Outperforming LRU via Competitive Analysis on Parametrized Inputs for Paging

Gabriel Moruz∗,† Andrei Negoescu∗

Abstract

Competitive analysis was often criticized because of its too pessimistic guarantees which do not reflect the behavior of paging algorithms in practice. For instance, many deterministic paging algorithms achieve the optimal competitive ratio of $k$, yet LRU and its variants clearly outperform the rest in practice. In this paper we aim to reuse and refine insights from the competitive analysis to obtain new algorithms that cause few cache misses in practice. We propose a new measure of the “evilness” of the adversary, which results in a parametrization of the input that we denote attack rate. This measure is based on the characterization in [22] of the optimal offline algorithm and uses the fact that a number of pages are for sure in its memory. We show that the attack rate $r$ is a tight bound on the competitive ratio of deterministic paging algorithms and give experimental results which show that $r$ is usually much smaller than the cache size $k$ and thus provides more realistic upper bounds for the competitive ratio of existing algorithms. Furthermore, we show that our input parametrization compares favorably concerning the fault rate with approaches based on locality of reference by Albers et al. [2] and Dorrigiv et al. [14]. We use a priority-based framework, which always yields $r$-competitive algorithms regardless of the priority assignment. In this framework, LRU can be obtained under a certain priority assignment and is thus only one algorithm among many other $r$-competitive ones. Using the enhanced flexibility given by this framework, we give a priority policy which leads to an algorithm outperforming LRU, RLRU and other practical algorithms on a wide selection of real-world cache traces.

∗Institut für Informatik, Goethe-Universität Frankfurt am Main, Robert-Mayer-Str. 11-15, 60325 Frankfurt am Main, Germany. Email: {gabi,negoescu}@cs.uni-frankfurt.de.
†Partially supported by the DFG grant ME 3250/1-2, and by MADALGO (Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation).
1 Introduction

Paging has a strong practical motivation and is one of the most studied problems in the field of online algorithms. We are provided with a cache of size $k$ and a memory of infinite size, and must process page requests online, i.e., without any knowledge about future requests. If the requested page is in cache, a cache hit occurs and the algorithm proceeds at no cost. Otherwise, a cache miss occurs and the algorithm must load the page in the cache. If the cache was full, one page must be evicted to accommodate the one requested. The cost is given by the number of cache misses.

Traditionally, the quality of online algorithms in general and paging in particular is measured by comparing their cost against the cost of an optimal offline algorithm, i.e., an algorithm that is provided with the input beforehand and processes it optimally. This measure, denoted competitive ratio \[20, 24\], states that some online algorithm $A$ is $c$-competitive if for any input sequence it holds that $\text{cost}(A) \leq c \cdot \text{cost}(OPT) + b$, where $\text{cost}(A)$ and $\text{cost}(OPT)$ denote the cost of $A$ and the optimal cost respectively, and $b$ is a constant. For deterministic paging algorithms, a lower bound of $k$ on the competitive ratio was shown in \[24\]. Several algorithms, such as LRU (Least Recently Used), FIFO (First In First Out), and FWF (Flush When Full) match this lower bound and are strongly competitive, while other algorithms, such as LIFO (Last In First Out) and LFU (Least Frequently Used) have no upper bounds on the competitive ratio \[8\]. For randomized algorithms, Fiat et al. \[16\] proved a lower bound of $H_k$ on the competitive ratio and gave an algorithm, denoted Mark, which is $2H_k - 1$ competitive. A series of strongly competitive randomized algorithms were proposed in \[1, 5, 11, 23\], each of them improving over its predecessors with respect to space and running time for processing a page, up to $O(k)$ space and $O(\log k)$ time \[11\].

Perhaps the biggest drawback of competitive analysis is that it provides worst-case guarantees which happen for inputs that are encountered in practice next-to-never. In practice, it is common knowledge that some algorithms consistently outperform others by wide margins, despite the same competitive ratios. For instance, it is well established that LRU achieves at most four times as many cache misses as the optimal algorithm \[27\], which makes it (together with its variants) very popular in practice \[26\]. This means upper bounds provided by competitive analysis on the performance of paging algorithms are of little use in practice. Nonetheless, competitive analysis is a simple and useful tool towards gaining insights regarding algorithm behavior. In particular, a structure keeping track of the behavior of an optimal offline algorithm is at the heart of all strongly competitive randomized paging. We use this to obtain algorithms performing few cache misses.

To address the gaps between the theoretical guarantees provided by competitive analysis and the observed behavior in practice, a variety of models have been proposed. One line of research is concerned with restricted versions of competitive analysis, such as the diffuse adversary \[22\] or loose competitiveness \[27\]. Other approaches consider comparing algorithms directly, without relating them to an optimal offline algorithm. Relevant examples include the Max/Max ratio \[7\], the random order ratio \[21\], the relative worst order ratio \[10\], and bijective analysis and average analysis \[3\].

