The optimal structure of algorithms for α-paging [☆]

Annamária Kovács, Ulrich Meyer, Gabriel Moruz, Andrei Negoescu[*]

Goethe University Frankfurt, 60325 Frankfurt am Main, Germany

Abstract

Paging is an important part of data management between two memory hierarchies, a fast *cache* and a slow *disk*. Its main application areas are modern operating systems and databases. Paging algorithms need to take decisions without precisely knowing the future behavior of the system, therefore paging is one of the most studied problems in the field of online-algorithms. In this paper we consider α-paging [13], a variation of the classical paging problem. It models the asymmetry between reading and writing data when the slow disk is implemented by means of flash memory. We develop an online structure that keeps track of the cache contents of the optimal offline algorithm. Based on this structure we design the algorithm class OPTMark which has the best possible competitive ratio and performs well on real-world traces.

Keywords: paging, online-algorithms, empirical competitive ratio

1. Introduction

Flash memory combines the advantages of semiconductor-based memory and non-volatile data storage devices: it allows fast random read access and no power supply is needed for archiving data. Compared to mechanical hard-disks flash memory devices are lighter, more shock-resistant and consume less power. Due to their decreasing price they have become an economically competitive alternative in many areas (in particular for mobile computing). Like most other storage technologies, flash memory works block-based. However, modifying a block on flash memory typically requires rewriting a number of neighboring blocks as well. Therefore, best writing performance is achieved by sequentially writing several neighboring blocks at once, whereas reading can be done efficiently by reading only a single block at once [2]. We consider this read/write asymmetry for the paging problem on two-level memory hierarchies. Paging strategies

[☆]Partially supported by the DFG grant ME 2088/3-1, and by MADALGO – Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

[*]Corresponding author
Email address: negoescu@cs.uni-frankfurt.de (Andrei Negoescu)

decide which memory pages reside in the fast and small cache (e.g. RAM) and which have to be loaded upon access from a slower and larger disk (e.g. a flash memory device).

Classical Paging. Provided with a *cache* of fixed size k and a *disk* of infinite size we have to serve a sequence of requests to memory pages of equal size. If the currently requested page is in the cache we are done; otherwise the requested page needs to be loaded from the disk into the cache, and we say that a *page fault* occurs. Upon a page fault the paging algorithm needs to decide which page to replace, if the cache is full. In the classical paging problem the goal is to minimize the number of page faults. There exist many variations of this simple core problem, e.g. using page weights [9], varying the cache size [14] or requesting sets of pages instead of single pages [8].

α-paging. In the α-paging model [13] it is assumed that pages evicted from the cache need to be written back to disk and that this is more efficient if done in bundles. Upon one eviction the algorithm is allowed to write out up to α arbitrary pages in order to make room for new ones. In contrast to the classical model the cost for α-paging is given by the number of *evictions*.

More precise, upon request of page p the algorithm pays cost 0 for loading p into the cache if needed. Since the cache size is bounded by k the algorithm may need to perform evictions to make room for new pages. In each step an arbitrary number of bundles $B_1, B_2, \ldots, B_l$, each containing at most α pages can be evicted at the cost of l (number of bundles). Although the model permits evictions at every page request, it was shown that it suffices to consider only *lazy* algorithms [13]. Lazy algorithms evict at most one bundle, and they do so iff the cache is full and the requested page is not in the cache.

Simple algorithms like LRU can be adapted in a straightforward way, e.g. α-LRU evicts the α least-recently requested pages instead of the least recently requested page. Note that for $\alpha = 1$ the model is equivalent to the classical paging model (up to a constant additive cost factor due to the different cost measure).

