP is \((2k + 1)\)-competitive.

**Proof.** Each constraint has \( 2k + 1 \) variables. Thus, ZAPPINGPAGINGCILP is \((2k + 1)\)-competitive. \( \square \)

Now we extend this approach to caching with zapping and present the algorithm ZAPPINGCACHINGCILP. We define \( Q(t) \) same as we defined for rental caching in section 3.1. We have the following linear program (LP-Caching-Zapping) for caching with zapping:

\[
\min \sum_{t=1}^{T} \text{cost}(f_t) \cdot x_t + N \sum_{f \in F} z_f
\]

s.t. \( \forall t, \forall Q \in Q(t) : \left( \sum_{s \in Q} \min(|x_s| + |z_{f_s}|, 1) \cdot \text{size}(f_s) \right) + \lfloor z_{f_t} \rfloor \cdot \text{size}(f_t) \geq \text{size}(f_t) \)

For each not yet satisfied constraint that ZAPPINGCACHINGCILP gets, it raises \( x_s \) at rate \( \frac{1}{\text{cost}(f)} \), and \( z_{f_s} \) at rate \( \frac{1}{N} \).
Theorem 4.2. ZAPPINGCACHINGCILP is \((2k + 1)\)-competitive.

Proof. Since each file has size at least 1, each constraint has at most \(2k + 1\) variables. Thus, the ZAPPINGCACHINGCILP is \((2k + 1)\)-competitive.

\[\square\]

Theorem 4.3. The algorithm that zaps the file when it is requested for the first time is \(N\)-competitive.

Proof. Let the total number of distinct files requested be \(T\). The total cost of OPT is at least \(T\) (1 to bring each file in the cache). The total cost of the algorithm is at most \(NT\). So, the ratio is at most \(N\).

\[\square\]

4.2 Lower Bounds

Theorem 4.4. For paging with zapping, the competitive ratio of any deterministic algorithm is at least \(\frac{2Nk + N - (k + 1)}{N + 2k}\).

Proof. The adversary maintains a set of \(k + 1\) distinct files at all times. Every time a file is zapped by the algorithm, it is replaced in the set by a new file that has never been requested by the adversary. At each time step, the adversary requests a file that is not present in the cache of the algorithm. We define a zap-phase as follows. A zap-phase ends every time a file is zapped and the following request marks the beginning of the next zap-phase. The first zap-phase starts with the first request of the input sequence. We define a round as follows. The first round starts with the first request of the input sequence. A round ends when the algorithm has zapped the all of the \(k + 1\) files that were requested in the first \(k + 1\) time steps in that round (some other files may have been zapped too). The total number of files zapped in a round is at least \(k + 1\). The adversary repeats the process for a large number of rounds.

Now we show the lower bound on the competitive ratio in each round. Consider any round. Let \(T \geq k + 1\) be the total number of files zapped by the algorithm in the round. Let \(H_j\) be the length of zap-phase \(j, 1 \leq j \leq T\).

Any deterministic algorithm faults at each time step and zaps a total of \(T\) files. So, the cost of any deterministic algorithm is at least, \(ALG = NT + \sum_{j=1}^{T} (H_j - 1) = (N - 1)T + \sum_{j=1}^{T} H_j\). Note that, \(\sum_{j=1}^{T} H_j \geq T\).

When \(\sum_{j=1}^{T} H_j = T\), the algorithm zaps each file when it is requested for the first time. In this case, adversary requests a new file at each step. OPT pays at most \(\min(1, N)\) at each step while the algorithm pays \(N\) at each step. For \(N > 1\), the ratio is at least \(N\).

Now we assume that \(\sum_{j=1}^{T} H_j > T\). Consider the offline algorithm \(F\) which, on any request sequence, does one of the following: (a) Does not zap any file, or (b) Chooses one file from the set of the first \(k + 1\) files requested and zaps it at the first step.