A characteristic of real-world inputs that the competitive analysis fails to take into account is locality of reference, which means that typically a small number of distinct pages is accessed during some time interval. Motivated by this input behavior, several models to reflect locality of reference have been proposed. In the working set model \[12, 13\] the paging strategy takes into account the most recently used pages, denoted working set. The access graph model \[9, 15, 18\] restricts input sequences by confining the next request to a restricted set of pages depending on the current request. In \[2, 14\] locality of reference is a function on the input and algorithms are analyzed using the cache size and this function. Many of these approaches are concerned with separating existing paging
algorithms to explain the differences observed in practice. In particular, several approaches (e.g. 
diffuse adversary, bijective analysis combined with locality of reference [4]) single out LRU as the 
best algorithm in the respective setting. In certain cases, these models also resulted in the design of 
new algorithms. Examples include RLRU (Retrospective LRU) [10] and FARL (Farthest-To-Last-
Request) [8, 17] which were designed according to the relative worst order ratio and access graph 
model respectively. Another paging algorithm, developed in the systems community, is EELRU 
(Early Eviction LRU) [25]. It simulates many algorithms and chooses the most promising one when 
deciding the page to evict. It outperforms LRU on many real-world traces.

Our contributions. Our contributions are three-fold. Motivated by properties of existing real-
world input traces from various applications, we first propose an input parametrization that we 
denote attack rate, which quantifies the “evilness” of the adversary. It is based on the characteriza-
tion of the optimal solution using offset functions in [22] and uses the fact that for certain requests 
we know for sure whether they are in the memory of OPT or not. We give empirical results showing 
that real-world inputs exhibit a low attack rate compared to worst-case inputs.

Secondly, we analyze the competitive ratio of deterministic algorithms with respect to the attack 
rate. We show that algorithms with unbounded competitive ratio like LIFO and LFU do not profit 
from a low attack rate. For inputs with attack rate at most $r$, we provide a lower bound of $r$ on 
the competitive ratio. This is matched by a class of algorithms, that we denote OnOPT, which 
maintain their cache content as close as possible to an optimal offline solution as possible. LRU belongs to 
this class. Conversely, in general marking algorithms are shown to benefit less on inputs with low 
attack rates. Although the attack rate inherits the simplicity of classical competitive analysis, the 
obtained bounds are more realistic. Our input parametrization further implies upper bounds on the 
fault rate for $r$-competitive algorithms, which usually lie far below 1%. We show experimentally 
that our parametrized bound for LRU outperforms the fault rate prediction using parametrizations 
based on locality of reference for various settings, especially for not too large cache sizes.

Finally, motivated by the optimal competitive guarantees and the fault rate analysis for the 
class OnOPT, we use a priority based framework to construct potential candidates in OnOPT 
to outperform LRU. We propose an algorithm from this class, denoted RDM, which outperforms 
LRU and two of its variants, RLRU and EELRU, on many real-world traces. This contrasts many 
models that single out LRU as the best paging algorithm. Since OnOPT contains deterministic 
algorithms we extracted from the strongly competitive randomized algorithm Equitable [1, 5], this 
shows that insights from classical competitive analysis can help to design algorithms with low fault 
rate on real world inputs.

2 Input parametrization

A classical optimal offline algorithm, denoted LFD (Longest Forward Distance), has been proposed 
in [6], and works by evicting, upon a cache miss, the page in cache which is requested farthest in 
the future. However, when designing an online algorithm we do not know the future requests. 
One approach is to keep the cache content of the online algorithm as close as possible to the one of 
LFD. An elegant characterization of the possible cache content of LFD is given in the context of 
work functions in [22]. Based on the request sequence seen thus far, the page currently requested 
can fall in one of the three categories below.

(I) Revealed pages, this are requests to pages we know for sure that they are in LFDs cache.
(II) *Unrevealed pages*, which might be in the cache of LFD, depending on the future requests.

(III) Pages which are for sure not in the cache of LFD and thus LFD faults on them.

Intuitively, we can fault on revealed pages only if we do not take advantage of the information from the sequence seen so far. If we view paging as a game, where the players are the online algorithm and the adversary constructing the input, requests to revealed pages do not turn out to be attacks, if the online algorithm keeps all revealed items in cache. An online algorithm which exhibits this property is LRU. Requests to type III pages, i.e. on which LFD faults, are due to the hardness of the input as the adversary pays as well. For the unrevealed requests the algorithm inflicts cache misses if it mispredicts the future.

Our experimental analysis of several real world traces (without consecutive identical requests) lead us to two observations. First, very many requests (regularly above 99%) are to revealed pages, and second, the ratio between requests to unrevealed pages and type III requests is quite small. Given a class of algorithms maintaining a good approximation of LFD’s cache content, the first observation leads to very small bounds of the fault rate and the second results in good guarantees for the empirical competitive ratio. This lead us to suspect that such a class may contain promising algorithms outperforming the existing ones.

### 2.1 Preliminaries

For some algorithm A, we denote by *cache configuration* the set of pages that are in the cache of A. For a fixed input sequence σ and some cache configuration C, the *offset function* ω maps C to the difference between the minimum cost of processing σ ending in C and the minimum cost of processing σ. A cache configuration C is denoted *valid* iff ω(C) = 0. In [22] it was shown that an offset function can be represented by k + 1 disjoint sets \( L_0, \ldots, L_k \), denoted *layers*, as follows. Initially, each layer in \( L_1, \ldots, L_k \) contains one of the first k pairwise distinct pages and \( L_0 \) contains all the remaining pages. Denoting by \( \omega^p \) the partition after processing p, we have the following:

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

The *support* of the offset function \( \omega \) is defined as \( S(\omega) = \bigcup_{i=1}^{k} L_i \). Denoting by *singleton* a set having a single element, let \( r \) be the smallest index such that all layers \( L_r, \ldots, L_k \) are singletons. We denote by *revealed pages* the set \( R(\omega) = \bigcup_{i=r}^{k} L_i \). Intuitively, the layer representation keeps track of the possible cache configurations of the optimal offline algorithm. To this end, in [22] it has been shown that a cache configuration is valid iff it holds that for each \( i \in \{1, \ldots, k\} \) we have \( |C \cap (\bigcup_{j=1}^{i} L_j)| \leq i \). This implies that any valid configuration will contain all revealed pages and no page from \( L_0 \). If the support contains only revealed pages, we know exactly the content of the cache of LFD and we say we are in a *cone*.

