

# Calculating lower bounds for caching problems

Leah Epstein\*

Rob van Stee†

## Abstract

We present a general method for computing lower bounds for various caching problems. We apply the method to two well known problems, companion caching and weighted caching. For weighted caching, we increase the interval of weights where FIFO is known to be optimal. For companion caching, we give much simpler proofs for several known results, and give a new bound for the case of three types without reorganization or bypassing.

**keywords:** paging, on-line, caching.

**AMS subject classification:** 68W25, 68W40.

## 1 Introduction

Modern operating systems have multiple memory levels. In simple structures, a memory is organized in equally sized pages. The basic paging model is defined as follows. The system has a slow and large memory (e.g. disk) where all pages are stored. The second level is a small, fast memory (cache) where the system brings a page in order to use it. If a page, which is not in the faster memory level, is requested, a page fault occurs, and a page in the faster level must be evicted in order to make room to bring in the new page. The processor must slow down until the page is brought into memory, and in practice for many applications the performance of the system depends almost entirely on the number of page faults. We define the cost of an algorithm as simply the total number of page faults.

In online paging problems, the input is a request sequence. Each request arrives after the previous request was served (i.e. the decision on the eviction of another page was made). The competitive ratio is the asymptotic worst case ratio between the cost of an

---

\*Department of Mathematics, University of Haifa, 31905 Haifa, Israel. [lea@math.haifa.ac.il](mailto:lea@math.haifa.ac.il).

†Fakultät für Informatik, Universität Karlsruhe, D-76128 Karlsruhe, Germany. [vanstee@ira.uka.de](mailto:vanstee@ira.uka.de).  
Research supported by the Alexander von Humboldt Foundation.

online algorithm and the cost of an optimal offline algorithm which knows all requests in advance. For the classical offline problem, there exists a polynomial simple optimal algorithm LFD (Longest Forward Distance) [2]. LFD always evicts the page for which the time until the next request to it is maximal. See Irani [11] for a survey on online paging problems.

Two common paging algorithms for the classical paging problem are LRU (Least Recently Used) and FIFO (First In First Out). LRU computes for each page, which is present in the fast memory level, the last time it was used and evicts the page for which this time is minimum. FIFO acts like a queue, evicting the page that has spent the longest time in the fast memory. Variants of both are common in real systems. It is generally accepted that LRU based algorithms typically perform better than FIFO in practice. However, in terms of competitiveness (for the classical problem described above) LRU and FIFO are known to be equally good, both having a competitive ratio equal to the size of the fast memory level (measured in pages). Further this ratio is known to be the best possible, see [12, 15].

In this paper we focus on the construction of lower bounds. We describe methods of sequence partitioning, and show how to use computer assistance to obtain a proof which is verifiable by humans. We next describe the caching models we are interested in.

## 1.1 Two-level cache systems

In the classical problem it is assumed that all faults cause the same performance degradation (in terms of delay) to the computer and therefore have the same cost. However, this is not always the case. Many systems have two or more levels of cache [1, 6] where the cost of accessing the different levels may differ (in order to model differences in speed).

We will now define one simple model consisting of three separate memory levels. The system has one level which is slow, but contains all items. The other two levels are both of size  $\kappa$ , but the time necessary to swap items from these two levels may differ. We allow any page to be stored in either level. We refer to this model as the “two-level cache model”. The price ratio between the cost of a fault (i.e. a page replacement) in the slower cache and a fault in the faster cache is  $w \geq 1$ . This model was studied as a server problem in a uniform space [7, 10]. In those papers  $\kappa = 1$ , so we have two servers and we need to serve a sequence of requests by each time moving one server to the current request. Moving one server costs more than the other, and the ratio of the costs is  $w$ . Since the two servers are different, and each server is associated with a given slot in the cache, we assume that re-arranging the cache is not allowed, i.e., it is not allowed to move pages for free between the different levels of the cache.