If \(F\) doesn’t zap any files, in the first zap-phase it pays \(k\) to bring the first \(k\) files into the cache and then pays at most \([\frac{H_j - k}{k}]\) in the remainder of the first phase. For any zap-phase \(j > 1\), \(F\) pays at most \([\frac{H_j}{k}]\).
\[ F = (k + \left\lceil \frac{H_1 - k}{k} \right\rceil) + \left( \sum_{j=2}^{T} \frac{H_j}{k} \right) \]
\[ \leq (k + \frac{H_1}{k}) + \left( \sum_{j=2}^{T} \frac{H_j}{k} + 1 \right) \]
\[ = k + T - 1 + \sum_{j=1}^{T} \frac{H_j}{k} \]

If \( F \) zaps 1 file, it incurs \( k \) faults in the first phase and 1 fault in each phase after that. It also pays \( N \) for zapping 1 file. The total cost in this case is \((k) + (T - 1) + N\).

Since \( \text{cost}(F) \geq \text{OPT} \), the competitive ratio is at least
\[
\frac{\sum_{j=1}^{T} H_j + (N - 1)T}{\min(k + T - 1 + \sum_{j=1}^{T} \frac{H_j}{k}, k + T + N - 1)}
\]

The ratio is minimized when \( k + T - 1 + \sum_{j=1}^{T} \frac{H_j}{k} = k + T + N - 1 \). Simplifying gives, \( \sum_{j=1}^{T} H_j = Nk \).

So, the ratio is at least
\[
\frac{Nk + (N - 1)T}{N + k + T - 1}
\]

For any given \( N \) and \( k \), and for \( T \geq k + 1 \), the ratio is minimized when \( T = k + 1 \). So, the competitive ratio is at least
\[
\frac{2Nk + N - (k + 1)}{N + 2k}
\]

Thus, the competitive ratio is at least \( \min(N, \frac{2Nk + N - (k + 1)}{N + 2k}) \). Note that, \( \frac{2Nk + N - (k + 1)}{N + 2k} = N \frac{2k + 1}{N + 2k} - \frac{k + 1}{N + 2k} \leq N \), for \( N \geq 1 \). Thus the competitive ratio of any deterministic algorithm for paging with zapping is at least \( \frac{2Nk + N - (k + 1)}{N + 2k} \).

5 Rental paging with zapping

5.1 Deterministic algorithm using CILP

In this section we present algorithm \texttt{RENTALZAPPINGPAGINGCILP} for rental paging with zapping.

We use the same notations defined in the previous sections. The cache size constraints are exactly the same as in case of paging with zapping. The rent-evict constraints are modified to have variables for eviction, renting, and zapping. We have the following formulation (LP-Paging-Rental-Zapping):

\[
\min \sum_{t=1}^{T} (x_t + \lambda \sum_{t \leq s < t'} y_{t,s}) + N \sum_{f \in F} z_f
\]

s.t. \( \forall t, \forall Q \in Q(t) : \left( \sum_{s \in Q} \lfloor x_s \rfloor + \lfloor z_{f,s} \rfloor \right) + \lfloor z_{f,t} \rfloor \geq 1 \) \hspace{1cm} (III)

\( \forall t, t \leq s < t' : \lfloor y_{t,s} \rfloor + \lfloor x_t \rfloor + \lfloor z_{f,s} \rfloor \geq 1 \) \hspace{1cm} (IV)
We refer to (III) and (IV) by cache-size constraints and rent-evict-zap constraints, respectively. RentalZappingPagingCILP is similar to RentalPagingCILP, and considers the rent-evict-zap constraints before the cache-size constraints. Whenever the algorithm gets a constraint that is not satisfied, it raises all the variables in the constraint as follows. It raises $x_s$ at unit rate, $y_{t,s}$ at rate $\frac{1}{2}$, and $y_f$ at rate $\frac{1}{\gamma}$.

We define RentalZappingPagingCILP$_\gamma$ as RentalZappingPagingCILP with the following modification. RentalZappingPagingCILP$_\gamma$ raises $y_{t,s}$ at the modified rate of $\frac{\gamma}{2}$, where $\gamma > 0$.

**Theorem 5.1.** For rental paging with zapping, (a) RentalZappingPagingCILP is 3-competitive when $\lambda \geq \frac{1}{2}$, (b) RentalZappingPagingCILP$_{k\lambda}$ is $(1 + \frac{2}{k\lambda})$-competitive when $\frac{1}{k^2} < \lambda < \frac{1}{k^2}$, and (c) RentalZappingPagingCILP$_\gamma$ is $(2k + 1)$-competitive when $\lambda \leq \frac{1}{k^2}$.