**Fact 1** Between two requests in \( L_0 \), at most \( k - 1 \) pairwise distinct pages are requested. Moreover, LFD faults on some page p iff \( p \in L_0 \).

---

1We use a different, yet equivalent notation to the one in [22].
2.2 Attack rate

As previously stated, any valid configuration contains all revealed pages and no item from \( L_0 \), and thus the remaining \( k - |R(\omega)| \) pages are unrevealed elements from the support. Since revealed pages can be identified by an online algorithm, it is desirable for algorithms not to have cache misses on requests to revealed pages. Given some input sequence \( \sigma \) let \( \lambda_r(\sigma), \lambda_u(\sigma), \) and \( \lambda_0(\sigma) \) denote the number of requests to revealed pages, unrevealed pages in the support, and pages in \( L_0 \) respectively.

**Definition 1** For some input \( \sigma \), the attack rate \( r(\sigma) \) is defined as \( r(\sigma) = \frac{\lambda_0(\sigma) + \lambda_u(\sigma)}{\lambda_0(\sigma)} \). Also, we denote by \( I(r) \) the set of inputs having an attack rate at most \( r \), i.e. \( I(r) = \{ \sigma | r(\sigma) \leq r \} \).

Taking into account that LFD always faults on requests in \( L_0 \), our attack rate is an upper bound on the competitive ratio for any algorithm that does not fault on revealed pages. We note that the attack rate \( r \in \mathbb{Q} \) is in the range \([1, k]\) and thus \( I(k) \) contains all possible inputs. We get \( r = 1 \) when \( \lambda_u = 0 \) and \( r = k \) by requesting a page in \( L_0 \) followed by \( k - 1 \) unrevealed pages.

**Attack rate in real-world inputs.** We conduct experiments on a collection of cache traces extracted from various applications\(^2\) from both Linux and Windows NT operating systems, ranging from gcc compiler to an AI program playing the game “Go”, and from a formula-rewrite program (grobner) to MSPowerpoint [19] (all datasets are described in the full version attached). We compare the observed attack rate \( r \) against \( k \) for all these traces, as \( r \) is an upper bound on the competitive ratio for algorithms that are always in a valid configuration, while \( k \) is the best upper bound in the classical model. The results in Figure 1 show that in practice \( r \) is significantly smaller than the cache size and converges to 1 as the cache sizes increases (and more pages fit in memory). Moreover, for most ranges and applications \( r \) is below \( 0.2k \), meaning that algorithms that are always in a valid configuration should improve the worst case guarantees on the number of cache misses given by standard competitive analysis by 500%, though in many cases the improvement is much larger.

**Fault rate.** The fault rate is a practical measure of the efficiency of paging algorithms and is defined as the ratio between the number of cache misses and the input size. Unfortunately,

\(^2\)We used all the available original reference traces from [http://www.cs.amherst.edu/~sfkaplan/research/trace-reduction/index.html](http://www.cs.amherst.edu/~sfkaplan/research/trace-reduction/index.html).
competitive analysis by itself does not capture this measure, as it is easy to construct inputs and algorithms which achieve the same competitive ratio and very different fault rates. In our setting, for algorithms that are always in a valid configuration the fault rate is at most \( \frac{\lambda_0 + \lambda_u + \lambda_r}{\lambda_0 + \lambda_u + \lambda_r} \); this bound extends trivially to all \( r \)-competitive algorithms. We stress that this is a guaranteed upper bound on the fault rate, however certain algorithms perform less than this amount. Empirical results showed that in practice the value of \( \lambda_r \) is very large, typically amounting to more than 99% of the requests, and thus our upper bound on the fault rate is usually smaller than 1%.

We compare our guaranteed upper bound on the fault rate against other approaches using input parametrization, based on locality of reference. We recall that for decades it is known that real-life inputs exhibit locality of reference [12]. To quantify the locality of reference in the input, Albers et al. [2] proposed two ways of dealing with locality of reference by inspecting all subintervals of the input having size \( n \) and measuring the maximum and average numbers of distinct pages in these sliding windows respectively. For these settings, denoted \textit{Max-model} and \textit{Average-model} respectively, they analyzed the fault rate on which they gave upper bounds for several text-book algorithms, such as LRU, FIFO, and Marking. More recently, Dorrigiv et al. [14] gave a measure quantifying the \textit{non-locality} existent in the input and also gave upper bounds on the fault rate of many classical algorithms. Perhaps the biggest drawback of the approaches based on (non-)locality of reference is that they do not distinguish between revealed and unrevealed pages and thus include revealed pages in their predicted upper bounds even though algorithms that are always in a valid configuration never fault on such pages (and LRU is one such algorithm). This tends to result in predicting higher upper bounds than necessary, especially for not too large cache sizes.