Note that α-paging captures some but not all characteristics of flash memory devices in paging scenarios. An important aspect concerns the degree of freedom to choose a set of α jointly evicted pages. If those were to be written back to their original positions on flash memory, improved writing behavior would only occur if these positions happen to be adjacent on the device. In practice, however, so-called *flash translation layers* like the ones used in EasyCo [7] or ExtremeFFS [17] efficiently overcome this issue by re-mapping the logical positions of those α flash pages so that they can be physically written in neighboring device positions whereas their old positions are internally marked to be invalid. Occasional compaction phases rearrange invalid slots so that they can be reused in consecutive fashion. Another issue not captured by this model is that reading pages from flash memory is not for free in practice.

Competitive Analysis. The most prominent way to analyze online algorithms is *competitive analysis* [11, 18], where the cost of the online algorithm is compared

to the cost of the optimal offline solution, i.e. an algorithm having full knowledge about the input sequence. A deterministic online algorithm A is c-competitive if for any input sequence it holds that

$$cost(A) \leq c \cdot cost(OPT) + b,$$

where $cost(A)$ and $cost(OPT)$ denote the cost of A and the optimal offline cost respectively, and b is a constant. For deterministic algorithms the lower bound on the competitive ratio is k [18]. Three of the most prominent k-competitive paging algorithms are LRU (Least Recently Used), FIFO (First In First Out) and FWF (Flush When Full) [18]. For relevant results on online algorithms, we refer the reader to comprehensive surveys in [3, 5]. In the α-paging model it was shown that k/α is a lower bound on the competitive ratio [13]. The lower bound is matched by the k/α-competitive algorithm α-LRU.

Optimal Offline Algorithm. For classical paging a simple optimal offline algorithm MIN is known [4], and works by evicting, upon a cache miss, the page in cache which is re-requested farthest in the future. Its adaptation α-MIN [13] evicts the α farthest distinct pages and was shown to be optimal in the α-paging model. An online characterization of the optimal offline algorithm describes precisely OPT's possible cache contents given only the sequence seen so far. The first online characterizations of MIN were provided in [15] for the design and analysis of the first strongly competitive algorithm and in [12] in order to prove that LRU is optimal in the *diffuse adversary* model. Alternative descriptions were provided in [1] and [6] for the purpose of space-efficient strongly competitive randomized algorithms.

In [16] the characterisation from [6] was used for designing deterministic online algorithms which always cache all *revealed* pages, these are pages which are for sure in MIN's cache independent of the future requests. This approach always leads to k-competitive algorithms. More important is the observation that most requests in real-word inputs are requests to revealed pages.

Contributions. We provide an online characterization of the optimal offline algorithm for α-paging and prove its correctness. It is an adaptation of the characterization from [6]. After each request it splits the page-set in three categories:

- *revealed* pages, contained in the optimal cache for all possible future requests
- pages which are not in the optimal cache for all possible future requests
- pages, where both possible scenarios for the future exist, one where they belong to the optimal cache and one where they do not.

Using this structure, we propose a new priority-based marking class, which is k/α competitive and caches preferentially revealed pages. We combine our algorithm class with the future prediction strategy RDM [16] and perform trace-driven simulations. We conclude that the resulting algorithm can outperform adaptations of FIFO and LRU.

$$L' = \begin{cases} (\ \mathbf{L_0} \setminus \{\mathbf{p}\} \mid L_1 \mid \ \dots \ \mid L_s \mid \{\mathbf{p}\}\ ) & \text{if } p \in L_0,\ s < k \quad \text{(1a)} \\ \big(\ \mathbf{L_0} \setminus \{\mathbf{p}\} \mid L_1 \mid \ \dots \ \mid L_{k-\alpha-1} \mid \mathbf{L'_{k-\alpha}} \mid \{\mathbf{p}\}\ \big) & \text{if } p \in L_0,\ s = k \quad \text{(1b)} \\ L'_{k-\alpha} := L_{k-\alpha} \cup L_{k-\alpha+1} \cup \dots \cup L_k & \\ \big(\ L_0 \mid L_1 \mid \ \dots \ \mid L_{i-2} \mid \mathbf{L'_{i-1}} \mid L_{i+1} \mid \ \dots \ \mid L_s \mid \{\mathbf{p}\}\ \big)\,, & \text{if } p \in L_i,\ i > 0 \quad \text{(1c)} \\ L'_{i-1} := L_{i-1} \cup L_i \setminus \{p\} & \end{cases}$$

Figure 1: Layer update rules for processing page p.

