Engineering Efficient Paging Algorithms*

Gabriel Moruz, Andrei Negoescu, Christian Neumann, and Volker Weichert

Goethe University Frankfurt am Main. Robert-Mayer-Str. 11-15, 60325 Frankfurt am Main, Germany. Email: {gabi,negoescu,cneumann,weichert}@cs.uni-frankfurt.de

Abstract. In the field of online algorithms paging is a well studied problem. LRU is a simple paging algorithm which incurs few cache misses and supports efficient implementations. Algorithms outperforming LRU in terms of cache misses exist, but are in general more complex and thus not automatically better, since their increased runtime might annihilate the gains in cache misses. In this paper we focus on efficient implementations for the OnOPT class described in [13], particularly on an algorithm in this class, denoted RDM, that was shown to typically incur fewer misses than LRU. We provide experimental evidence on a wide range of cache traces showing that our implementation of RDM is competitive to LRU with respect to runtime. In a scenario incurring realistic time penalties for cache misses, we show that our implementation consistently outperforms LRU, even if the runtime of LRU is set to zero.

1 Introduction

Paging is a prominent, well studied problem in the field of online algorithms. It also has significant practical importance, since the paging strategy is an essential efficiency issue in the field of operating systems. Formally, the problem is defined as follows. Given a cache of size k and a memory of infinite size, the algorithm must process pages *online*, i.e. make decisions based on the input sequence seen so far. If the page requested is not in the cache, a *cache miss* occurs and the page must be loaded in the cache; additionally, if the cache was full, some page must be evicted to accommodate the new one. The goal is to minimize the number of cache misses.

Traditionally, when evaluating the performance of paging algorithms, most work focuses exclusively on the number of misses incurred. However, in practice, apart from cache misses, factors such as runtime and space usage have a major impact in deciding on which algorithms to use [15, Section 3.4]. In particular, the fact that LRU (Least Recently Used) and its variants are widely popular stems not only from the fact that they incur few cache misses (typically no more than a factor of four more than the optimal cost [16]), but also because they have efficient implementations with low overhead in terms of space and runtime.

* Partially supported by the DFG grants ME 3250/1-3 and MO 2057/1-1, and by MADALGO (Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation). A full version of this paper containing all experimental results is available online at www.ae.cs.uni-frankfurt.de/sea12/sea12.pdf.

Typically, online algorithms in general and paging algorithms in particular are analyzed using *competitive analysis* [10, 14], where the online algorithm is compared against an optimal offline algorithm. An algorithm is c-competitive if the number of misses incurred is up to a factor of c away from an optimal offline solution. Any deterministic paging algorithm has a competitive ratio of at least k [14], and several k-competitive algorithms are known. Examples include LRU, FIFO, and FWF (Flush When Full); furthermore, all these algorithms can be implemented efficiently in terms of space and runtime. For randomized algorithms, in [7] a lower bound of H_k on the competitive ratio was shown[1], and a $2H_k$-competitive algorithm, denoted Mark, was proposed. Subsequently, several H_k-competitive paging algorithms were proposed, namely Partition [12], Equitable and Equitable2 [1, 2], and OnMIN [6].

Based on the layer partition in [11], we proposed in [13] a measure quantifying the "evilness" of the adversary that we denoted *attack rate*. For inputs having attack rate r, we introduced a class of r-competitive algorithms, denoted OnOPT, and we showed that these algorithms achieve a small fault rate on many practical inputs. Finally, we singled out an algorithm in this class, denoted Recency Duration Mix (in short RDM), which we showed to consistently outperform LRU and some of its variants with respect to cache misses on most inputs and cache sizes considered, at times by more than a factor of two.