**Previous results** The paper of Chrobak and Sgall [7] gives a tight bound on general algorithms for  $\kappa = 1$  (on the uniform space, where the distance between any two distinct points is 1). Specifically they give deterministic and memoryless randomized 5-competitive algorithms and matching lower bounds.

Epstein et al. [8] show some further results as a function of  $w$ . Here FIFO and LRU are simply the versions of those two algorithms that ignore the difference between the two different types of cache locations and simply treat them as one larger cache, in which re-arranging is not allowed. The paper also concentrates on the case  $\kappa = 1$ , i.e. a small two-level cache having one slot of each level. For a cost ratio  $w$  it turns out that FIFO has competitive ratio  $\max(2, \frac{3}{4}(1+w))$  whereas LRU is slightly less successful with competitive ratio  $1+w$ . The paper proves that FIFO has the optimal competitive ratio among all possible algorithms for  $1 \leq w \leq 2.2$ . This result is surprising from two points of view. The first is that one would expect an algorithm to work better if it takes the price difference into account. The second is that in another paging model (using access graphs) Chrobak and Noga [5] showed that LRU is never worse and often much better than FIFO, proving a conjecture by Borodin et al. [3].

**Our results** In this paper we extend the techniques used in Epstein et al. [8] to show that FIFO has the best competitive ratio for a much wider interval of  $w$ . This is done by introducing new mathematical ideas, and by simplifying the computer program that was used in Epstein et al. [8].

## 1.2 Companion paging systems

Another cache structure that has recently gained interest is companion caching. Unlike the basic model, where each slot in the cache may be used for all purposes, a set associative cache is divided into parts, usually of equal size, denoted by  $\kappa$ . Each part can store only certain pages. A special case, which is common in existing architectures, is direct-mapped caches, where there is only one slot of each type. The set associative cache model is very simple and allows an overload of one type. If there are many requests for one type, any algorithm would have to keep evicting and loading pages. To overcome this problem, it is more common now to use a hybrid architecture where a second small (fully associative) cache is added, i.e. each slot of this cache may store any page. This extra cache is known as the *companion cache*.

**Previous results** This combined structure has been studied in two papers [4, 9]<sup>1</sup>. There are several variants which need to be examined. Some systems allow *bypassing* which gives the algorithm an option of reading a page from the slow memory without bringing it to the cache. It is assumed that the cost of bypassing is the same as the price of a fault. It is folklore that introducing bypassing in the basic model of paging does not change the fact that LRU and FIFO are in the class of best possible online algorithms, and the best competitive ratio increases by 1.

Another sub-model allows *re-organization* (which is irrelevant to regular paging, where all slots are the same). When it is allowed, the cache may move pages (free of charge) between the main cache and the companion cache, inserting them into slots that are valid for them.

Fiat et al. [9] focus on the case with re-organization but no bypassing. Brehob et al. [4] study the three other models, but focus on the case  $\kappa = 1$  (i.e. each set in the main cache is of size one), and general  $n$  (size of companion cache). It is shown that for the two models with no re-organization, the competitive ratios are  $\Theta(n)$ . For the model with both bypassing and re-organization, only an upper bound of  $O(n)$  is given for the competitive ratio. An interesting special case, which is proved by adapting methods from formal language theory, shows a tight result of 3 for  $\kappa = 1$ , two types, and  $n = 1$ .

Fiat et al. [9] unify these models by showing that the four different competitive ratios (with/without re-organization and with/without bypassing) are within a constant multiplicative factor of each other. The main result of Fiat et al. [9] is a tight competitive ratio for the case where reorganization is allowed but bypassing is not, which is  $n\kappa + n + \kappa$ . Recall that  $n$  is the size of the companion cache and  $\kappa$  is the size given to each type in the main cache. The lower bound follows from results for simple caches while the algorithm giving the upper bound is a generalization of LRU.