2. Optimal configurations

The optimal offline algorithm α-MIN, in the following denoted OPT, uses the following strategy: if the cache is full and a page fault occurs it evicts the α pages re-requested furthest in the future. Given the processed sequence σ_1 we are interested in all possible cache contents of OPT. A cache configuration C is a set containing at most k pages. We say that C is *valid* iff there exists a continuation σ_2 such that OPT has cache content C after processing σ_1 given that the overall input is $\sigma_1\sigma_2$. In the following we define a set structure L which keeps track online of all valid configurations in the α-paging model. It partitions the page-set $\mathcal{P}$ in $s+1$ consecutive sets denoted *layers*

$$L = (\ L_0 \mid L_1 \mid L_2 \mid \ \dots \ \mid L_s\ )\,.$$

The highest index s varies between $k-\alpha+1$ and k and L_s contains exactly one page. We prove later that s is the *precise number of pages in OPT's cache*, regardless of the continuation of the request sequence. Furthermore let $s-r$ be the minimal index such that $L_{s-r+1}, L_{s-r+2}, \dots, L_s$ are singletons. We call these r rightmost singletons the *revealed region* and the corresponding pages *revealed*. We prove later that revealed pages are for sure in OPT's cache for all possible sequences σ_2. The precise relation between L and the set of valid configurations is given in Lemma 1. First we deal with the initialization and update of the layer structure.

Initialization. Let $p_1, p_2, \dots, p_k$ be the first k requested pairwise distinct pages in arbitrary order. Initially we define $L = (\ L_0 \mid \{p_1\} \mid \{p_2\} \mid \ \dots \ \mid \{p_k\}\ )$ and L_0 contains all other pages.

Layer Update Rules. Layers are updated after each request. Let L be the layer structure before and L' after the request to page p respectively. The three rules are listed in Figure 1. Rules (1a) and (1b) cover requests from L_0. If $s < k$ the requested page is moved from L_0 into a newly created rightmost (singleton) layer and thus the maximal index s increases by 1. If $s = k$ we merge the rightmost $\alpha+1$ layers into the new layer with index $k-\alpha$ which is followed by the layer $\{p\}$ and thus s decreases by $\alpha-1$. There are exactly $\alpha-1$ requests of type (1a) between two consecutive requests of type (1b). Rule (1c) is applied

if $p \notin L_0$. We remove $p \in L_i$ and merge L_{i-1} and L_i . This causes a decrease by 1 for every layer index to the right of L_i. Layer $\{p\}$ is appended as a new rightmost layer and thus $s' = s$. The third rule is identical to the layer update rule for classical paging from [6].

The revealed region increases by one every time rule (1a) is applied. Upon rule (1c) the revealed region increases by one only if p was not a revealed page before the request. Rule (1b) sets the size of the revealed region to 1 due to Fact 1. Note that layers L_i with $i > k - \alpha$ are always revealed.

Fact 1. *For all $i > 0$ it holds $|L_i| \geq 1$.*

Proof. The invariant clearly holds for the initialization and it is preserved by all layer update rules. Cases (1a) and (1b) only merge existing layers and create a new one with one element. For case (1c) we have that $L'_{i-1} := L_{i-1} \cup L_i \setminus \{p\}$ contains at least $|L_{i-1}| \geq 1$ elements. □

Selection Process. The following selection process [6] was originally used to develop online algorithms which stay in valid configurations in the classical paging model. Here we use it to prove that the proposed layer structure characterizes the set of valid configurations in the α-paging model.