Our contributions. In this work we focus on the runtime of paging algorithms that, together with the cache misses, is an important factor in practice. We propose a compressed representation of the layer partitioning in [6, 11]. Based on this and on the fact that typically most requests are to so-called *revealed pages* (pages that are for sure in the cache of an optimal algorithm), we engineer speed-up techniques for implementating the OnOPT class. If the fraction of revealed requests is $1 - O(1/k)$ these yield an RDM implementation with an amortized runtime of $O(1)$ per request with very small constant factors. We show on real-world input traces[2] that the new implementation outperforms the tree based approach in [6]. Moreover, we compare the runtime of RDM with LRU and FIFO and show that the runtimes of RDM and LRU are comparable, albeit slower than FIFO. Finally, we use a more general performance measure for paging algorithms, namely the sum of runtime and cache miss penalties. Assuming a realistic cache miss penalty of 9ms, the fact that RDM typically incurs fewer misses than both LRU and FIFO ensures that it achieves better performance for many traces and cache sizes, even if we charge LRU and FIFO a runtime of zero. This shows that OnOPT algorithms in general and RDM in particular may be of practical value.

Related work. Although competitive analysis seems too pessimistic, some of its refinements have lead to paging algorithms with low fault rates on traces extracted during the execution of real-world programs. In [8], heuristics motivated

[1] $H_k = \sum_{i=1}^{k} 1/i$ is the k^{th} harmonic number.

[2] We used all the available original reference traces from `http://www.cs.amherst.edu/~sfkaplan/research/trace-reduction/index.html`.

by the access graph model from [4] outperformed LRU. These perform an online approximation of the access graph, which models the page access pattern. Another algorithm, RLRU (Retrospective LRU), was proposed in [5], where it was proven to be better than LRU with respect to the relative worst order ratio. RLRU uses information about the optimal offline solution for its decisions. EELRU (Early Eviction LRU) [9] is an adaptive paging algorithm from a less theoretical direction, which simulates a large collection of about 256 parametrized instances of an algorithm which is a mix of LRU and MRU (Most Recently Used). All of these algorithms, including the OnOPT class, have in common that they are more complex than classical algorithms like LRU and FIFO. Because of this it is not obvious whether there exist fast implementations such that the savings in cache misses compensate for the higher runtime overhead.

1.1 Preliminaries

Layer partitioning. Given the request sequence σ seen so far, in an online scenario it is of interest to know the actual cache content C_{OPT} of the optimal offline algorithm LFD (Longest Forward Distance), which evicts, upon a cache miss, the page in cache which is re-requested farthest in the future [3]. Although in general C_{OPT} is not known since it depends also on the future request sequence τ, we are provided with partial information (from σ) about the structure of C_{OPT}, e.g. it contains for sure the most recently requested page, and pages not requested in σ are not in C_{OPT}. We say that immediately after processing σ a set C of k pages is a *valid* configuration iff there exists a future request sequence τ such that LFDs cache content equals C. A precise mathematical characterization of all possible valid configurations was given by Koutsoupias and Papadimitriou [11] and an equivalent variant of this characterization is used by the OnOPT algorithm class [13]. It consists of a partition $L = (L_0|\dots|L_k)$ of the pageset in $k+1$ disjoint sets, denoted layers. Initially, each layer in $L_1, \dots, L_k$ contains one of the first k pairwise distinct pages and L_0 contains all the remaining pages. If L is the layer partition for input σ, let L^p denote the layer partition for σp, the sequence resulting by the request of page p. The layers are updated as follows:

$$L^p = \begin{cases} (L_0 \setminus \{p\}|L_1|\dots|L_{k-2}|L_{k-1} \cup L_k|\{p\}), & \text{if } p \in L_0 \\ (L_0|\dots|L_{i-2}|L_{i-1} \cup L_i \setminus \{p\}|L_{i+1}|\dots|L_k|\{p\}), & \text{if } p \in L_i, i > 0 \end{cases}$$

In [11] it has been shown that a cache configuration C is valid iff it holds that for each $i \in \{1, \dots, k\}$ we have $|C \cap (\cup_{j=1}^{i} L_j)| \leq i$.

The *support* of L is defined as $L_1 \cup \dots \cup L_k$. Denoting by singleton a layer with one element, let r be the smallest index such that $L_r, \dots, L_k$ are singletons; the pages in $L_r \cup \dots \cup L_k$ are denoted *revealed*. We denote by *Opt-miss* pages the pages in L_0, while the remaining pages, i.e. pages in support that are not revealed, are *unrevealed* pages. A valid configuration contains all revealed pages and no page from L_0. Note that by the layer update rule all layers are non-empty.