**Our results** In this paper we discuss the companion caching without reorganization. We re-prove several results of Brehob et al. [4] using entirely different, and much simpler, proofs. We consider  $\kappa = 1$  as in Brehob et al. [4]. For the case without bypassing we show a new lower bound of 4.35 for  $n = 2$  and three types. The old lower bound was 3 and the upper bound is 6 [4]. We also re-prove the lower bound of 3 for two types and  $n = 1$ , and the lower bound of  $n + 1$  for  $n + 1$  types. Our proof of the lower bound of 3 significantly simplifies the proof. The proof we present is possible to verify. We use computer assistance, but the results of the execution of our program are part of the proof presented here. In contrast, the proof of Brehob et al. [4] is done entirely by a computer

---

<sup>1</sup>The paper [14] contains the same results as [9].

program. For the case with bypassing we re-prove the lower bound of 4 for  $n = 1$  and two types.

## 2 Multilevel caching systems

In this section we give a description of the general ideas and the details for the two-level cache problem. The details for companion paging are given in the next section.

We reconsider the weighted two-level caching problem from [8], and prove that FIFO is optimal in the interval  $2 \leq w \leq 2.97$ . This extends the interval where FIFO is known to be optimal by a factor of more than  $3/2$ . We add many new ingredients to the proof in [8]. According to the results of that paper, the competitive ratio of FIFO is  $\max(2, \frac{3}{4}(1+w))$ , which is  $\frac{3}{4}(1+w)$  in the interval  $[2, 2.97]$  that we are considering.

**The sequence** We construct a sequence of requests, which consists of requests for at most 3 different pages: 0, 1, 2. Then there are six possible configurations for the cache. The initial construction of the sequence and another basic ingredient of the proof, namely tracking several offline algorithms simultaneously, is similar to the lower bound for the  $k$ -server problem [13].

As in [13], the sequence of requests is constructed in such a way that the online algorithm has a fault in every request. That is, in each step a request is added for the only page that is not present in any of the two caches of the online algorithm. In order to give a lower bound on the asymptotic worst-case ratio, we build a sequence of arbitrary length. Note that the online cost of such a sequence of  $N$  requests is at least  $N$ .

As mentioned above, another similarity to [13] is a comparison of the online algorithm to several offline algorithms. We keep a fixed number of offline algorithms that all process the complete sequence along with the online algorithm, and consider their average cost. This is an upper bound for the optimal offline cost. For the two-level cache, we simply keep five offline algorithms that need to serve the same sequence of requests. (We explain the reason for keeping exactly five such algorithms later.)

A sequence of requests that is constructed in this way can be transformed into a sequence of costs that the online algorithm pays in every step. We define a cost sequence as a sequence of 1's and  $w$ 's, where 1 indicates a page is evicted from the cheap cache whereas  $w$  indicates it is evicted from the expensive cache. Given a starting configuration of the online algorithm (without loss of generality the cheap cache contains page 0 and the expensive cache contains page 1) and a cost sequence, it is easy to recall the request sequence.

**Good patterns** A good pattern is a subsequence where the initial and final states of the online are the same, and the online pays at least  $f(w) = \frac{3}{4}(1+w)$  times not only the average cost of OPT, but at least  $f(w)$  times the cost of any optimal algorithm that has the same initial state and final state (any state out of the possible six). This means that no matter in what state an offline algorithm is, even if it is identical to the online state, it pays much less and does not change its state after processing the good pattern.

For the two-level cache case we find the following good patterns.

1ww	111w111w111w	1w111w1w111w1w111w
1w1w1w	11w1w111w1w11w	1w1111w1w11w11w11w
11w1w11w1w	1w111w1w11w11w	11w1w1111w1w11w11w
1w11w1w11w	1w1111w111w1111w	1w1w111w11w111w11w
		1w1w11w111w111w11w

Among these good patterns,  $1ww$  and  $1w1w1w$  were found by hand. Since it was quite useful for our proof, we found additional good patterns using a computer program. It will soon become clear why these patterns are so useful and why we have these constraints on them.