In the following we assume that all pages (except the ones in L_0) have pairwise distinct priorities that will be defined in what follows. A page is assigned a new priority only upon request. Priorities shall reflect the order of future requests, namely the higher the priority of a page p, the sooner we assume p to be requested again in the future. If a page is never requested again we assign the priority ∞ and ties are broken arbitrarily. For some set S let $\min_j(S)$ and $\max_j(S)$ denote the subset of S of size j with the smallest and the largest priorities, respectively. Given L and a priority assignment, the following selection process returns a cache configuration. This cache configuration is shown to be OPT's cache content given that the priorities indeed reflect the future request order (see Theorem 2).

Definition 1 ([6], Definition 1)**.** *We construct iteratively $s+1$ selection sets $C_0, \dots, C_s$ from the layer partition L as follows: we first set $C_0 = \emptyset$ and then for $j = 1, \dots, k$ we set $C_j = \max_j(C_{j-1} \cup L_j)$. The outcome C is given by C_s.*

Fact 2. *The final selection C contains s pages; none of them is from L_0. It contains all revealed pages and the page with the highest priority from $L_1, \dots, L_s$.*

Proof. By Fact 1 we have $|L_i| \geq 1$ and the invariant $|C_i| = i$ follows inductively. This leads to $|C| = |C_s| = s$. By the definition of the selection process no page from L_0 is selected in set C_0 and thus also in no other selection set. If p has the highest priority from $L_1, \dots, L_s$ it will be the selected in $C_i = \max_i(C_{i-1} \cup L_i)$ since no pages from L_0 compete. The same argument holds for $C_{i+1}, C_{i+2}, \dots, C_s$.

□

Theorem 1 gives an alternative characterization of the final selection C, namely how this selection changes upon each request. Together with Fact 2 it is

used to prove that the layer structure characterizes the set of valid configurations (Theorem 2 and Lemma 1). This characterization enables us in Lemma 2 to give a sufficient condition for strongly competitive algorithms.

Theorem 1. *Let L, L' be the layer representations before/after requesting p, and C, C' the corresponding selections. If $p \in C$ we have $C' = C$ and otherwise C' can be obtained as follows:*

$$C^{'} = \begin{cases} C \cup \{p\} & p \in L_0,\ s < k \quad (2a) \\ C \setminus \min_\alpha(C) \cup \{p\} & p \in L_0,\ s = k \quad (2b) \\ C \setminus \min_1(M_j) \cup \{p\} & p \in L_i,\ i > 0 \quad (2c) \end{cases}$$

where $M_l := (L_1 \cup L_2 \cup \cdots \cup L_l) \cap C$, $j := \min\{l \mid l \geq i, |M_l| = l\}$.

Proof. First note that $j := \min\{l \mid l \geq i, |M_l| = l\}$ is well-defined since $C = C_s$ contains exactly s pages from $L_1 \cup L_2 \cup \cdots \cup L_s$ due to Fact 2. The layer update rule (1c) is identical to the one in [6] which implies identical changes to the final selection. This directly covers (2c) and the case $p \in C$, since $p \in C$ implies $p \in L_i \neq L_0$. If $p \in L_0$ and $s < k$ the layer update rule (1b) appends the layer $L_{s+1} = \{p\}$ which does not affect the first s selection sets ($C'_l = C_l$ for all $l \leq s$). By Definition 1 we obtain

$$C' = C'_{s+1} = \max_{s+1}\left(C'_s \cup L'_{s+1}\right) = C_s \cup \{p\} = C \cup \{p\}.$$