OnOPT *algorithms.* Algorithms from the OnOPT class use the layer partition as a subroutine. The currently requested page is assigned a priority which shall reflect the rank of its next request among the other pages. Given a priority based future prediction the cache update rule ensures that their cache content is always identical to LFD's, if the prediction is correct. The pseudo-code is given in Algorithm 1. The fact that no cache misses are performed on revealed requests guarantees a reasonable performance, due to the high percentage of revealed requests in the input. In OnOPT we singled out RDM, which combines two priority policies, one based on recency and the other on the time-frame that pages spent in support. RDM achieves good results, outperforming LRU on many real-world traces and cache sizes [13].

Algorithm 1 OnOPT framework

```
procedure OnOPT(Page p, Cache M)                          ▷ Processes page p
    Assign p its priority
    if p ∉ M and p ∈ L_0 then                             ▷ Update cache
        Evict page in M with smallest priority
    else if p ∉ M and p ∈ L_i, i > 0 then
        Identify minimal j, with j ≥ i, satisfying |(L_1 ∪ ··· ∪ L_j) ∩ M| = j
        Evict page from (L_1 ∪ ··· ∪ L_j) ∩ M having smallest priority
    end if
    Update the layers                                     ▷ Layers update
end procedure
```

1.2 Revealed requests

We give experimental evidence that a very high percentage of requests are to revealed pages, which is the main motivation for the OnOPT implementations we propose in this paper. For the remainder of the paper we use a collection of cache traces extracted from various applications for our experiments; due to space limitations, details about these traces are given in the full version.

The charts in Figure 1 show that if enough pages fit in memory (usually about 10%), almost all of the requests are to revealed pages. In these cases the ratio of revealed requests in the input is about $(k-1)/k$, which we approximate by $1 - O(1/k)$. For the remainder of the paper we will focus on how to process these requests as fast as possible at the expense of increasing the worst case time for processing requests to Opt-miss and unrevealed pages.

2 Compressed Layers

We simplify the layer partition with the main purpose of reducing the runtime for layer updates. The layer partition can be seen as a sequence of conditions that a valid configuration must fulfill. Consider the initial partition, where each L_i

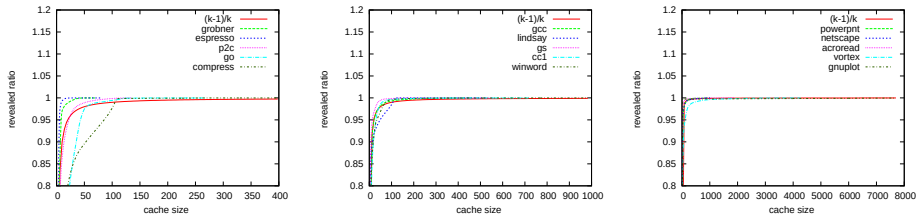

Fig. 1. The ratio of revealed requests against for all cache traces. For decently large cache sizes the ratio of revealed requests is about $(k-1)/k$ for all traces.

contains exactly one page p_i. The partition implies the constraints that a valid configuration contains at most one element from $\{p_1\}$, two elements from $\{p_1, p_2\}$ and so on. Since each layer has only one page, these k conditions can be reduced to one, namely at most k pages from $\{p_1, \ldots, p_k\}$. We generalize this example as follows. Given the original layer partition L, we define a compressed partition $\mathcal{L}$ which groups all consecutive singletons of L into the first non-singleton layer to the right. An algorithmic description of this process is given in Algorithm 2, an example for $k = 7$ is provided below.

$$L = (10, 3|2|7, 5|4|1, 11|8|9|6),\ \mathcal{L} = (10, 3|\emptyset|2, 7, 5|\emptyset|4, 1, 11|\emptyset|\emptyset|8, 9, 6)$$

Algorithm 2 Layer compression

procedure LAYER COMPRESSION(Partition $L = (L_0, \ldots, L_i)$) ▷ Compress L
  $T = \emptyset$;
  for $i = 1$ to $k - 1$ **do**
    if $|L_i| = 1$ **then** ▷ L_i is singleton
      $\mathcal{L}_i = \emptyset$; $T = T \cup L_i$;
    else ▷ L_i is not singleton
      $\mathcal{L}_i = L_i \cup T$; $T = \emptyset$;
    end if
  end for
  $\mathcal{L}_k = L_k \cup T$;
end procedure

The compressed partition $\mathcal{L}$ may contain empty sets and describes the same valid configurations as L. For $\mathcal{L}$ we provide a corresponding update rule, which has the advantage that upon revealed requests nothing changes, leading to significant runtime improvements of ONOPT algorithms. Another advantage is that on the cache traces considered the number of non-empty layers is much smaller than k, which allows for more efficient implementations.