We conduct experiments which show the fault rate as predicted by the four approaches, together with the actual fault rate of LRU. In Figure 2 we show results for some selected datasets (the results for all datasets are given in the full version attached in Figures 2 and 3). They show that for all inputs and all cache sizes our approach gives more realistic upper bounds on the fault rate of LRU than non-locality of reference and locality of reference in the average model, for some datasets by huge margins, i.e. factors larger than 100. Typically for cache sizes smaller than 1/3 of the pageset our parametrization clearly outperforms locality of reference in the Max setting, in many cases by factors of thousands. Up to 2/3 of the cache size our approach still outperforms it but by smaller margins, whereas for cache sizes exceeding approximately two thirds of the pageset the locality of reference in the Max model gives the best upper bounds, though by very small margins. On the one hand, the Max model allows good theoretical bounds because it is based on a worst case parameter of the input. On the other hand, even small subintervals without locality of reference cause bad predictions for the whole input. The larger the input sequence, the higher the probability to find such an interval. This happens for example if the working set of a program changes\(^3\). We conclude that overall our parametrization provides tighter bounds than existent locality of reference for \( r \)-competitive algorithms in general and LRU in particular.

\(^3\)For all datasets we considered the full input to compute the parameters for locality of reference in the Max model, as opposed to [2] where they truncated inputs longer than \( 10^7 \) requests; in our experiments the input size ranges from \( 7 \cdot 10^6 \) to \( 5 \cdot 10^8 \), hence the slightly different behavior compared to [2] for the same application.
Figure 2: The offset function-based predicted fault rate $of = \frac{\lambda_0 + \lambda_u}{\lambda_0 + \lambda_u + \lambda_r}$, the locality of reference in the Max- and Average-model, and the non-locality of reference, with the performance of LRU for three selected datasets. The x-axis shows the cache size and the y-axis shows the fault rate.

3 Input-parametrized competitive ratio

3.1 Priority-based paging algorithms

Most paging algorithms can be viewed as consisting of two components: a predictor and an eviction policy. The predictor assigns priorities to pages in an attempt to guess the order of future requests. Based on the predictor, the strategy decides which page is to be evicted upon a cache miss. Without loss of generality we assume the smaller the priority of a page, the more in the future its next request is predicted. For instance, LRU may assign as priority for the current page the current timestamp and evict the page having the smallest priority. Depending on the eviction policy, we consider three classes of algorithms introduced below, namely CacheMin, Marking, and OnOPT.

CacheMin. Upon a cache miss, an algorithm in this class evicts the page in cache that is predicted to occur the farthest in the future, i.e. that has the smallest priority. Most textbook deterministic algorithms belong to this class. Setting for each request the current timestamp as priority yields LRU; if we set the priority to the negated current timestamp we obtain MRU. Similarly, setting the priority of a page to the last timestamp it faulted we obtain FIFO, and the negated of this value yields LIFO. Assigning for a page the request frequency as priority results in LFU.

Marking. The marking algorithms assign marks to pages and work in phases as follows. A phase begins when all pages in cache are marked and a cache miss occurs. In this case all pages are unmarked, the page in cache predicted to be requested farthest in the future is evicted, and the new page is loaded in cache and marked. For each request to some page $p$ within a phase, if $p$ is a cache hit it gets marked and if it’s a cache miss the unmarked page in cache predicted to be requested farthest in the future is evicted, after which $p$ is loaded in the cache and marked.

OnOPT. The algorithms in this class are based on the layer partition in [22] previously described. They always have a cache configuration identical to LFD if the priority assignment reflects future requests. This implies that they are always in a valid configuration according to the current work function. These algorithms maintain the layer partition and process some page $p$ by first applying
an eviction policy in the case of a cache miss followed by updating the layers, as shown in [11]. The eviction policy is implemented as follows. If \( p \) is in the cache then nothing needs to be done. If \( p \) is not in the cache we distinguish between two cases: \( p \in L_0 \) and \( p \in L_i \) with \( i > 0 \). If \( p \in L_0 \) then the page in cache having the smallest priority is evicted. If \( p \in L_i \) and \( p \) triggers a cache fault, we first identify the layer \( L_j \) with \( j \geq i \) such that the cache contains exactly \( j \) pages in \( L_1 \cup \cdots \cup L_j \), i.e. \( |M \cap (\bigcup_{l=1}^{j} L_l)| = j \). The page in cache from \( L_1 \cup \cdots \cup L_j \) having the smallest priority is evicted. This eviction policy ensures that in the case that the priority assignment reflects the future requests, the cache contents of the online algorithm and LFD are identical.

We note that, since implementations are given, each of the three classes can be viewed as a framework which, provided with a priority assignment, results in a paging algorithm. Assuming that the only priority change happens for the current request, algorithms in all three classes support very fast implementations and thus are not prohibitively expensive in practice. Algorithms in the CACHEMIN and MARKING classes can be easily implemented using a dictionary and a priority queue, which take \( O(k) \) space and \( O(\log k) \) time per page request. For the algorithms in the ONOPT class we showed in [11] how to implement them in \( O(m) \) space and \( O(\log m) \) time per request where \( m \) is the size of the pageset. A variant with similar behavior and supporting a faster implementation can be achieved by using the forgiveness mechanism introduced in [5]. The resulted implementation uses \( O(k) \) space and \( O(\log k) \) time per request as well, however the theoretical guarantees are compromised. Nonetheless, experimental results show that the number of cache misses done by the two implementations is virtually identical. However the bounds provided are generic and apply to all algorithms in a given framework, but certain algorithms can be implemented significantly faster, e.g. FIFO takes \( O(1) \) time per request.