If $p \in L_0$ and $s = k$ the layer update rule (1a) merges the α rightmost singleton layers with $L_{k-\alpha}$. This does not affect $L_1, \ldots, L_{k-\alpha-1}$, therefore we have $C'_{k-\alpha-1} = C_{k-\alpha-1}$. Since $L_{k-\alpha+1}, \ldots, L_k$ are singletons it holds $C = C_{k-\alpha} \cup L_{k-\alpha+1} \cup \cdots \cup L_k$ and we get:

$$\begin{aligned} C' =& C'_{k-\alpha+1} = C'_{k-\alpha} \cup \{p\} \\ =& \max_{k-\alpha}\left(C'_{k-\alpha-1} \cup L'_{k-\alpha}\right) \cup \{p\} \\ =& \max_{k-\alpha}\left(C_{k-\alpha-1} \cup L_{k-\alpha} \cup \cdots \cup L_k\right) \cup \{p\} \\ =& \max_{k-\alpha}\left(\max_{k-\alpha}\left(C_{k-\alpha-1} \cup L_{k-\alpha}\right) \cup \cdots \cup L_k\right) \cup \{p\} \\ =& \max_{k-\alpha}(C) \cup \{p\} \end{aligned}$$

Since C contains k pages $\max_{k-\alpha}(C)$ is equivalent to $C \setminus \min_\alpha(C) \cup \{p\}$. □

Optimal Algorithm. Let τ be a priority assignment, which assigns each page upon its (re-)request a prority that is distinct from all other page priorities. Further we define A_τ to be a paging algorithm which always caches the final selection C from Definition 1 based on the priority assignment τ and the layer structure. Theorem 1 tells us precisely which evictions have to be performed by A_τ in order to preserve this invariant. Note that in general A_τ is not a lazy algorithm because of (2c), which evicts only one page and not α pages.

Theorem 2. *Let τ assign each page q upon request the negated timestamp of q's next request. The cache content C of A_τ is identical to C_{OPT}, the cache content of the optimal offline algorithm.*

Proof. The claim holds for the initial cache configuration, since both algorithms store the first k pairwise distinct requested pages. Let p be the currently requested page and assume that $C = C_{OPT}$ before processing p. We show that $C' = C'_{OPT}$, where C' and C'_{OPT} are the cache contents directly after processing p.

If $p \in L_i \neq L_0$ then p has the maximal priority among all pages in $L_1, \ldots, L_s$ and p is in C due to Fact 2. By the induction hypothesis p is in OPT's cache and thus $C'_{OPT} = C_{OPT}$. By Theorem 1 we get that $C' = C$ and thus $C' = C'_{OPT}$. Next we deal with the cases where $p \in L_0$ and by Fact 2 we have that $p \notin C$ and C contains s pages. By the induction hypothesis OPT caches also the same s pages. If $s < k$ both algorithms load p in the cache without an eviction. In the case $s = k$ the cache of OPT is full and it evicts the α pages which are requested furthest in the future, and so does A_τ due to Theorem 1 since these are the α pages with the lowest priority by the definition of τ. □

Now we can prove that the layer structure precisely describes all possible cache configurations of OPT after processing the subsequence σ_1. We defined that C is *valid* iff there exists a continuation σ_2 such that OPT has cache content C after processing σ_1 given that the overall input is $\sigma_1\sigma_2$.

Lemma 1. *A cache configuration S is valid iff $|S| = s$ (the current highest layer index) and for all $i = 0, 1, \ldots, s$ it holds: S contains at most i pages from $L_0 \cup L_1 \cup \cdots \cup L_i$.*

Proof. Let σ_1 be the processed sequence and L the corresponding layer representation. We denote by σ_2 a possible continuation. By Theorem 2 and Fact 2 we know that all valid configurations contain exactly s pages.

Let us first assume that there exists an index $j \leq s$ such that S contains at least $j + 1$ pages from $L_0, L_1, \ldots, L_j$. By Definition 1 the outcome C of the selection set contains at most j pages from these layers. Thus for all priority assignments it holds $S \neq C$ and Theorem 2 implies that there does not exist a σ_2 such that S is the cache content of OPT.