**Partitioning the sequence** We use two steps in order to partition the sequence. Those steps are done until the part that is left contains at most  $s - 1$  requests, where  $s$  is the longest length of a pattern in the code that we will construct. (This last part is accounted for by the additive constant for the competitive ratio, and will be ignored from now on.) The first step is removing the good patterns which are defined above. We do this step recursively, until there are no good patterns left. The second part is to partition the sequence into members of a code  $S$  where each member (which is a cost sequence of the online algorithm) has the following properties:

- The ratio of the online cost to the average offline cost of the five offline algorithms is at least a prespecified goal ratio  $\mathcal{R}$
- Each of the six algorithms (that is, the five offline algorithms and the online algorithm) has a different configuration both at the start and at the end of this sequence.

We call the points in the cost sequence where a member of  $S$  ends ‘breakpoints’. The starting point of the complete sequence is also considered a breakpoint. The second property means that at every breakpoint, exactly one algorithm has each possible configuration. This is the reason for having exactly five offline algorithms. Note that we do not require a certain order among the configurations of the offline algorithms.

It is possible to use a different approach, and not have all possible configurations but only some of them. Then the number of algorithms to examine can be reduced. However, we have used the most natural and simple approach and considered all configurations.

Since we assume that certain subsequences do not occur (since good patterns are removed first), the code will contain several sequences which do not need to be examined as they may not occur. Regarding the good patterns, note that a good pattern does not necessarily occur at a breakpoint in the sequence but can occur anywhere. Thus the offline algorithms need not be in distinct configuration, and moreover they can have the same configuration as the online algorithm. This is why a requirement of a good pattern is that *any* offline algorithm can serve this pattern at cost at most a factor of  $f(w)$  less than the online algorithm, while ending up in the *same* state as it started in. This ensures that we can remove a good pattern from a sequence, since the states before and after the good pattern are exactly the same no matter what states the several algorithms were in.

**Calculations** For each of the patterns  $p \in S$ , we compute the cost of the online algorithm on  $p$  and compute the cost of the five offline algorithms to serve all requests and end up in a valid configuration (i.e. so that all six final configurations are again all different). Note that this gives room to the offline algorithms to choose which configuration each of them ends up in, and there are  $5! = 120$  possibilities.

As we need to show a lower bound of  $\mathcal{R}$ , we compare the online cost on  $p$ , denoted by  $\text{ONL}(p)$ , to the total cost of the five offline algorithms  $\text{OFF}(p)$ , and show  $5 \cdot \text{ONL}(p) \geq \mathcal{R} \cdot \text{OFF}(p)$ . This implies that for the total offline cost we have

$$\text{OFF}(\sigma) \leq \frac{5}{\mathcal{R}} \cdot \text{ONL}(\sigma) + 5(1 + w) + 5(s - 1)w.$$

The value  $5(1 + w)$  is the setup cost of the five offline algorithms to reach the correct starting configurations, and  $5(s - 1)w$  is an upper bound on the cost of serving the ‘remainder’ of the request sequence which is not a pattern in  $S$ . As  $\text{OPT}$  is an optimal offline algorithm, its cost is at most the average cost of the five offline algorithms and so  $\text{OPT}(\sigma) \leq 1/\mathcal{R} \cdot \text{ONL}(\sigma) + 1 + sw$ . We get that

$$\text{ONL}(\sigma) \geq \mathcal{R} \cdot (\text{OPT}(\sigma) - 1 - sw).$$

It is left to find the set  $S$  that implies the lower bound.

**Determining the set  $S$**  For this proof we require a large amount of patterns, some of which are very long. We use one pattern set for the interval  $[2, 2.97]$ , which contains 44,837,506 patterns of length at most 139. To calculate online and offline costs for these

Input: weight  $w$

Target competitive ratio:  $\mathcal{R} = \frac{3}{4}(w + 1)$ .

Start with a path which contains only a 1 (the first request is served using the cheap cache). We use a depth first search in a binary tree. For each node we encounter, we do the following. Denote the path from the root of the tree until the current node by  $p$ .

