The Worst Page-Replacement Policy

Kunal Agrawal[1], Michael A. Bender[2], and Jeremy T. Fineman[1]

[1] MIT, Cambridge, MA 02139, USA
[2] Stony Brook University, Stony Brook, NY 11794-4400, USA

Abstract. In this paper, we consider the question: what is the worst possible page-replacement strategy? Our goal is to devise an online strategy that has the highest possible fraction of misses as compared to the worst offline strategy. We show that there is no deterministic, online page-replacement strategy that is competitive with the worst offline strategy. We give a randomized strategy based on the "most-recently-used" heuristic, and show that this is the worst possible online page-replacement strategy.

1 Introduction

Since the early days of computer science, thousands of papers have been written on how to optimize various components of the memory hierarchy. In these papers a recurrent question (at least four decades old) is the following: Which page-replacement strategies are the best possible?

The point of this paper is to address the reverse question: *Which page-replacement strategies are the worst possible?* In this paper we explore different ways to formulate this question. In some of our formulations, the worst strategy is a new algorithm that (luckily) has little chance of ever being implemented in software or silicon. In others, the worst strategy may be disturbingly familiar.

We proceed by formalizing the paging problem. We assume a two-level memory hierarchy consisting of a small fast memory, the ***cache***, and an arbitrarily large slow memory. Memory is divided into unit-size blocks or ***pages***. Exactly k pages fit in fast memory. In order for a program to access a memory location, the page containing that memory location must reside in fast memory. Thus, as a program runs, it makes ***page requests***. If a requested page is already in fast memory, then the request is satisfied at no cost. Otherwise, the page must be transferred from slow to fast memory. When the fast memory already holds k pages, one page from fast memory must be evicted to make room for the new page. (The fast memory is initially empty, but once it fills up, it stays full.) The cost of a program is measured in terms of the number of transfers.

The objective of the paging problem is to minimize the number of page transfers by optimizing which pages should be evicted on each page requests.

This research was supported in part by NSF grants CCF 0621439/0621425, CCF 0540897/05414009, CCF 0634793/0632838, CCF 0541209, and CNS 0627645, and by Google Inc.

When all page requests are known a priori (the offline problem), then the optimal strategy, proposed by Belady, is to replace the page whose next request occurs furthest in the future [2].

More recent work has focused on the online problem, in which the paging algorithm must continually decide which pages to evict without prior knowledge of future page requests. Sleator and Tarjan introduce competitive analysis [10] to analyze online strategies. Let $A(\sigma)$ represent the cost incurred by the algorithm A on the request sequence σ, and let $\mathrm{OPT}(\sigma)$ be the cost incurred by the optimal offline strategy on the same sequence. For a minimization problem, we say that an online strategy A is ***c-competitive*** if there exists a constant β such that for every input sequence σ,

$$A(\sigma) \leq c \cdot \mathrm{OPT}(\sigma) + \beta \ .$$

Sleator and Tarjan prove that there is no online strategy for page replacement that is better than k-competitive, where k is the memory size. Moreover, they show that the ***least-recently-used (LRU)*** strategy, in which the page chosen for eviction is always the one requested least recently, is k-competitive. If the online strategy operates on a memory that is twice the size of that used by the offline strategy, they show that LRU is 2-competitive. Since this seminal result, many subsequent papers have analyzed paging algorithms using competitive analysis and its variations. Irani [7] gives a good study of many of these approaches.

Page-replacement strategies are used at multiple levels of the memory hierarchy. Between main memory and disk, memory transfers are called ***page faults***, and between cache and main memory, they are ***cache misses***. There are other differences in between levels besides mere terminology. In particular, because caches must be fast, a cache memory block, called a ***cache line***, can only be stored in one or a limited number x of cache locations. The cache is then called ***direct mapped*** or ***x-way-associative***, respectively. There have been several recent algorithmic papers showing how caches with limited associativity can use hashing techniques to acquire the power of caches with unlimited associativity [5, 9].