On the other hand if S contains at most j pages from $L_0, L_1, \ldots, L_j$ let σ_2 be a sequence where the s pages in S are requested first. Using the priorities defined in Theorem 2 we get that $C = S$ and thus S is in the cache of OPT given that $\sigma_1\sigma_2$ is the overall sequence. □

Lemma 2. *Let A be a lazy online algorithm which always caches at least $\min\{r, k - \alpha\}$ out of r revealed pages. Algorithm A is $\frac{k}{\alpha}$-competitive.*

Proof. Let p be a requested page which causes an eviction for OPT. By Lemma 1 p is a page from L_0. Consider the phase which starts with the request of p and ends right before the next eviction for OPT (which again is triggered by a request from L_0). A request from L_0 causes the union of the last two layers (which are

both non-empty due to Fact 1 and a new singleton layer is appended. Thus directly after OPT evicts, we have $r = 1$ in the layer structure. Each following request to a non-revealed page during this phase increases r by 1. Since $r \leq k$, there are at most k requests to non-revealed pages during this phase. Since A is a lazy algorithm it evicts only bulks of α pages and only if the cache is full and a page fault occurs. After an eviction for A it takes $\alpha - 1$ further page faults in order to have again k pages in the cache. Thus if A never faults on revealed pages it performs at most k/α evictions during the phase and we are done. Otherwise consider A's last eviction during this phase with $r \leq k - \alpha$. A performed at most $k/\alpha - 1$ evictions since A never faulted on revealed requests. After the next eviction the cache contains $k - \alpha + 1$ revealed requests and $\alpha - 1$ free slots. There can be at most $\alpha - 1$ pages requested (not in A's cache) until the end of the phase that do not trigger an eviction for OPT. The cost of A is at most k/α in this phase. □

3. A new algorithm

Algorithms from the OnOPT class [16] for classical paging always stay in valid configurations. One could do the same for α-paging using the update rule from Theorem 1. Unfortunately this leads to a bad empirical performance, since in the α-paging model this approach leads to non-lazy algorithms due to update rule (2c). More precisely in this case we would perform an eviction containing only one page, although we could evict α pages for the same cost.

In the following we describe a new priority-based and lazy algorithm class OPTMark, which preferentially caches revealed pages. These pages are contained in every valid configuration. Note that the algorithm does not stay in valid configurations and thus keeping track of the selection sets is not needed for the implementation. We use only the layer structure as subroutine.

Initially the first k pairwise distinct pages in the input sequence are marked in the cache and we have $s = k$ singleton Layers (Figure 1). The pseudo-code for the following requests is given in Algorithm 1. We use the fact that revealed pages become non-revealed iff the currently requested page triggers an eviction for OPT. OPTMark protects the revealed pages using a simple marking system and evicts only non-revealed pages if the number of revealed pages does not exceed $k - \alpha$. This allows applying Lemma 2 and we get that OPTMark is k/α-competitive for every priority strategy. To distinguish whether a request leads to an eviction for OPT it uses the layer partition with the update rule from Figure 1.

4. Experiments

We use the RDM priority strategy [16] for OPTMark and name the resulting algorithm α-RDM. This priority strategy is based on the current timestamp t which counts the number of requests (excluding requests to revealed pages). RDM assigns the requested page p the priority $0.8t + 0.1(t - t_0)$, where t_0 is

Algorithm 1 Pseudo-code for OPTMark

```
procedure OPTMark(Page p, Cache M)
    Assign p its priority
    if p ∈ L_0 and s = k then                                ▷ OPT eviction
        Unmark all pages in M
    end if
    if p ∉ M and |M| = k then                                ▷ Eviction
        m ← #unmarked pages
        if m ≥ α then
            Evict the α unmarked pages with lowest prio
        else
            Evict all unmarked and α − m marked pages
            with lowest prio
        end if
    end if
    Load p; Mark p; Update Layers; return
end procedure
```

the value of t upon p's last request from L_0. The first parameter represents a recency priority and the second one p's duration of stay in OPT's cache. We run experiments on the traces used in [10, 16, 14] and measure the empirical competitive ratio of α-RDM compared to the adaptations of LRU and FIFO. The empirical competitive ratio on four traces is given for $\alpha = 16$ (Figure 3) and the corresponding absolute number of evictions in Figure 4. Detailed information about these traces are provided in Figure 2.