- If the last elements of  $p$  form a good pattern, exclude it
- For each starting state  $j$  except the online starting state, and each final state  $m$ ,
  - For each penultimate state  $\ell$ , calculate the cost of serving  $p$  while going from  $j$  to  $m$  via  $\ell$ , using the stored costs of going from  $j$  to  $\ell$  while serving the first  $i - 1$  requests of  $p$
  - Store the minimum cost of serving  $p$  while going from  $j$  to  $m$
- Find the optimal matching of starting states and final states, excluding the online starting state and the online final state (a minimum-weight assignment). This gives the cost for the five offline algorithms.
- Compare this cost to the online cost: if the ratio is at least  $\mathcal{R}$ , mark this path as done (return).

If we are not done with this path, append a 1 and call the procedure above (resursively); then append  $w$  instead and call this procedure.

Finally, repeat the whole thing starting with a  $w$  as the first element of the path.

Figure 1: Algorithm

patterns, we have used a depth first search in a binary tree. In this tree, each node is a request, and the two ways to serve this request (cheap or expensive cache) lead to two subtrees.

For each node in the tree, our algorithm checks whether the path from the root to this node gives the desired result, i.e. a competitive ratio of at least  $\frac{3}{4}(w + 1)$ . If not, it looks deeper in the tree (using a depth first search). Our algorithm is described in Figure 1. For the innermost steps, note that the cost to get from  $\ell$  to  $m$  while serving the last request in the sequence can be calculated in constant time.

Using this method, it is possible to show that for  $w = 2.97$ , FIFO is optimal. For this value of  $w$ , we need to check in total a set of about 45,000,000 patterns, all of length at most 139. Some of these actually end in a good pattern and do not require calculation of the optimal costs. For each of the other patterns we have verified that they give the correct ratio for  $1 \leq w \leq 2.97$ , as follows. If the five offline algorithms do not pay  $w$  at all



(they only use their cheap caches), we compare two linear functions in  $w$  and find either a lower bound or an upper bound for the values of  $w$  that give the correct ratio. Otherwise, we solve a quadratic equation to find the interval where the pattern gives the correct ratio. In all cases, this interval is found to cover  $[1, 2.97]$ . This is all done automatically in the computer program, which can be found at <http://i10www.ira.uka.de/vanstee/program/> together with an example output for the value 2.9.

### 3 Companion paging systems

We consider companion paging without rearranging for the case  $\kappa = 1$ , as in [4]. We concentrate on cases where the companion cache has size  $n$  and the number of types is one more, i.e.  $n + 1$ . In these cases we can construct a sequence consisting of pages  $0, \dots, 2n + 1$ , where pages  $2i - 2, 2i - 1$  are of type  $i$  ( $1 \leq i \leq n + 1$ ).

#### 3.1 $n = 1$ , no bypassing

We start with the case  $n = 1$ . We give an alternative proof to the one from [4] that the lower bound on the competitive ratio for companion paging with two types, one place for each, and companion cache of size one, is 3 (with no bypassing). The advantage of our proof in contrast to the known proof, is that it is easily verifiable, unlike the original proof which required building a finite automaton with 13,000 states.

As before we use a sequence where each request is a fault. We can convert the request sequence into a sequence of 1's and  $c$ 's, as a function of the behavior of the online algorithm: 1 means a page is evicted from the main cache, and  $c$  means it is evicted from the companion cache. There are eight possible configurations of the cache, therefore we compare our algorithm to seven offline algorithms. Without loss of generality, we let the online algorithm always start at some fixed initial state.

We use two steps in order to partition the sequence. Those steps are done until the sequence contains at most 10 requests.