### 3.2 Competitive analysis

In this section we give lower and upper bounds on the competitive ratio for deterministic paging algorithms, as a function of the attack rate \( r \). The results are summarized in Table 1.

**Lemma 1** The competitive ratio for any deterministic paging algorithm on an input in \( I(r) \), for any arbitrary rational \( r \in [1 \ldots k] \), is at least \( r \).

**Proof.** Recall that \( I(r) \) contains all inputs having the attack rate at most \( r \). Consider some arbitrary deterministic algorithm \( A \). To prove the claimed bound we build an input sequence on which \( A \) is guaranteed to perform \( r \) times more cache misses than LFD. We consider a set containing \( k + 1 \) pages, on which we build a subsequence which starts in a cone and ends in a cone. We first use the standard lower bound construction from classical competitive analysis and request \( k \) pages such that for each request \( A \) does a cache miss and \( \lambda_0 = 1 \). We then request as many unrevealed pages as necessary until we end in a cone. Since the only first request is in \( L_0 \) and we end in a cone, for each such subsequence we have \( \lambda_0 = 1 \) and \( \lambda_u = k - 1 \). Also, by construction \( A \) does at least \( k \) cache misses.

---

<table>
<thead>
<tr>
<th>Class</th>
<th>Competitive ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>CACHEMIN</td>
<td>( \infty )</td>
</tr>
<tr>
<td>MARKING</td>
<td>([2r - 1, 2r] )</td>
</tr>
<tr>
<td>ONOPT</td>
<td>( \frac{r}{r} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Competitive ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>LFU, MRU, LIFO</td>
<td>( \infty )</td>
</tr>
<tr>
<td>FWF</td>
<td>([2r - 1, 2r] )</td>
</tr>
<tr>
<td>LRU, FIFO</td>
<td>( \frac{r}{r} )</td>
</tr>
</tbody>
</table>

Table 1: The guaranteed competitive ratio for the generic classes and for classic algorithms.
We request this subsequence $n_1$ times using the same set of $k+1$ pairwise distinct pages, followed by $n_2$ requests to pages in $L_0$ that were never requested. For such an input, we have $\lambda_0 = n_1 + n_2$ and $\lambda_u = (k - 1)n_1$, which leads to an attack ratio $r = \frac{kn_1 + n_2}{n_1 + n_2}$. Using the fact that LFD faults only on requests in $L_0$, the competitive ratio is at least $\frac{kn_1 + n_2}{n_1 + n_2} = r$. Combining different values for $n_1$ and $n_2$ we obtain any possible rational value for $r \in [1, k]$ and the proof concludes.

**Fact 2** Any algorithm is $1$-competitive on inputs in $I(1)$.

**CacheMin, LIFO, MRU, and LFU.** Both LIFO and LFU belong to the CacheMin class, and for both of them the arguments from the standard competitive analysis carry on to our parametrized inputs. For LIFO, after the first $k$-pairwise distinct pages we request two new pages $x$ and $y$ alternately and infinitely, i.e. the input sequence $\sigma = p_1, \ldots, p_k, (xy)^\infty$. LIFO does a cache miss on each request while OPT does only 2 cache misses (we exclude the first $k$ pairwise distinct pages). We note that we used an input having attack ratio of $3/2$, but it can be easily extended to any value $r > 1$. The same argument holds for MRU. For LFU, we request the first $k$ pairwise distinct items $n$ times each and then we cyclically request two new pairwise distinct pages $n - 1$ times each, i.e. the input is $\sigma = (p_1, \ldots, p_k)^n(p_{k+1}, p_{k+2})^{n-1}$. Similarly to LIFO and MRU, LFU faults on each page while OPT incurs 2 misses. For infinitely large $n$ the competitive ratio is unbounded. Similarly to LIFO, the attack rate is $3/2$ but can be extended to any value in $(1, \ldots k]$.

**Marking algorithms.** For the marking algorithms, we first show that they are $2r$-competitive and then we show that there exist priority assignments which are very close to this bound. Although FWF is not in our Marking framework, the following result applies to it as well, both for the lower and upper bounds.

**Lemma 2** The competitive ratio for any marking algorithm on an input in $I(r)$ is at most $\min(2r, k)$; there exist marking algorithms which are at least $\min(2r - 1, k)$-competitive for any value of $r$.

**Proof.** Due to space constraints, the proof is included only in the full version. To get the lower bound, we combine different types of inputs for a Marking algorithms using MRU-like priorities. For the upper bound, we use the fact that on a subsequence of size $k$ a page faults at most twice.

**OnOPT and FIFO.** By construction, the algorithms in the OnOPT class never fault on revealed items and are thus $r$-competitive on inputs in $I(r)$. Since LRU is in OnOPT, it is also $r$-competitive. For FIFO, we show that it is $r$-competitive in spite of the fact that it is not always in a valid configuration. It is possible to build inputs for which FIFO faults on revealed page.

**Lemma 3** FIFO is $r$-competitive on any input in $I(r)$.

**Proof.** We split the input in phases where each phase starts with a request in $L_0$ and finishes just before the next request in $L_0$. By Fact 1 each phase consists of at most $k$ pairwise distinct pages. We note that a page can fault at most once during a phase, since $k$ more pairwise distinct pages are required until the same page faults again. Since at the beginning of a phase all pages, except for the request in $L_0$ starting the phase, are unrevealed, this immediately implies that each page in this phase is requested exactly once from $L_0$ or from an unrevealed layer. We thus can charge each cache miss on a page to a request to the same page in $L_0$ or an unrevealed layer. We obtain that overall FIFO does $\lambda_u + \lambda_0$ cache misses, while OPT does $\lambda_0$ and the proof concludes.
4 An algorithm better than LRU

We first give a priority assignment to be used in the framework of OnOPT algorithms previously introduced, which leads to an algorithm that we denote *Recency Duration Mix* (RDM). As its name implies, it combines two priority policies, one based on recency and the other on the time-frame that pages spend in support. We conduct experiments showing that for most inputs and cache sizes RDM outperforms not only LRU, but also two of its variants shown to behave well in practice.