We are now ready to describe what we mean by the worst page-replacement strategy. First of all, we are interested in "reasonable" paging strategies. When we say that the strategy is ***reasonable***, we mean that it is only allowed to evict a page from fast memory when

1. an eviction is necessary to service a new page request, i.e., when the fast memory is full, and
2. the evicted page is *replaced* by the currently requested page.

Without this reasonableness restriction a paging strategy could perform poorly by preemptively evicting all pages from fast memory. In contrast, we explore strategies that somehow try to do the right thing, but just fail miserably.

Thus, our ***pessimal-cache problem*** is as follows: identify replacement strategies that maximize the number of memory transfers, no matter how efficiently code happens to be optimized for the memory system.

As with traditional paging problems, we use competitive analysis. Since we now have a maximization problem, the definition of competitive is slightly different: An online algorithm A is ***c-competitive*** if there exists a constant β such that for every input sequence σ,

$$A(\sigma) \geq \frac{1}{c} \cdot \mathrm{OPT}(\sigma) - \beta \ .$$

An input sequence σ consists of a sequence of page requests.[3] The objective of the online algorithm A is to maximize the number of page faults on the input sequence, and OPT is the offline strategy that maximizes the total number of page faults.

Since we are turning the traditional problem on its head, terminology may now seem backwards. Optimal now means optimally bad from a traditional point of view. The adversary is trying to give us an input sequence for which we do not have many more page faults than OPT. Thus, in some sense, the adversary is our friend, who is looking out for our own good, whereas we are trying to indulge in bad behavior.

Note that the pessimal-cache problem still assigns cost in the same way, and thus counts competitiveness in terms of the number of *misses* and not the number of *hits*. We leave the problem in this form because OPT may have no hits, whereas even the best online strategies have infinitely many.

Results In this paper, we present the following results:

- We prove that there is no deterministic, competitive, online algorithm for the pessimal-cache problem (Section 2).
- We show that there is no (randomized) algorithm better than k-competitive for the pessimal-cache problem (Section 2).
- We give an algorithm for the pessimal-cache problem that is expected k-competitive, and hence optimal (Section 3). Since this strategy exhibits a $1/k$ fraction of the maximum number of page faults on every input sequence, this strategy is the worst page-replacement strategy.
- We next examine page-replacement strategies for caches with limited associativity. We prove that for the direct-mapped caches, the page-replacement is, in fact, worst possible, under the assumption that page locations are random (Section 4).

2 Lower Bounds

This section gives lower bounds on the competitiveness of the pessimal-cache problem. We show that no deterministic strategy is competitive. Then we show

[3] If a page is requested repeatedly without any other page being interleaved, all strategies have no choice, and there is no page fault on any but the first access. Thus, we consider only sequences in which the same page is repeated only when another page is accessed in between.

that no strategy can be better than expected k-competitive, where k is the fast-memory size.

Our first lemma states that there is no deterministic online strategy that is competitive with the offline strategy.

Lemma 1. *Consider any deterministic strategy A for the pessimal-cache problem with fast-memory size $k \geq 2$. For any $\varepsilon > 0$ and constant β, there exists an input sequence σ such that $A(\sigma) < \varepsilon \cdot \mathrm{OPT}(\sigma) - \beta$.*

Proof. Consider a sequence σ that begins by requesting pages $v_1, v_2, \ldots, v_{k+1}$. While the first k pages are requested, all strategies have no choice and have a fast-memory containing pages $v_1, \ldots, v_k$. At the time v_{k+1} is requested, one of the pages must be evicted from fast memory. Suppose that the deterministic strategy chooses to evict page v_i. Then for any j with $1 \leq j \leq k$ and $i \neq j$, consider the sequence $\sigma = v_1, v_2, \ldots, v_{k+1}, v_j, v_{k+1}, v_j, v_{k+1}, \ldots$ that alternates between v_{k+1} and v_j. Since the deterministic strategy has $v_1, \ldots, v_k$ in fast memory when v_{k+1} is requested, and $v_i \neq v_j$ is evicted, both v_j and v_{k+1} are in fast memory after the request. Thus, all future requests are to pages already in fast memory, and this strategy does not incur any more page faults after the first $k+1$. The offline strategy OPT, on the other hand, still incurs a page fault on every request by evicting page v_j when v_{k+1} is requested and vice versa. Extending the length of the sequence proves the lemma.