Denoting $S_i = L_1 \cup \cdots \cup L_i$, a set of k pages is a valid configuration iff $|C \cap S_i| \leq i$ for all i. Similarly, let $\mathcal{S}_i = \mathcal{L}_1 \cup \cdots \cup \mathcal{L}_i$.

Lemma 1. *The compressed partition $\mathcal{L}$ describes the same valid configurations as L, more precisely it holds for all i: $|C \cap S_i| \leq i$ iff $|C \cap \mathcal{S}_i| \leq i$.*

Proof. Let x and y, $x < y$, be two indices such that $|L_x| > 1$, $|L_y| > 1$, and $L_{x+1}, \ldots, L_{y-1}$ are singletons. Further let L' be the partially compressed layer partition up to the iteration step $i = x$. We assume that L' and L describe the same valid configurations and show that this also holds for L'', the latter resulting from iterating up to $i = y$. For $j \leq x$ or $j \geq y$ it holds $S'_j = S''_j$ and thus $|C \cap S'_j| \leq j$ iff $|C \cap S''_j| \leq j$. It remains to prove the equivalence for $x < j < y$. Assume that C is a valid configuration in L'. This means $|C \cap S'_x| \leq x < j$ and $S''_j = S''_{j-1} = \cdots = S''_x = S'_x$ resulting in $|C \cap S''_j| < j$.

Now let C be a valid configuration in L'' implying $|C \cap S''_x| \leq x$. We have $|C \cap S'_j| = |C \cap (S'_x \cup L_{x+1} \cup \cdots \cup L_j)| \leq x + (j - x) = j$. The last inequality results from $S'_x = S''_x$ and the fact that $L_{x+1}, \ldots, L_j$ are singletons. □

Given the compressing mechanism which shows how to construct $\mathcal{L}$ from L we adapt the update rule of L for $\mathcal{L}$. Let $p_1, \ldots, p_k$ be the first k pairwise distinct pages. We initially set $\mathcal{L}_k$ to the set of these k pages, $\mathcal{L}_0$ contains all other pages and the remaining layers are empty. The update rule of $\mathcal{L}$ is given in Theorem 1.

Theorem 1. *Let $\mathcal{L}$ and $\mathcal{L}^p$ be the compressed partition of L and L^p respectively. $\mathcal{L}^p$ can be obtained directly from $\mathcal{L}$ as follows:*

$$\mathcal{L}^p = \begin{cases} (\mathcal{L}_0 \setminus \{p\} | \mathcal{L}_1 | \ldots | \mathcal{L}_{k-2} | \mathcal{L}_{k-1} \cup \mathcal{L}_k | \{p\}), & \text{if } p \in \mathcal{L}_0 \\ (\mathcal{L}_0 | \ldots | \mathcal{L}_{i-2} | \mathcal{L}_{i-1} \cup \mathcal{L}_i \setminus \{p\} | \mathcal{L}_{i+1} | \ldots | \emptyset | \mathcal{L}_k \cup \{p\}), & \text{if } p \in \mathcal{L}_i, 0 < i < k \\ (\mathcal{L}_0 | \mathcal{L}_1 | \ldots | \mathcal{L}_{k-1} | \mathcal{L}_k), & \text{if } p \in \mathcal{L}_k \end{cases}$$

Proof. Due to space limitations, the proof is available only in the full version.

3 Engineering an implementation for RDM

In this section we first engineer a novel implementation for OnOPT algorithms in general and RDM in particular, with the goal of obtaining runtimes as fast as possible. We then give experimental results which support that our improved implementation not only significantly outperforms the original approaches from [6], but also is competitive with LRU and FIFO.