Application	#pages	#requests	OS / Collected by	Description
go	268	106790719	Windows NT / Etch	AI program playing "Go"
compress	397	129116176	Windows NT / Etch	Compression utility
gcc	459	37524334	Linux / VMTrace	GNU C/C++ compiler
vortex	4276	543247591	Windows NT / Etch	Database program

Figure 2: Details on the cache traces on which we ran experiments – the number of pairwise distinct pages requested, the total number of requests, the operating system under which the application was run, the tool used to collected the trace, and a description of the given application. The page size was 4KB [10].

The results on all 15 traces are similar to the classical paging model. FIFO is always the worst. As long as the cache size is not larger than 50% of the page-set α-RDM has a significantly better empirical competitive ratio than α-LRU on most traces. For larger cache sizes α-LRU is the better option. One explanation could be that α-LRU is almost optimal if almost all pages fit into the cache and thus deviating from the LRU strategy leads to worse results. The empirical competitive ratio of α-RDM is usually bounded by 2 except for the `compress` trace. The `go` trace yields the best performance for α-RDM compared

to α-LRU and the gcc trace the worst respectively. Traces with a very large page-set, i.e. the vortex with 4500 pages, have an almost identical chart to the classical model since α is very small compared to the considered cache sizes. Experiments with $\alpha = 32$ and higher values reveal that the gap between the performance of the three algorithms becomes smaller since their behavior comes closer to Flush When Full, a simple algorithm which flushes the whole cache upon an eviction. Other weightings of recency and duration for the priority value can lead to better results on single traces, but the best performance on all traces was achieved by the weighting from [16]. The absolute number of evictions for the compress trace show that OPTMark suffers from Belady's anomaly (it happens that we have more evictions although the cache size increases).

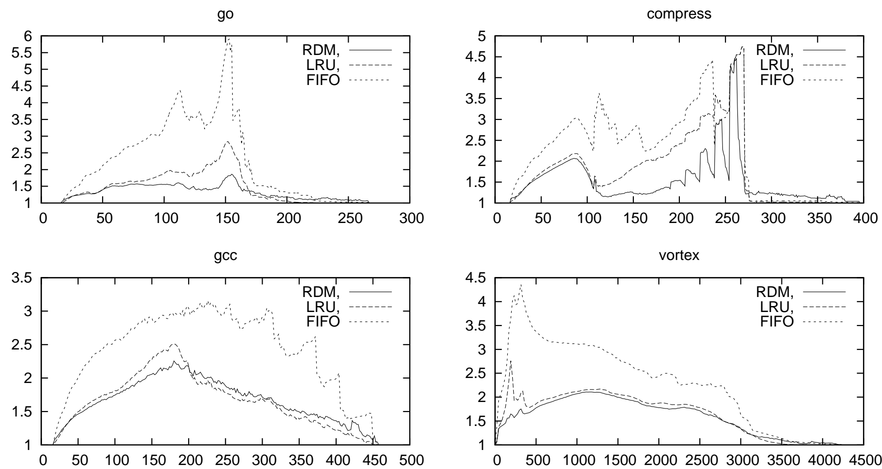

Figure 3: Empirical competitive ratio for OPTMark with RDM priorities and adaptations of LRU and FIFO for $\alpha = 16$. The x-axis denotes the cache size.

5. Conclusion

In this paper, we studied the optimal offline algorithm for α-paging. We provided an online layer structure which determines all possible cache contents of OPT in an online scenario. The layer structure was used to show that online algorithms which cache enough revealed pages have the best possible competitive ratio. Additionally we used the layer structure to design a new algorithm class OPTMark, which approximates the optimal solution on all inputs. Due to the experimental results we conclude that the performance of RDM in the classical model can be transferred to the α-paging model using the OPTMark class. It is an interesting open problem whether other online problems permit similar online approximations of OPT with respect to their empirical performance.