Lemma 1 also holds even if we introduce ***resource deaugmentation***, that is, the online strategy runs with a *smaller* fast memory of size $k_{\mathrm{on}} \geq 2$ and offline optimal strategy runs with a larger fast memory of size $k_{\mathrm{off}} \geq k_{\mathrm{on}}$.[4] Even so, there is still no competitive deterministic strategy. The same proof still applies with the same sequence–the proof just relies on the fact that there are two particular pages in the fast memory of the online algorithm.

We now turn our attention to randomized strategies with an ***oblivious adversary***, meaning that the adversary must choose the entire input sequence before seeing the result of any of the coin tosses used by the randomized algorithm. Note that in the presence of a nonoblivious adversary, randomization does not provide extra power for pessimal-cache problem.

The following lemma states that no randomized strategy is better than expected k-competitive when both the online and offline strategies have the same fast-memory size k. Moreover, when the offline strategy uses a fast memory of size k_{off} and the online strategy has a fast memory of size $k_{\mathrm{on}} \leq k_{\mathrm{off}}$, no online strategy is better than $k_{\mathrm{off}}/(k_{\mathrm{off}} - k_{\mathrm{on}} + 1)$.

Lemma 2. *Let k_{off} be the fast memory size of the offline strategy and k_{on} (with $1 \leq k_{\mathrm{on}} \leq k_{\mathrm{off}}$) be the fast memory size of the online strategy. Consider any (randomized) online strategy A. For any $c < k_{\mathrm{off}}/(k_{\mathrm{off}} - k_{\mathrm{on}} + 1)$ and constant β, there exists an input σ such that $E[A(\sigma)] < \frac{1}{c} \cdot \mathrm{OPT}(\sigma) - \beta$.*

[4] Note that resource deaugmentation in the pessimal-cache problem means is the analog of resource augmentation in the classical problem, in which the online algorithm has a *larger* cache than the offline algorithm.

Proof. The proof is similar to that of Lemma 1. After the $(k_{\text{off}}+1)$st page request $v_{k_{\text{off}}+1}$, the online algorithm A has k_{on} pages in fast memory. Page $v_{k_{\text{off}}+1}$ is definitely in fast memory. Of the remaining k_{off} pages requested so far, $k_{\text{on}}-1$ are in A's fast memory.

Now let v_j be a randomly selected page from $v_1, \ldots, v_{k_{\text{off}}}$. Page v_j is in A's fast memory with probability $(k_{\text{on}}-1)/k_{\text{off}}$. Now consider the sequence $v_1, \ldots, v_{k_{\text{off}}}, v_{k_{\text{off}}+1}, v_j, v_{k_{\text{off}}+1}, v_j, v_{k_{\text{off}}+1}, \ldots$. With probability $(k_{\text{on}}-1)/k_{\text{off}}$, page v_j is still in fast memory after $v_{k_{\text{off}}+1}$ is requested. In this case, no future page requests cause page faults, giving the online strategy a total of $k_{\text{off}}+1$ page faults. With probability $(k_{\text{off}}-k_{\text{on}}+1)/k_{\text{off}}$, v_j is not in memory, and the strategy may be able to attain the optimal ℓ page faults, where ℓ is the length of the sequence following the first request for $v_{k_{\text{off}}+1}$. Thus, the expected number of page faults is at most $(k_{\text{off}}+1)+\ell(k_{\text{off}}-k_{\text{on}}+1)/k_{\text{off}}$, whereas the offline strategy attains $(k+1)+\ell$. Choosing a long enough sequence proves the lemma.