3.1 Implementation

Given the overwhelming amount of requests to revealed pages in practical inputs, our implementation mainly focuses on processing these as fast as possible. We first recall that RDM is an OnOPT algorithm which assigns to each requested page the priority $0.8t + 0.1(t - t_0)$, where t is the current timestamp and t_0 is the timestamp when the page lastly entered the support. Moreover, t is not increased upon revealed requests.

Throughout this section we denote by n the input size (the number of requests), by l the number of non-empty layers (at the current request time), and by m the page-set size (the number of pairwise distinct pages in the input).

Structure. We require that for each page p in the support the following information is stored: $p.t$ – the timestamp of the last request, $p.prio$ – the priority of the page, and any additional fields that might be required for computing the priority (e.g., in the case of RDM $p.t_0$ – the timestamp when p entered the support). To do so we use direct addressing, i.e. an array of page-set size where for each page the associated information is accessed by a look-up at the corresponding element. We note that an alternative implementation using a hash table has the advantage of using space proportional to the support size, but this increases the runtime via higher constant factors.

We first note that new layers are created only upon requests to Opt-miss pages, i.e. pages in $\mathcal{L}_0$. When this happens, we assign to the newly created layer a timestamp t equal to the current timestamp. This value is not modified while the layer is in support, i.e. until it is merged with $\mathcal{L}_0$.

Fact 1 *For each layer $\mathcal{L}_i$ having timestamp t and for any pages $p \in \mathcal{L}_i$ and $q \in \mathcal{L}_j$ with $j < i$, it holds that $t \leq p.t$ and $t > q.t$.*

Proof. By construction, for each i with $0 \leq i < k$ we have that the last request for any page in $\mathcal{L}_i$ is smaller than the last request of any page in $\mathcal{L}_{i+1}$. □

We store in a *layer structure* information only about the non-empty layers in the support. We do not store the empty layers – it suffices for each non-empty one to keep the number of empty layers preceding it. For each non-empty layer $\mathcal{L}_i$, we keep the following: the timestamp t, a value v which is at all times equal to $1 + e_i$ where e_i is the number of consecutive empty layers preceding $\mathcal{L}_i$, and mem which stores the amount of pages in $\mathcal{L}_i$ that are in the cache, see e.g. Figure 2. We store these layers in an array $(l_1, \ldots, l_l)$ where l_i corresponds to the ith non-empty layer. By Fact 1, we have that $l_1.t < l_2.t < \cdots < l_l.t$. Therefore, identifying the layer that a certain page belongs to can be done using binary search with its last request as key. Also, layers can be inserted and deleted in $O(l)$ time. Finally, it supports a *find-layer-j* operation, which, given a layer index i, returns some layer l_j, with $j \geq i$ such that $|M \cap \mathcal{S}_j| = j$. This layer is identified as the first l_j with $j > i$, satisfying $\sum_{i=1}^{j} l_i.v = \sum_{i=1}^{j} l_i.mem$.

We note that, asymptotically, a search tree augmented with fields for prefix sum computations is much more efficient than an array. Nonetheless, we chose the array structure because of the particular characteristics of the layers: insertions are actually appends and take $O(1)$ time, few non-empty layers, and the constants involved are small.

Finally, we store the pages contained in the cache in an (unsorted) array of size k, where page replacements are done by overwriting.

Implementing OnOPT. We implement OnOPT algorithms using the structures described above.

If a page is revealed, no replacement is done because it is in cache. Moreover, no layer changes are required. A page is revealed iff its last request is greater than or equal to $l_l.t$. Therefore, processing a revealed page takes $O(1)$ time.

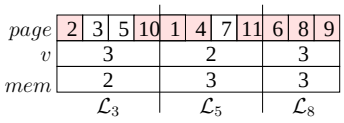

Fig. 2. Example for $\mathcal{L} = (\emptyset|\emptyset|2,3,5,10|\emptyset|1,4,7,11|\emptyset|\emptyset|6,8,9)$, emphasized pages are in cache $M = (2,10,1,4,11,6,8,9)$. Pages are not stored in the layer structure.