4.1 RDM

We recall that the framework of OnOPT algorithms ensures that regardless of the priority assignment we get an $r$-competitive algorithm which is always in a valid configuration. This gives us the freedom to explore various priority policies. Furthermore, this framework can be implemented efficiently with respect to both space and running time to give it practical value.

We use a global counter $t$, which keeps track of the amount of requests to pages in $L_0$ and unrevealed layers. Thus before assigning a priority to the requested page $p$, we increment $t$ only if $p$ is not revealed. We do so because requests to revealed pages trigger only a permutation of the revealed layers. More precisely, only the layer representation of the offset function changes, but not the function itself. Thus, such requests do not provide any new information about the possible states of an optimal solution and consequently should not affect the priority assignment. Also, for each page $p$ in the support we store a value $t_0$ which stores the value of $t$ at the time that $p$ entered the support. More exactly, for any request $p$ from $L_0$ we set $t_0(p) = t$. We describe the two priority assignment strategies that we will later combine into a new priority assignment which we plug into the OnOPT framework to obtain RDM.

**Recency.** We assign each page upon request the current counter $t$ as priority. It is inspired by LRU in that it assigns for each page $p$ the current counter as priority, but unlike LRU our counter ignores requests to revealed pages.

**Duration.** A major drawback of LRU is that it performs very bad when repeatedly requesting the same sequence having more than $k$ pages, e.g. repeatedly scanning an array. This priority policy addresses this drawback taking into account the time that a page spent in the support. When requested, each page is assigned as priority the value $t - t_0$. The intuition behind this strategy is that if a page is frequently requested during a period, it remains in OPT's cache during this period and gets a high priority. A particular strength of this strategy is the fact that it adapts to repeatedly requesting the same sequence of more than $k$ pages. After the first iteration all the pages are in the support, at the second iteration the first $k - 1$ requests become revealed and get their priorities increased while the remaining ones are evicted from the support and at their next request they are assigned a new $t_0$ value giving them low priorities, avoiding an LRU-like behavior. We have empirically determined that assigning priorities according to the duration policy alone outperforms LRU for certain datasets and cache sizes. However, using a linear combination of recency and duration the performance improves significantly. Overall, we have achieved the best results when assigning for each page upon request the value $0.8t + 0.1(t - t_0)$ as priority, and this priority is used in the experimental results.
4.2 RDM on real-world traces

We conduct experimental to compare the performance of RDM against the performance of LRU and two of its variants which were shown to behave better than LRU in practice, namely RLRU [10] and EELRU [25]. RLRU (Retrospective LRU) was proposed in [10], where it was also proven to be better than LRU with respect to the relative worst order ratio. It is a marking-like algorithm which assigns marks based on what OPT would have in cache and evicts unmarked pages using a LRU strategy. Empirical results over various datasets showed RLRU to perform fewer cache misses than LRU, though the differences observed were small (mostly up to 5% improvement). EELRU (Early Eviction LRU) is an adaptive paging algorithm from a less theoretical direction. It simulates a large collection of about 256 parametrized instances of an algorithm which is a mix of LRU and MRU (Most Recently Used). To decide which page to evict EELRU consults the results of these 256 instances for the recent past, and the most promising is simulated on the actual request. If none is promising, it switches to LRU and it is guaranteed that it can never be worse than a factor of three compared to LRU. In [25] it was shown that EELRU achieves good performance compared to LRU in practice, outperforming LRU on many datasets, at times by significant amounts.

For each dataset and cache size, we measure for each of the four algorithms considered the competitive ratio, i.e. the number of cache misses performed normalized by the performance of OPT. In Figure 3 we show the results for three datasets (the results for all datasets are given in the full version in Figures 4 and 5). The results show that on all datasets and for all cache sizes RLRU has a similar performance to LRU, though it outperforms it consistently by small margins. For EELRU, we note that gnuplot is the only dataset on which it outperforms all other algorithms by large margins. For all the remaining inputs, except for certain cache sizes on the espresso and lindsay datasets, EELRU is at least as good as LRU and RLRU; however, on several datasets (e.g. compress, gcc, grobner) there are cache sizes for which it outperforms LRU by factors ranging from two to four. In what concerns RDM, it outperforms LRU and RLRU on all datasets and for all cache sizes, except for a narrow range on the gcc dataset. The margins vary among datasets, with improvements by more than a factor of 100% on three datasets (compress, grobner, and go) and more than 10% on most of the remaining datasets. Moreover, it rarely happens that RDM has a competitive ratio of more than two. Finally, we note that, except for gnuplot, RDM outperforms EELRU as well on most cache sizes, in many cases by significant margins.