3 Most-Recently Used

This section describes two k-competitive strategies for the pessimal-cache problem. The first strategy uses one step of randomization followed by the deterministic "most-recently-used" (MRU) heuristic. The second strategy uses more randomization to achieve the optimal result even when the offline and online strategies have different fast-memory sizes.

Since least-recently-used (LRU) is k-competitive and optimal for traditional paging, we explore reverse strategies for the the pessimal-cache problem. The ***most-recently-used*** (***MRU***) heuristic always evicts the page in fast memory that was used most frequently. It might be reasonable to expect MRU to be k-competitive for the pessimal-cache problem. MRU, however, is deterministic, and Lemma 1 states that no deterministic strategy can be competitive.

Instead, we consider a natural variation on MRU, which we call ***randomized MRU***. In randomized MRU, the first page evicted is chosen at random. (Recall that this first eviction happens when the $(k+1)$th distinct page is requested.) All subsequent evictions follow the MRU strategy. Randomized MRU gets around the alternating-request strategy used to prove lower bounds in Lemmas 1 and 2. The following lemma shows that MRU keeps a (slightly) random set of pages in fast memory.

Lemma 3. *Let k be the size of fast memory (for both online and offline strategies), and consider any request sequence σ. After the $(k+1)$st distinct page is requested, randomized MRU guarantees that there are k pages each having probability exactly $1-1/k$ of being in fast memory, and there is one page, the most-recently-used page, that has probability 1 of being in fast memory. All other pages are definitely not in fast memory.*

Proof. We prove the claim by induction on the requests over time.

Base case. The base case is after the $(k+1)$st distinct page is requested, which is after the first eviction. Since there are k pages in fast memory at the

time that the $(k+1)$st distinct page is requested, and one is chosen to be evicted at random, the claim holds for the base case.

Inductive step. Suppose that the claim holds up until the tth request. Assume that the next request is for page v_i. There are several cases.

Case 1. Suppose that v_i is definitely not in fast memory. Then the most-recently-used page v_j is evicted, and hence v_i is definitely in fast memory and v_j is definitely not.

Case 2. Suppose that v_i is in fast memory with probability 1. Then none of the probabilities change.

Case 3. Suppose that v_i is in fast memory with probability $1 - 1/k$ and that v_j is the most-recently-used page. Then with probability $1/k$ we have v_i not in fast memory, and hence the request for v_i evicts v_j. Otherwise, v_j stays in fast memory. Thus, the probability that v_j is in fast memory is $1 - 1/k$, and the probability that the most recently used page v_i is in fast memory is 1.

The probability of any other page (other than the ones mentioned in the appropriate case) being in fast memory is unchanged across the request.

The following theorem states that randomized MRU is k-competitive, where k is the size of fast memory.

Theorem 1. *Randomized MRU is expected k-competitive, where k is the size of fast memory.*

Proof. Consider any input sequence σ. If sequence σ contains requests to fewer than $k+1$ distinct pages, then randomized MRU has at the same number of page faults as the offline strategy OPT (Both strategies have page faults only the first time each distinct page is requested.) Consider any request after the first $(k+1)$st distinct page is requested. If the request is for the most-recently-used page, then neither OPT nor randomized MRU have a page fault, since that page must be in fast memory. Otherwise, OPT causes a page fault. By Lemma 3, randomized MRU incurs a page fault with probability at least $1/k$. Specifically, MRU incurs a fault with exactly probability $1/k$ for any of k pages and probability 1 for any of the other pages. Thus, in expectation, randomized MRU incurs at least $1/k$ page faults for each page fault incurred by OPT.

This result for randomized MRU is not quite analogous to the result of Sleator and Tarjan's [10] result for LRU. It is true that LRU is k-competitive for the traditional paging problem, and randomized MRU is k-competitive for the pessimal-cache problem. However, LRU also has good performance with resource augmentation. Specifically, if LRU has a fast memory of size k and the offline strategy has a fast memory size $(1-1/c)k$, then LRU is c-competitive. In particular, if the LRU has twice the fast memory of offline, then LRU is 2-competitive. The above result for the pessimal-cache problem does not generalize in the same way–the competitive ratio depends only on the size of randomized MRU's fast memory. If randomized MRU has a size-k fast memory and the offline strategy has a size $2k$ fast memory, then randomized MRU is still only k-competitive.