**Step 1** Removal of good patterns. Similar to Section 2, a good pattern is a subsequence where the initial and final states of the online algorithm are the same, and the online algorithm pays at least three times the cost of any optimal algorithm that has the same initial state and final state (any state out the possible 8). The good patterns are:  $1c11c1$ ,  $11c11c$ , and  $c11c11$ . The removal is done recursively until the sequence contains no good patterns. It is enough to show the lower bound for the resulting sequence.

**Step 2** Using a list of 36 patterns (omitted), remove subsequences from the sequence.

Another method we used for the same purpose is the following. Instead of keeping track of all offline algorithms with all initial states except for the one of the online algorithm, we also remove the initial state that is *similar* to the online algorithm. This is the state where the cache contains exactly the same pages, but in different locations. There is exactly one such similar state in the present case. Therefore we keep track of only six offline algorithms. Clearly if the online algorithm terminates at a different state than it started, the offline algorithms have to reach the six final states that have different pages in the cache than the final state of the online algorithm. This does not change the good patterns as they are defined for all states. We get a significantly shorter set of patterns. The patterns  $c1c11c1$  and  $c1c1c11c1$  are excluded as they have  $1c11c1$  as a subsequence. Using this method, we reduced the number of patterns required for the proof from 36 to 13, see Table 1.

Pattern $p$	ONL( $p$ )	OFF( $p$ )	Pattern $p$	ONL( $p$ )	OFF( $p$ )
1	1	2	$c1c111c$	7	14
$cc$	2	4	$c1c11cc$	7	12
$c111$	4	8	$c1c1c1c$	7	14
$c11c$	4	8	$c1c1c1111$	9	16
$c1cc$	4	8	$c1c1c111c$	9	18
$c1c1cc$	6	10	$c1c1c11cc$	9	16
$c1c1111$	7	12			

Table 1: Final set of patterns for  $n = 1$ , no bypassing

As an example we consider the pattern 1. Assuming the starting state of the online algorithm is 021 (pages 0 and 2 in the main cache, page 1 in the companion cache), it loads page 3 and ends up in state 031. The request sequence is  $\{3\}$ . There are no offline algorithms in the states 021 and 120 at the start. The offline algorithms that have page 3 in cache and are not in a state similar to 031 (states 123, 132, 023, 032) pay 0. The offline algorithms in the similar states load the page 2 instead of 3 (after serving the request for 3, of course). Thus one moves from 031 to 021 and one moves from 130 to 120. They both pay 1. Now all offline algorithms are again not in similar states as the online algorithm. The total offline cost is 2 for six offline algorithms, so the average offline cost is only  $1/3$ . Since the online cost is 1, this proves the desired ratio.

### 3.2 $n = 2$ , no bypassing

To prove a lower bound of 4.35 for  $n = 2$  we need to consider 23 offline algorithms (there are 24 configurations). We use 28,229 patterns of length at most 40. Here there are three possible choices for each page eviction (main cache, and the two slots of the companion cache). Note that this lower bound is not tight. The patterns can be found at <http://i10www.ira.uka.de/vanstee/program/> together with the computer program that was used to find them.

### 3.3 $n = 1$ , bypassing

It is possible to adapt our program in order to prove a lower bound of 4 for  $n = 1$  and bypassing. The online algorithm again has three types of costs on pages not present in the cache, these are evictions from the main cache, evictions from the companion cache and bypassing. The calculation of the offline cost also has to allow bypassing, this needs to be taken into account in the dynamic programming. The amount of patterns in the proof is not large, yet the proof of [4] is easier in this case. We include our proof for completeness. See Table 2 for the patterns. The symbol  $-$  in the sequence means that the online algorithm did not load the page but read it by bypassing. The symbol  $*$  means that all three cases 1,  $c$  and  $-$  are combined into one line in the table. In this occasion, the offline costs written in the table are the case that gives the lowest competitive ratio.