**Proof.** (a) $\frac{1}{k} \leq \lambda$: We claim that, at any given time, if all the rent-evict-zap constraints are satisfied, the cache-size constraints are satisfied too. We prove this by showing that each file is evicted within $k$ steps from its latest request, by considering just the rent-evict-zap constraints. At any given time, the algorithm considers the rent-evict-zap constraint corresponding to each file in the cache.

Consider the rent-evict-zap constraint at time $s$ for the file whose latest request was at time $t$. For this constraint, when $y_{t,s}$ goes from 0 to 1, $x_t$ increases by $\lambda$. So, the file has been in the cache for $s$ time steps since its latest request and $x_t = s\lambda$. This value is at least 1 if the file has been in the cache for $s \geq \frac{1}{\lambda}$ steps. Since, $k \geq \frac{1}{\lambda}$, $x_s$ is at least 1 when $s = k$, and consequently all the cache-size constraints where $x_s$ participates will be satisfied. Note that, if $z_{f,t}$ is 1 before $x_t$ is 1, the cache-size constraints are still satisfied. Thus, the algorithm does work only on rent-evict-zap constraints, each of which has exactly 3 variables. Thus, RentalZappingPagingCILP is 3-competitive.

(b) $\frac{1}{k^2} < \lambda < \frac{1}{k}$: When the algorithm considers a rent-evict-zap constraints, it raises $x_s$ at unit rate, but raises $y_{t,s}$ at rate $\frac{\gamma}{2}$, where $\gamma = k\lambda$. The increment in $x_s$ is $\frac{1}{\gamma}$ at each time step. So, for $\gamma \leq k\lambda$, within $k$ steps $x_s \geq 1$ and hence the corresponding file is evicted. Thus, like in the previous case, the algorithm never does any work on the cache-size constraints. Now we show that this algorithm is $(1 + \frac{1}{k\lambda})$-competitive. We use the following potential function for our proof:

$$\phi = \sum_{t=1}^{T} \left( \max (x_t^* - x_t, 0) + \sum_{t \leq s < t'} \lambda \max (y_{t,s}^* - y_{t,s}, 0) \right) + \sum_{f \in F} N \max (z_{f,t}^* - z_{f,t}, 0)$$

Consider the rent-evict-zap constraint at time $s$ for the file whose most recent request was at time $t$.

When the algorithm raises the variables in the constraint, the cost of the algorithm increases at the rate $(2 + \gamma)$. Also, $\phi$ decreases at the rate $\min(1, \gamma)$. Thus, the algorithm maintains the invariant $\text{ALG}/(2 + \gamma) + \phi/(\min(1, \gamma)) \leq \text{OPT}$. It is true initially, because $\text{ALG} = 0$ and $\phi = \text{OPT}$. Since, $\phi \geq 0$, this implies that $\text{ALG} \leq \frac{2 + \gamma}{\min(1, \gamma)} \text{OPT}$. Also, $\gamma = k\lambda \leq 1$. So, $\text{ALG} \leq (1 + \frac{2}{k^2})\text{OPT}$.

(c) $\lambda \leq \frac{1}{k^2}$: In this case, RentalZappingPagingCILP$_\gamma$ does work on both cache-size constraints and rent-evict-zap constraints, and thus, the algorithm is $(2k + 1)$-competitive.

To extend the algorithm to rental caching with zapping, we combine the ideas from Sections 3.1 and 4.1. We modify $Q(t)$ to account for file sizes. $Q(t) = \{Q \subseteq R(t) - \{t\} : k - \text{size}(f) < \text{size}(Q) \leq$
\( k \). The following is the formulation of rental caching with zapping (LP-Caching-Rental-Zapping):

\[
\min \sum_{t=1}^{T} \left( \text{cost}(f_t) \cdot x_t + \lambda \sum_{t \leq s < t'} y_{t,s} \right) + N \sum_{f \in F} z_f \\
\text{s.t.} \forall t, \forall Q \in Q(t) : \left( \sum_{s \in Q} \min([x_s] + [z_{f,s}], 1) \cdot \text{size}(f_s) \right) + [z_{f_t}] \cdot \text{size}(f_t) \geq \text{size}(f_t)
\]

\( \forall t, t \leq s < t' : [y_{t,s}] + [x_t] + [z_{f_s}] \geq 1 \) (IV)

When \text{RentalZappingCachingCILP} gets a rent-evict-zap constraint that is not yet satisfied, it raises \( x_t \) at rate \( \frac{1}{\text{cost}(f_t)} \), \( y_{t,s} \) at rate \( \frac{1}{\lambda} \), and \( z_{f_t} \) at rate \( \frac{1}{N} \). When it gets a cache-size constraint, it raises \( x_s \) at rate \( \frac{1}{\text{cost}(f_s)} \) and \( z_{f_s} \) at rate \( \frac{1}{N} \).