We now give a more powerful MRU algorithm, ***reservoir MRU***, that achieves a better competitive ratio for the case of resource deaugmentation. As before, let k_{off} and $k_{\text{on}} \leq k_{\text{off}}$ be the sizes of the offline and online's fast memory, respectively.

The main idea of reservoir MRU is to keep a reservoir of $k_{\text{on}} - 1$ pages, where each previously-requested page resides in the reservoir with equal probability. (This technique is based on Vitter's reservoir sampling [11].) Reservoir MRU works as follows. For the first k_{on} distinct requests, the fast memory is not full, and thus there are no evictions. Subsequently, if there is a request for a previously-requested page v_i, and the page is not in memory, then the most-recently requested page is evicted. Otherwise, when the nth new page is requested, for any $n > k_{\text{on}}$, with probability $1 - (k_{\text{on}} - 1)/(n - 1)$, the most recently requested page is evicted. Otherwise, the page to evict (other than the most-recently-used page) is chosen uniformly at random.

Reservoir MRU has an invariant that is a generalization of Lemma 3. After any request, the page that was requested most recently has probability 1 of being in fast memory. All other $n - 1$ pages have probability $(k_{\text{on}} - 1)/(n - 1)$ probability of being in fast memory.

Lemma 4. *Let k_{off} and $k_{\text{on}} \leq k_{\text{off}}$ be the fast memory sizes of the offline strategy and of reservoir MRU, respectively. Consider any page-request sequence σ to reservoir MRU. After the $n > k_{\text{on}}$th distinct page is requested, there is a single page, the most-recently-used page, that has probability 1 of being in fast memory. All other pages have probability $(k_{\text{on}} - 1)/(n - 1)$ of being in fast memory.*

Proof. The proof is by induction on the requests, and is reminiscent of the proof of Lemma 3.

Base case. After the $(k_{\text{on}} + 1)$th distinct request, the $(k_{\text{on}} + 1)$th page is definitely in fast memory, and one page randomly chosen has been evicted. Reservoir MRU evicts the most recently used page with probability $1 - (k_{\text{on}} - 1)/k_{\text{on}} = 1/k_{\text{on}}$ and all other page with the same probability. Thus, every page, except the last one has probability $1 - 1/k_{\text{on}}$ of being in fast memory, and the lemma holds for the base case.

Inductive step. Consider a request for page v_i after the nth distinct page has been requested. Assume by induction that the most-recently-used page v_j is definitely in fact memory and that all other $n - 1$ pages are in fast memory with probability $(k_{\text{on}} - 1)/(n - 1)$. There are several cases.

Case 1. Suppose that page $v_i = v_j$, i.e., v_j is in fast memory with probability 1. Then none of the probabilities change.

Case 2. Suppose that page v_i has been previously requested, but $v_i \neq v_j$. If v_i is already in fast memory then nothing is evicted. Otherwise, by the properties of reservoir MRU, page v_j is evicted. Since v_i was in fast memory with probability $(k_{\text{on}} - 1)/(n - 1)$, page v_j is evicted with probability $1 - (k_{\text{on}} - 1)/(n - 1)$ and remains in fast memory with probability $(k_{\text{on}} - 1)/(n - 1)$. None of the probabilities for pages other than v_i and v_j change.

Case 3. Suppose that page v_i has never been requested before, that is, v_i is the $(n+1)$st distinct request. By the properties of reservoir MRU, the most-recently-

used page v_j (which is definitely in fast memory) is evicted with probability $1-(k_{\text{on}}-1)/n$ and remains in fast memory with probability $(k_{\text{on}}-1)/n$. Thus, the probability that v_j is in fast memory is at the desired value.