Figure 3: The empirical competitive ratio on selected datasets for RDM, LRU, RLRU, and EELRU. The x-axis shows the cache size and the y-axis shows the competitive ratio.
References


1 Introduction

Paging has a strong practical motivation and is one of the most studied problems in the field of online algorithms. We are provided with a cache of size $k$ and a memory of infinite size, and must process page requests online, i.e. without any knowledge about future requests. If the requested page is in cache, a cache hit occurs and the algorithm proceeds at no cost. Otherwise, a cache miss occurs and the algorithm must load the page in the cache. If the cache was full, one page must be evicted to accommodate the one requested. The cost is given by the number of cache misses.

Traditionally, the quality of online algorithms in general and paging in particular is measured by comparing their cost against the cost of an optimal offline algorithm, i.e. an algorithm that is provided with the input beforehand and processes it optimally. This measure, denoted competitive ratio \cite{20, 24}, states that some online algorithm $A$ is $c$-competitive if for any input sequence it holds that $\text{cost}(A) \leq c \cdot \text{cost}(\text{OPT}) + b$, where $\text{cost}(A)$ and $\text{cost}(\text{OPT})$ denote the cost of $A$ and the optimal cost respectively, and $b$ is a constant. For deterministic paging algorithms, a lower bound of $k$ on the competitive ratio was shown in \cite{24}. Several algorithms, such as LRU (Least Recently Used), FIFO (First In First Out), and FWF (Flush When Full) match this lower bound and are strongly competitive, while other algorithms, such as LIFO (Last In First Out) and LFU (Least Frequently Used) have no upper bounds on the competitive ratio \cite{8}. For randomized algorithms, Fiat et al. \cite{16} proved a lower bound of $H_k$ on the competitive ratio and gave an algorithm, denoted Mark, which is $2H_k - 1$ competitive. A series of strongly competitive randomized algorithms were proposed in \cite{1, 5, 11, 23}, each of them improving over its predecessors with respect to space and running time for processing a page, up to $O(k)$ space and $O(\log k)$ time \cite{11}.

Perhaps the biggest drawback of competitive analysis is that it provides worst-case guarantees which happen for inputs that are encountered in practice next-to-never. In practice, it is common knowledge that some algorithms consistently outperform others by wide margins, despite the same competitive ratios. For instance, it is well established that LRU achieves at most four times as many cache misses as the optimal algorithm \cite{27}, which makes it (together with its variants) very popular in practice \cite{26}. This means upper bounds provided by competitive analysis on the performance of paging algorithms are of little use in practice. Nonetheless, competitive analysis is a simple and useful tool towards gaining insights regarding algorithm behavior. In particular, a structure keeping track of the behavior of an optimal offline algorithm is at the heart of all strongly competitive randomized paging. We use this to obtain algorithms performing few cache misses.

To address the gaps between the theoretical guarantees provided by competitive analysis and the observed behavior in practice, a variety of models have been proposed. One line of research is concerned with restricted versions of competitive analysis, such as the diffuse adversary \cite{22} or loose competitiveness \cite{27}. Other approaches consider comparing algorithms directly, without relating them to an optimal offline algorithm. Relevant examples include the Max/Max ratio \cite{7}, the random order ratio \cite{21}, the relative worst order ratio \cite{10}, and bijective analysis and average analysis \cite{3}.

A characteristic of real-world inputs that the competitive analysis fails to take into account is locality of reference, which means that typically a small number of distinct pages is accessed during some time interval. Motivated by this input behavior, several models to reflect locality of reference have been proposed. In the working set model \cite{12, 13} the paging strategy takes into account the most recently used pages, denoted working set. The access graph model \cite{9, 15, 18} restricts input sequences by confining the next request to a restricted set of pages depending on the current request. In \cite{2, 14} locality of reference is a function on the input and algorithms are analyzed using the cache size and this function. Many of these approaches are concerned with separating existing paging
algorithms to explain the differences observed in practice. In particular, several approaches (e.g. diffuse adversary, bijective analysis combined with locality of reference [4]) single out LRU as the best algorithm in the respective setting. In certain cases, these models also resulted in the design of new algorithms. Examples include RLRU (Retrospective LRU) [10] and FARL (Farthest-To-Last-Request) [8, 17] which were designed according to the relative worst order ratio and access graph model respectively. Another paging algorithm, developed in the systems community, is EELRU (Early Eviction LRU) [25]. It simulates many algorithms and chooses the most promising one when deciding the page to evict. It outperforms LRU on many real-world traces.

Our contributions. Our contributions are three-fold. Motivated by properties of existing real-world input traces from various applications, we first propose an input parametrization that we denote attack rate, which quantifies the “evilness” of the adversary. It is based on the characterization of the optimal solution using offset functions in [22] and uses the fact that for certain requests we know for sure whether they are in the memory of OPT or not. We give empirical results showing that real-world inputs exhibit a low attack rate compared to worst-case inputs.

Secondly, we analyze the competitive ratio of deterministic algorithms with respect to the attack rate. We show that algorithms with unbounded competitive ratio like LIFO and LFU do not profit from a low attack rate. For inputs with attack rate at most $r$, we provide a lower bound of $r$ on the competitive ratio. This is matched by a class of algorithms, that we denote OnOPT, which maintain their cache content as close to an optimal offline solution as possible. LRU belongs to this class. Conversely, in general marking algorithms are shown to benefit less on inputs with low attack rates. Although the attack rate inherits the simplicity of classical competitive analysis, the obtained bounds are more realistic. Our input parametrization further implies upper bounds on the fault rate for $r$-competitive algorithms, which usually lie far below 1%. We show experimentally that our parametrized bound for LRU outperforms the fault rate prediction using parametrizations based on locality of reference for various settings, especially for not too large cache sizes.