[1] Achlioptas, D., Chrobak, M., Noga, J., 2000. Competitive analysis of ran-

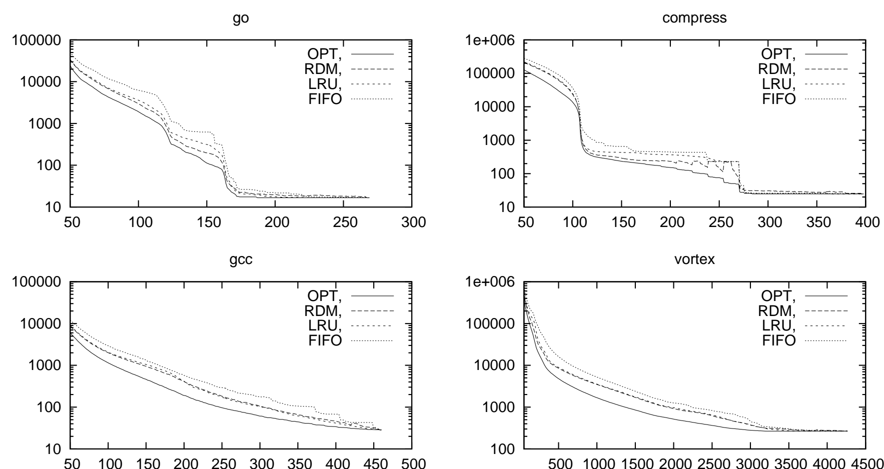

Figure 4: Absolute number of evictions for OPTMark with RDM priorities and adaptations of LRU, FIFO and OPT for $\alpha = 16$. The x-axis denotes the cache size.

domized paging algorithms. Theoretical Computer Science 234 (1-2), 203–218.

[2] Ajwani, D., Beckmann, A., Jacob, R., Meyer, U., Moruz, G., 2009. On computational models for flash memory devices. In: SEA. pp. 16–27.

[3] Albers, S., 2003. Online algorithms: a survey. Mathematical Programming 97 (1–2), 3–26.

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

[5] Borodin, A., El-Yaniv, R., 1998. Online computation and competitive anlysis. Cambridge University Press.

[6] Brodal, G., Moruz, G., Negoescu, A., 2012. Onlinemin: A fast strongly competitive randomized paging algorithm. Theory of Computing Systems, 1–19.

[7] EasyCo, Nov. 2006. Managed flash technology.
URL `http://www.easyco.com/mft/`

[8] Epstein, L., van Stee, R., Tamir, T., 2009. Paging with request sets. Theory Comput. Syst. 44 (1), 67–81.

[9] Fiat, A., Ricklin, M., 1994. Competitive algorithms for the weighted server problem. Theor. Comput. Sci. 130 (1), 85–99.

[10] Kaplan, S. F., Smaragdakis, Y., Wilson, P. R., 2003. Flexible reference trace reduction for VM simulations. ACM Transactions on Modeling and Computer Simulation 13 (1), 1–38.

[11] Karlin, A. R., Manasse, M. S., Rudolph, L., Sleator, D. D., 1988. Competitive snoopy caching. Algorithmica 3, 77–119.

[12] Koutsoupias, E., Papadimitriou, C. H., 2000. Beyond competitive analysis. SIAM Journal on Computing 30, 300–317.

[13] Kovács, A., Meyer, U., Moruz, G., Negoescu, A., 2009. Online paging for flash memory devices. In: ISAAC. pp. 352–361.

[14] López-Ortiz, A., Salinger, A., 2012. Minimizing cache usage in paging. In: WAOA. pp. 145–158.

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

[16] Moruz, G., Negoescu, A., 2012. Outperforming LRU via competitive analysis on paramtrized inputs for paging. In: Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 1669–1680.

[17] SanDisk, Nov. 2006. Sandisk introduces extremeffs, aims 100 times faster ssds.
URL `http://phys.org/news145186489.html`

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