The probability that each additional page is in shared memory now also needs to decrease since the number of distinct pages has increased by one. Since with probability $(k_{\text{on}}-1)/n$, a random page from the other $k_{\text{on}}-1$ pages is evicted from fast memory, each page in fast memory is evicted with probability $1/n$. The probability that any page is in fast memory after this process is the probability that the page was in a fast memory before the $(n+1)$st distinct page request times the probability that the page was not evicted by this request, which is $(k_{\text{on}}-1)/(n-1)(1-1/n) = (k_{\text{on}}-1)/n$. Since the number of distinct pages requested is now $n+1$, this probability also matches the lemma statement.

We now use the previous lemma to prove a better competitive ratio for reservoir MRU in the case of resource deaugmentation.

Theorem 2. *Reservoir MRU is expected $k_{\text{off}}/(k_{\text{off}}-k_{\text{on}}+1)$-competitive, where k_{off} is the size of fast memory of the offline strategy, and $k_{\text{on}} \leq k_{\text{off}}$ is the size of fast memory for reservoir MRU.*

Proof. Before k_{off} distinct requests, reservoir MRU has at least as many page faults as the offline strategy. And after this point, each time the offline strategy has a page fault, since $n > k_{\text{off}}$, reservoir MRU incurs a page fault with probability at least $1-(k_{\text{on}}-1)/k_{\text{off}}$ from Lemma 4.

This theorem means that when the offline strategy and reservoir MRU have the same fast-memory size k, reservoir MRU is k-competitive. When reservoir MRU has fast-memory size k_{on} and the offline strategy has fast-memory size $(1+1/c)k_{\text{on}}$, reservoir MRU is $(c+1)$-competitive.[5]

Reservoir MRU requires some additional state—in particular, we need one bit per page to indicate whether the page has been requested before. Consequently, if the sequence requests n distinct pages, then we need $O(n)$ extra bits of state. In contrast, Achlioptas et. al.'s [1] optimal randomized algorithm for the page-replacement problem requires only $O(k^2 \log k)$ extra bits of state. The extra state is unavoidable for reservoir MRU, however, because we must know when n, the number of distinct pages, increases. Fortunately, these extra bits can be stored in the slow memory, associated with each page—only the more reasonable $O(\log n)$ bits for the counter storing n need be remembered by the algorithm at any given time.

4 Direct Mapping

In this section we consider the page-replacement strategy used in direct-mapped caches. We show that for the pessimal-cache problem, direct mapping is k-competitive under some assumptions about the mapping strategy or about the layout in slow memory.

[5] In fact, the offline strategy can have a slightly smaller memory–with size $\lceil(1+1/c)(k_{\text{on}}-1)\rceil$–and we still attain the $(c+1)$-competitiveness.

In a direct-mapping strategy (see, e.g., [6]) each page v_i can be stored in only a single location $L(v_i)$ in fast memory. Thus, in a direct-mapped cache, once the function $L(v_i)$ is chosen, there are no algorithmic decisions to make: whenever a page v_i is requested, we must evict the page that is currently stored in location $L(v_i)$ and store v_i there instead.

In the following, we show that if $L(v_i)$ is randomly chosen for each v_i, then direct mapping is k-competitive with the optimal offline strategy (with no direct-mapped restrictions).

In fact, typically in real caches, the function $L(v_i)$ is determined by the low-order bits in the address of v_i in slow memory; it is not random. However, if each page v_i is stored in a random memory address in slow memory then our theorem still applies. While it is often unrealistic to assume that each page v_i is randomly stored, this approach was also used in [5, 9] to enable direct-mapped caches to simulate caches with no restrictions on associativity.

Observe that direct mapping is not a reasonable strategy when compared with the optimal off-line strategy with no mapping restrictions. In particular, a direct-mapped fast memory may evict a page before the rest of the fast memory is full. However, since caches with limited associativity are so common, it is of interest to explore this special case.

The following theorem states that direct mapping is competitive with the optimal offline strategy for the pessimal-cache problem.