**Theorem 5.2.** \text{RentalZappingCachingCILP} is \((2k + 1)\)-competitive for rental caching with zapping.

*Proof.* Since each file has size at least 1, each cache-size constraint has at most \((2k + 1)\) variables and each rent-evict-zap constraint has exactly 3 variables. So, for the general case of caching with zapping, the algorithm is \((2k + 1)\)-competitive. \(\square\)

**Corollary 5.1.** For rental caching with zapping for the case of fault model, (a) \text{RentalZappingCachingCILP} is \(3\)-competitive when \( \lambda \geq \frac{1}{k} \), (b) \text{RentalZappingCachingCILP}_{k,\lambda} is \((1 + \frac{2}{k\lambda})\)-competitive when \( \frac{1}{k^2} < \lambda < \frac{1}{k} \), and (c) \text{RentalZappingCachingCILP} is \((2k + 1)\)-competitive when \( \lambda \leq \frac{1}{k^2} \).

*Proof.* For the fault model, \( \text{cost}(f) = 1 \) for each file \( f \). So, the cost function and the rent-evict-zap constraints are the same as in case of rental paging with zapping. Thus, the three cases of Theorem 5.1 still hold. \(\square\)

### 5.2 RentalZappingCachingMeta

Analogous to Theorem 3.5, we have the following theorem for rental caching with zapping.

**Theorem 5.3.** If there is an \( \alpha \)-competitive algorithm \( \text{ALG}_{SR} \) for ski-rental, and a \( \beta \)-competitive algorithm for caching with zapping (no rental cost) \( \text{ALG}_Z \), then there is \((\alpha + \beta)\)-competitive algorithm for caching with zapping and rental cost.

We present the RentalZappingCachingMeta algorithm. Our algorithm uses \( \text{ALG}_{SR} \) and \( \text{ALG}_Z \) to generate a solution for rental caching with zapping. On an input sequence \( \sigma \) and cache size \( k \), RentalZappingCachingMeta does the following. It simulates \( \text{ALG}_Z \) on the input sequence \( \sigma \) and cache \( C_1 \) of size \( k \). In parallel, it simulates \( \text{ALG}_\infty \) on the request sequence \( \sigma \) and cache \( C_2 \) of infinite size. \( \text{ALG}_\infty \) in turn simulates \( \text{ALG}_{SR} \) on each request. At any time, the cache of RentalZappingCachingMeta contains the intersection of the files present in caches \( C_1 \) and \( C_2 \). If \( \text{ALG}_Z \) nukes a file, RentalZappingCachingMeta nukes it.

**Claim 5.1.** The total size of the items in the cache of RentalZappingCachingMeta never exceeds \( k \).
Proof. Total size of all items in the cache of \( ALG_Z \) is at least the total size of all items in the cache of \( \text{RentalZappingCachingMeta} \). This proves our claim, because \( ALG_Z \) maintains the invariant that the total size of items in the cache is at most \( k \).

\[ \text{Claim 5.2. } E[\text{RentalZappingCachingMeta}] \leq E[\text{ALG}_{SR}] + E[\text{ALG}_Z] \]

Proof. \( \text{RentalZappingCachingMeta} \) evicts a file, when at least one of \( \text{ALG}_{SR} \) and \( \text{ALG}_Z \) evicted the file. For each eviction, charge the cost of eviction for \( \text{RentalZappingCachingMeta} \) to the algorithm that evicted the file, breaking ties arbitrarily. Charge the cost of zapping to \( \text{ALG}_Z \) and charge the rental cost to the rental cost of \( \text{ALG}_{SR} \). This proves our claim.