If the requested page is an Opt-miss page, it is not in the cache and we first evict the page having the smallest priority. We identify the victim page by scanning the cache array for the minimum priority. Finally, we replace the selected page with the requested one. To update the layers, we first merge $\mathcal{L}_{k-1}$ and $\mathcal{L}_k$ as follows: if $l_l.v > 1$ then set $l_l.v = l_l.v-1$ as $\mathcal{L}_{k-1}$ was empty; otherwise, i.e. $l_l.v = 1$, delete this layer. Afterwards, we simply append a new layer l_l with $l_l.t$ set to the current timestamp, $l_l.v = 1$, and $l_l.mem = 1$: there are no empty sets before the last layer and the new $\mathcal{L}_k$ has one element which is in memory. Altogether, processing an Opt-miss page takes $O(k)$ time.

It remains to deal with requests to unrevealed pages. If a cache miss occurs we first identify a page to evict as follows. We look the page's layer l_i up in the layer structure. Using the operation *find-layer-j*, we identify the layer l_j, and then by scanning the cache array find and evict the page with the smallest priority among the pages having last request strictly less than $l_{j+1}.t$. This ensures that the selected page is in the first j layers. To update the layers, we set $l_i.v = l_i.v-1$ if $l_i.v > 1$ and delete l_i otherwise. This not only sets $\mathcal{L}_{i-1} = \mathcal{L}_{i-1} \cup \mathcal{L}_i$, but also ensures the necessary left shifts of the layers to the right. Finally, we set $l_l.v = l_l.v + 1$ to reflect a new empty layer before $\mathcal{L}_k$. After updating the last request for the requested page, it becomes revealed since this value is greater than $l_l.t$. Thus, processing an unrevealed page takes $O(l)$ time for a cache hit and $O(k)$ time for a cache miss.

Theorem 2. *Assuming m pairwise distinct pages are requested, a cache of size k, and l non-empty layers, our implementation uses $O(m)$ space and processes a revealed page in $O(1)$ time and an Opt-miss page in $O(k)$ time. Unrevealed pages take $O(l)$ time for cache hits and $O(k)$ time for cache misses.*

Corollary 1. *Assuming that a ratio of $1-O(1/k)$ requests are to revealed pages, our implementation processes a request in $O(1)$ amortized time.*

3.2 Experimental results

In this section we conduct experiments which demonstrate empirically that our implementation for OnOPT algorithms is competitive with both LRU and FIFO, which leads us to believe that algorithms in this class, and RDM in particular, are of practical interest.

Experimental setup. For OnOPT algorithms, apart from the engineered version previously introduced we implemented the two versions described in [6]. The first one uses linked lists and processes a page in $O(|S|)$ time and the second uses a binary search tree which takes $O(\log |S|)$ time per page, where $|S|$ is the support size. Furthermore, for each of these implementations we also developed versions using the compressed layer partition. We also consider two implementations for LRU and one for FIFO. Similarly to the OnOPT implementations, we assume that for each page we associate $O(1)$ information which can be accessed in $O(1)$ time. This is done by direct addressing, i.e. we store an m-sized array where the ith entry stores data about page i. For LRU, the first implementation, denoted LRUlist, uses a linked list storing the pages in cache sorted by their last request. Keeping for each cached page a pointer to the corresponding list element, a page request takes $O(1)$ time. The second implementation, LRUlinear, uses an array of size k to store the cache contents. On a cache miss, the array is scanned to identify the page to evict. The first implementation treats a cache miss much faster than the second one but pays more time per cache hit to update the recency list. For FIFO, a circular array stores the FIFO queue.

All the experiments were conducted on all cache traces on a regular Linux computer having an Intel i7 hex-core CPU at 3.20 GHz, 10 GB of RAM, kernel version 3.1, and the sources were compiled using gcc version 4.5.3 with optimization -O3 enabled. For each data set and each cache size the runtimes were obtained as the median of five runs. Due to space limitations we show experimental results for only two cache traces, namely `go` and `winword`; the results for all traces are available in the full version.

Amount of non-empty layers. We first compare the amount of non-empty layers that we use in our implementation against the k layers used in the non-compressed one. In Figure 3 it is shown that typically both the maximum and the average number of layers are much smaller than k. As an extreme example, the `gnuplot` trace has a page-set of nearly 8000 pages, yet the maximum number of layers never exceeds 7, and the average is mostly between 2 and 3. This greatly reduces the runtime for updating the layers.