Theorem 3. *Direct-mapping is k-competitive, where k is the fast-memory size of the both be the direct-mapping and offline strategies.*

Proof. We claim that a particular page is requested many times and the offline strategy incurs a page fault on ℓ of these requests, then direct mapping incurs at least ℓ/k page faults on v_i in expectation. We prove this claim by induction on the number of requests to v_i.

The first time that v_i is requested, there is a page fault. If v_i is requested again immediately (without any interleaving page requests), then both strategies have the page in fast memory. If v_i is requested again after another page is requested, then the offline strategy may have a page fault. The direct-mapping strategy incurs a page fault with probability at least $1/k$, because at least one page v_j is requested between v_i requests, and this page v_j has a $1/k$ probability of evicting v_i from fast memory.

5 Conclusions

For the pessimal-cache problem, randomization is necessary to achieve any competitive ratio, and the best competitive ratio without resource deaugmentation is k. In contrast, for the original problem, deterministic strategies can be k-competitive [10], and upper [1, 4, 8] and lower [4] bounds of $\Theta(\log k)$ exist for randomized strategies against oblivious adversaries.

In this paper, competitive ratios are k or larger; is there some model in which the competitive ratio is smaller? Essentially, we're trying to get a better

definition of reasonable strategies giving the adversary just the right amount of power. This concept is similar to many approaches for the original page-replacement problem—for example, the graph-theoretic approach [3] tries to better model locality. Unfortunately, the traditional approaches seem to have little impact for the pessimal-cache problem. For example, looking at access patterns matching a graph, little can be said even if the graph is just a simple line. Adding power like lookahead to the online strategy, on the other hand, trivializes the problem since the optimal offline strategy can be implemented with a lookahead of 1. It would be nice to come up with a more accurate model that allows us to beat k-competitiveness.

It's interesting that direct-mapped cache is optimally bad when the program shows no locality (i.e., as in a multiprogrammed environment). In this model, however, we cannot show anything about the badness of a 2-way (or, more generally, c-way) set-associative cache using LRU. In particular, the LRU subcomponent forces the cache to make the "right" choice for eviction, and the sequence ping-ponging between two pages is sufficient to guarantee no future misses.

One way of weakening the adversary is to restrict the definition of "reasonable" strategies by disallowing the eviction of the most (or perhaps the c most) recently used pages. In some sense, we're forcing the cache to model some small amount of locality, since, after all, that is the purpose of the cache. This modification of the problem has the nice property that it allows us to analyze the pessimal-cache problem for a c-way set-associative cache. In particular, a 2-way set-associative cache is roughly k^2-competitive for the pessimal-cache problem. This result appears to generalize for c-way set-associative caches as well.

It would nice to see if anything from this paper applies to other problems, or generalizations of the paging problem, like the k-servers on a line problem, for example.

References

1. Dimitris Achlioptas, Marek Chrobak, and John Noga. Competitive analysis of randomized paging algorithms. In *Annual European Symposium on Algorithms (ESA)*, volume 1136 of *Lecture Notes in Computer Science*, pages 419–430, Barcelona, Spain, September 25-27 1996.
2. Laszlo A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5(2):78–101, 1966.
3. Allan Borodin, Sandy Irani, Prabhakar Raghavan, and Baruch Schieber. Competitive paging with locality of reference. *Journal of Computer and System Sciences*, 50(2):244–258, April 1995.
4. Amos Fiat, Richard M. Karp, Michael Luby, Lyle A. McGeoch, Daniel D. Sleator, and Neal E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, December 1991.
5. Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297, New York, New York, October 17–19 1999.

6. John L. Hennessy and David A. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, Third edition, 2003.
7. Sandy Irani. Competitive analysis of paging. In *Developments from a June 1996 Seminar on Online Algorithms*, volume 1442 of *Lecture Notes in Computer Science*, pages 52–73, London, UK, 1998. Springer-Verlag.
8. Lyle A. McGeoch and Daniel D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6:816–825, 1991.
9. Sandeep Sen and Siddhartha Chatterjee. Towards a theory of cache-efficient algorithms. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 829–838, San Francisco, California, January 2000.
10. Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, February 1985.
11. Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.