Pattern	ONL	OFF	Pattern	ONL	OFF	Pattern	ONL	OFF
$-$	1	2	$1c11$	4	7	$c1c1 - *$	6	10
$11$	2	3	$1c1-$	4	7	$1c1c11*$	7	12
$1-$	2	3	$c1cc$	4	7	$1c1c1c-$	7	12
$cc$	2	3	$c1c-$	4	7	$1c1c1 - *$	7	12
$c-$	2	3	$1c1ccc$	6	10	$1c1cc1*$	7	12
$1cc$	3	5	$1c1cc-$	6	10	$c1c1c1*$	7	12
$1c-$	3	5	$1c1c - *$	6	10	$c1c1cc*$	7	12
$c11$	3	5	$c1c11*$	6	10	$1c1c1c1*$	8	14
$c1-$	3	5	$c1c1c-$	6	10	$1c1c1cc*$	8	14

Table 2: The patterns when bypassing is allowed

### 3.4 General $n$ , no bypassing

Finally we give a very simple alternative proof of the lower bound  $n + 1$  for the case without bypassing. We do not use partitioning into patterns here. Instead consider a single type. If all inputs consist of pages of this type, we can see our cache as a simple cache with  $n + 1$  slots. Previous results [15, 12] on simple caches give the lower bound  $n + 1$ .

## 4 Conclusion

We have developed methods to calculate lower bounds for on-line paging problems using computer assisted development of proofs. We are sure that our methods can be applied to other paging and caching problems. It is interesting to find out what the lowest weight ratio  $w$  is where FIFO stops being the best on-line algorithm for the two-level cache problem ( $\kappa = 1$ ). (It is possible that FIFO is optimal on more disjoint intervals.) From [8] we know that for a sufficiently large  $w$ , FIFO is no longer optimal. We conjecture that the lowest such value for  $w$  is 3. For the companion paging problem, it would be interesting to extend our methods to show a tight lower bound of  $2n + 1$  (if this is indeed the true bound) for  $n + 1$  types,  $\kappa = 1$  and a companion cache of size  $n$  (no bypassing, no rearranging).

## References

- [1] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of the 19th ACM Symposium on the Theory of Computing*, pages 305–313. ACM, 1987.
- [2] L. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966.
- [3] Allan Borodin, Sandy Irani, Prabhakar Raghavan, Baruch Schieber. Competitive paging with locality of reference. *Journal of Computer and Systems Sciences*, 50:244–258, 1995.
- [4] M. Brehob, R. Enbody, E. Torng, and S. Wagner. On-line restricted caching. *Journal of Scheduling*, 6(2):149-166, 2003.
- [5] M. Chrobak and J. Noga. LRU is better than FIFO. *Algorithmica*, 23:180–185, 1999.

- [6] M. Chrobak and J. Noga. Competitive algorithms for multilevel caching and relaxed list update. *Journal of Algorithms*, 34(2):282–308, 2000.
- [7] M. Chrobak and J. Sgall. The weighted 2-server problem. *Theoretical Computer Science*, 324(2-3):289–312, 2004.
- [8] L. Epstein, Cs. Imreh, and R. van Stee. More on weighted servers or FIFO is better than LRU. *Theoretical Computer Science*, 306(1-3): 305–317, 2003.
- [9] A. Fiat, M. Mendel, and S. S. Seiden. Online companion caching. In *Proc. of the 10th Annual European Symposium on Algorithms (ESA '2002)*, pages 499–511, 2002.
- [10] A. Fiat and M. Ricklin. Competitive algorithms for the weighted server problem. *Theoretical Computer Science*, 130:85–99, 1994.
- [11] S. Irani. Competitive analysis of paging. In A. Fiat and G. Woeginger, editors, *Online Algorithms - The State of the Art*, chapter 3, pages 52–73. Springer, 1998.
- [12] A. Karlin, M. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, 1988.
- [13] Mark Manasse, Lyle A. McGeoch, and Daniel Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11:208–230, 1990.
- [14] M. Mendel and S. S. Seiden. Online companion caching. *Theoretical Computer Science*, 324(2-3): 183–200, 2004.
- [15] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communication of the ACM*, 28:202–208, 1985.