OnOPT implementations. We compare the five OnOPT implementations using the RDM priority assignment, namely the one previously described and the two implementations in [6], each of them using the compressed and uncompressed layer partition. Surprisingly, both implementations using the binary trees were hopelessly slow, mainly because they require for each request, revealed or not, to update the path in the binary tree from the requested page to the root. Instead we use an approximation of RDM for the tree implementation using the compressed partition, where for revealed requests priorities do not change and this update is not necessary; the results shown are for this approximation. The runtime results for the two selected traces are given in Figure 4. As expected, our new implementation outperforms the previous ones, for small cache sizes by significant factors. Also, the implementations using the compressed layer partition significantly outperform their non-compressed counterparts.

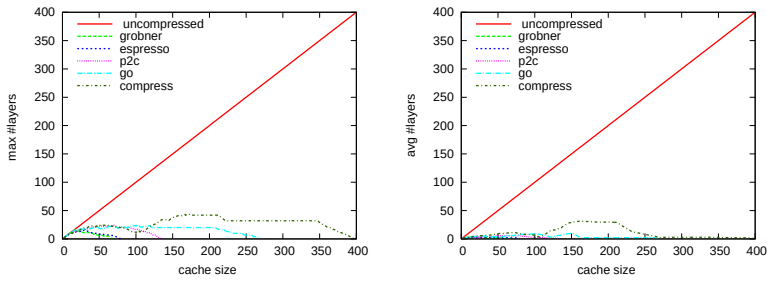

Fig. 3. Maximum (left) and average (right) number of non-empty layers in the compressed layer partition against the number of layers k in the uncompressed partition.

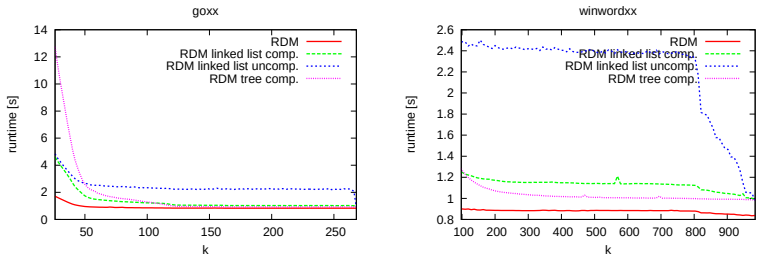

Fig. 4. The runtime for the various implementations of RDM on selected datasets.

OnOPT vs. LRU and FIFO. Having established that our new implementation is the fastest for OnOPT algorithms in general and RDM in particular, we compare it against FIFO and the two LRU implementations. The results in Figure 5 show that typically FIFO is the fastest algorithm while the LRUList is the slowest. While FIFO being the fastest is expected due to its processing pages in $O(1)$ time with very small constants, the fact that LRULinear outperforms LRUList despite its worst case of $O(k)$ time per page is explained by the overwhelming amount of cache hits (over the observed ratio of $1 - O(1/k)$ of revealed requests). For these requests, LRULinear only updates the last request for the requested page, whereas LRUList moves elements in the recency list which triggers higher constants in the runtime. Finally, we note that RDM typically is slower than LRULinear by small margins, which can be explained by the fact that both algorithms process revealed requests very fast and cache misses by scanning the memory; RDM has a slight overhead in runtime to update the layers and assign priorities. An interesting behavior is that for large cache sizes RDM is slightly faster than LRULinear, which we explain by a machine-specific optimization which does not write a value in a memory cell if the cell already stores the given value. Essentially, LRULinear always updates the last request for the current page, while RDM does not increase the timer for

revealed requests meaning that no data associated with pages changes if many consecutive revealed requests occur. This typically happens for large cache sizes.