Also, \( E[\text{ALG}_{SR}] \leq \alpha \cdot \text{OPT}_{SR} \leq \alpha \cdot \text{OPT} \) and \( E[\text{ALG}_Z] \leq \beta \cdot \text{OPT}_Z \leq \beta \cdot \text{OPT} \), where \( \text{OPT}_{SR} \) denotes the optimal cost for rental caching with infinite cache, \( \text{OPT}_Z \) denotes the optimal cost for caching with zapping, and \( \text{OPT} \) denotes the optimal cost for rental caching with zapping. So, \( E[\text{RentalZappingCachingMeta}] \leq (\alpha + \beta)\text{OPT} \), and hence, \( \text{RentalZappingCachingMeta} \) is \( (\alpha + \beta) \)-competitive algorithm for rental caching. If both \( \text{ALG}_Z \) and \( \text{ALG}_{SR} \) are deterministic, \( \text{RentalCachingMeta} \) is also deterministic, otherwise it is randomized.

6 Conclusions and further directions

We present lower and upper bounds, in deterministic and randomized settings, for rental paging and rental caching. For most cases, the lower and upper bounds are tight up to constant factors. When \( \frac{1}{2} < \lambda < \frac{1}{\sqrt{k}} \), there is a gap between the randomized lower and upper bounds shown in this paper. The lower bounds in this paper assume that the cache of \( \text{OPT} \) is always full, and consequently, in each phase is \( \text{OPT} \)'s rental cost is no longer \( O(\text{OPT} \text{'}s eviction cost). \) It may be possible to show better lower bounds using a modified analysis or possibly by using another adversary strategy.

The deterministic lower and upper bounds for paging with zapping are tight up to constant factors. The next step would be to study randomized lower bounds and randomized algorithms for caching with zapping.

For rental caching with zapping, we present the upper bounds in both deterministic and randomized settings. It would be interesting to study the lower bounds and .

The models in this paper assume uniform rental cost and uniform zapping cost in this study. Note that, in our model for rental caching, the total rental cost depends only on the size of a file. A natural extension would be to consider models with (arbitrary) non-uniform rental and zapping costs.

The CILP based approach by Koufogiannakis and Young \cite{14} is a general and elegant approach for deriving deterministic algorithms for online paging and caching problems. We use this approach for all the new variants studied in this paper. The algorithms thus derived may not be optimal, as we show for the case of rental paging (or caching) and also for rental paging (or caching) with zapping. For the problems in this paper, we were able to apply simple modifications to achieve upper bounds within constant factors on the lower bounds.

The primal-dual approach in \cite{2, 3, 4, 5, 9} is a powerful framework for deriving randomized algorithms for online caching problems. It would be interesting to investigate if their approach can be used to give randomized algorithms for the variants studied in this paper.
7 Acknowledgments

We would like to thank Marek Chrobak and Li Yan for useful discussions.

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 *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1681–1689. SIAM, 2012.

[3] N. Bansal, N. Buchbinder, and J. Naor. A primal-dual randomized algorithm for weighted paging. In *Foundations of Computer Science, 2007. FOCS’07. 48th Annual IEEE Symposium on*, pages 507–517. IEEE, 2007.

[4] N. Bansal, N. Buchbinder, and J. Naor. Randomized competitive algorithms for generalized caching. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 235–244. ACM, 2008.

[5] N. Buchbinder and J. Naor. Online primal-dual algorithms for covering and packing. *Mathematics of Operations Research*, 34(2):270–286, 2009.

[6] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems*, volume 193, 1997.

[7] M. Chrobak. Sigact news online algorithms column 17. *ACM SIGACT News*, 41(4):114–121, 2010.

[8] M. Chrobak, H. Karloff, T. Payne, and S. Vishwanathan. New results on server problems. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 291–300. Society for Industrial and Applied Mathematics, 1990.

[9] L. Epstein, C. Imreh, A. Levin, and J. Nagy-György. On variants of file caching. *Automata, Languages and Programming*, pages 195–206, 2011.

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

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

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

[13] M. Khare and N. Young. Caching with rental cost and nuking. *Arxiv preprint arXiv:1208.2724*, 2012.
[14] C. Koufogiannakis and N. E. Young. Greedy delta-approximation algorithm for covering with arbitrary constraints and submodular cost. *Algorithmica*, 2012. Journal version of [2009].

[15] A. Lopez-Ortiz and A. Salinger. Minimizing cache usage in paging. 2012.

[16] L. McGeoch and D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6(1):816–825, 1991.

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

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

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