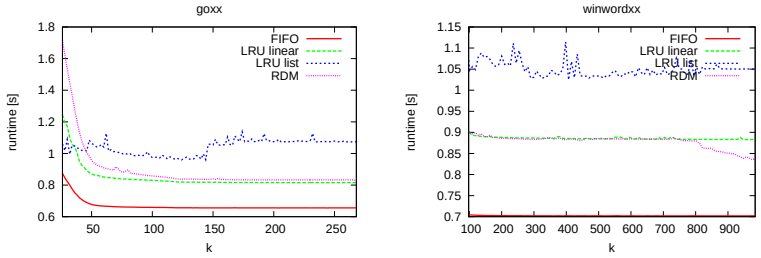

Fig. 5. The runtime for LRU, FIFO, and our RDM implementation.

Misses with time penalty. We simulate a scenario where each cache miss inflicts a time penalty. We choose a typical cost of 9ms [15, Chapter 1.3.3] per cache miss. Again, we compare RDM against LRU and FIFO, however the runtime of the algorithm will be given by its actual runtime plus the penalty of 9ms for each miss, i.e. $total = runtime + \#misses \cdot 9ms$. Moreover, for both LRU and FIFO we set the runtime to zero, so they only pay the penalty for cache misses. In this scenario the total cost is dominated by this penalty. The results in Figure 6 show that despite the zero runtime for LRU and FIFO, RDM still outperforms them on many cache sizes; in general, these results hold for the other thirteen cache traces as well. This is because the runtime component for RDM is about one second, which corresponds to about 100 misses. Given that typically RDM outperforms LRU and FIFO by more than 100 misses, it becomes the best algorithm for most datasets if the cache size is not too large; for large cache sizes algorithms incur significantly fewer misses and the runtime component becomes more important.

References

1. D. Achlioptas, M. Chrobak, and J. Noga. Competitive analysis of randomized paging algorithms. *Theoretical Computer Science*, 234(1-2):203–218, 2000.
2. W. W. Bein, L. L. Larmore, J. Noga, and R. Reischuk. Knowledge state algorithms. *Algorithmica*, 60(3):653–678, 2011.
3. L. A. Belady. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.
4. A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive paging with locality of reference. *Journal of Computer and System Sciences*, 50(2):244–258, 1995.

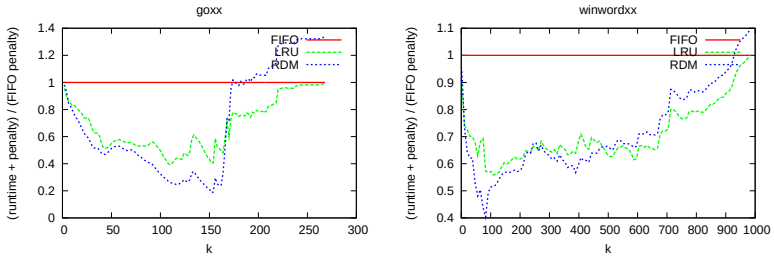

Fig. 6. RDM and LRU compared to FIFO when a cache miss costs 9ms.

5. J. Boyar, L. M. Favrholdt, and K. S. Larsen. The relative worst-order ratio applied to paging. *Journal of Computer and System Sciences*, 73(5):818–843, 2007.
6. G. S. Brodal, G. Moruz, and A. Negoescu. OnlineMIN: A fast strongly competitive randomized paging algorithm. In *Proc. 9th Workshop on Approximation and Online Algorithms*, 2011. To appear.
7. A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.
8. A. Fiat and Z. Rosen. Experimental studies of access graph based heuristics: Beating the LRU standard? In *Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 63–72, 1997.
9. S. F. Kaplan, Y. Smaragdakis, and P. R. Wilson. Flexible reference trace reduction for VM simulations. *ACM Transactions on Modeling and Computer Simulation*, 13(1):1–38, 2003.
10. A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3:77–119, 1988.
11. E. Koutsoupias and C. H. Papadimitriou. Beyond competitive analysis. *SIAM Journal on Computing*, 30:300–317, 2000.
12. L. A. McGeoch and D. D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6(6):816–825, 1991.
13. G. Moruz and A. Negoescu. Outperforming LRU via competitive analysis on paramtrized inputs for paging. In *Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1669–1680, 2012.
14. D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.
15. A. S. Tanenbaum. *Modern operating systems (3. ed.)*. Pearson Education, 2008.
16. N. E. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, 1994.