Finally, motivated by the optimal competitive guarantees and the fault rate analysis for the class OnOPT, we use a priority based framework to construct potential candidates in OnOPT to outperform LRU. We propose an algorithm from this class, denoted RDM, which outperforms LRU and two of its variants, RLRU and EELRU, on many real-world traces. This contrasts many models that single out LRU as the best paging algorithm. Since OnOPT contains deterministic algorithms we extracted from the strongly competitive randomized algorithm Equitable [1, 5], this shows that insights from classical competitive analysis can help to design algorithms with low fault rate on real world inputs.

2 Input parametrization

A classical optimal offline algorithm, denoted LFD (Longest Forward Distance), has been proposed in [6], and works by evicting, upon a cache miss, the page in cache which is requested farthest in the future. However, when designing an on-line algorithm we do not know the future requests. One approach is to keep the cache content of the online algorithm as close as possible to the one of LFD. An elegant characterization of the possible cache content of LFD is given in the context of work functions in [22]. Based on the request sequence seen thus far, the page currently requested can fall in one of the three categories below.

(I) Requests to revealed pages, this are requests to pages we know for sure that they are in LFDs
(II) Requests to unrevealed pages, which might be in the cache of LFD, depending on the future requests.

(III) Requests to pages which are for sure not in the cache of LFD and thus LFD faults on them.

Intuitively, we can fault on revealed pages only if we do not take advantage of the information from the sequence seen so far. If we view paging as a game, where the players are the online algorithm and the adversary constructing the input, requests to revealed pages do not turn out to be attacks, if the online algorithm keeps all revealed items in cache. An online algorithm which exhibits this property is LRU. Requests to type III pages, i.e. on which LFD faults, are due to the hardness of the input as the adversary pays as well. For the unrevealed requests the algorithm inflicts cache misses if it mispredicts the future.

Our experimental analysis of several real world traces (without consecutive identical requests) lead us to two observations. First, very many requests (regularly above 99%) are to revealed pages, and second, the ratio between requests to unrevealed pages and type III requests is quite small. Given a class of algorithms maintaining a good approximation of LFDs cache content, the first observation leads to very small bounds of the fault rate and the second results in good guarantees for the empirical competitive ratio. This lead us to suspect that such a class may contain promising algorithms outperforming the existing ones.

2.1 Preliminaries

For some algorithm A, we denote by cache configuration the set of pages that are in the cache of A. For a fixed input sequence σ and some cache configuration C, the offset function ω maps C to the difference between the minimum cost of processing σ ending in C and the minimum cost of processing σ. A cache configuration C is denoted valid iff ω(C) = 0. In [22] it was shown that an offset function can be represented by $k + 1$ disjoint sets $L_0, \ldots, L_k$, denoted layers, as follows. Initially, each layer in $L_1, \ldots, L_k$ contains one of the first $k$ pairwise distinct pages and $L_0$ contains all the remaining pages. Denoting by $\omega^p$ the partition after processing $p$, we have the following:

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

The support of the offset function $\omega$ is defined as $S(\omega) = \bigcup_{i=1}^{k} L_i$. Denoting by singleton a set having a single element, let $r$ be the smallest index such that all layers $L_r, \ldots, L_k$ are singletons. We denote by revealed pages the set $R(\omega) = \bigcup_{i=r}^{k} L_i$. Intuitively, the layer representation keeps track of the possible cache configurations of the optimal offline algorithm. To this end, in [22] it has been shown that a cache configuration is valid iff it holds that for each $i \in \{1, \ldots, k\}$ we have $|C \cap (\bigcup_{j=1}^{i} L_j)| \leq i$. This implies that any valid configuration will contain all revealed pages and no page from $L_0$. If the support contains only revealed pages, we know exactly the content of the cache of LFD and we say we are in a cone.

Fact 1 Between two requests in $L_0$, at most $k - 1$ pairwise distinct pages are requested. Moreover, LFD faults on some page $p$ iff $p \in L_0$.

---

4We use a different, yet equivalent notation to the one in [22].
2.2 Attack rate

As previously stated, any valid configuration contains all revealed pages and no item from $L_0$, and thus the remaining $k - |R(\omega)|$ pages are unrevealed elements from the support. Since revealed pages can be identified by an online algorithm, it is desirable for algorithms not to have cache misses on requests to revealed pages. Given some input sequence $\sigma$ let $\lambda_r(\sigma)$, $\lambda_u(\sigma)$, and $\lambda_0(\sigma)$ denote the number of requests in $\sigma$ to revealed pages, unrevealed pages in the support, and pages that are not in support respectively.

**Definition 1** For some input $\sigma$, the attack rate $r(\sigma)$ is defined as $r(\sigma) = \frac{\lambda_0(\sigma) + \lambda_u(\sigma)}{\lambda_0(\sigma)}$. Also, we denote by $I(r)$ the set of inputs having an attack rate at most $r$, i.e. $I(r) = \{\sigma | r(\sigma) \leq r\}$.

Taking into account that LFD always faults on requests in $L_0$, our attack rate is an upper bound on the competitive ratio for any algorithm that does not fault on revealed pages. We note that the attack rate $r \in \mathbb{Q}$ is in the range $[1, k]$ and thus $I(k)$ contains all possible inputs. We get $r = 1$ when $\lambda_u = 0$ and we obtain $r = k$ by requesting a page in $L_0$ followed by $k - 1$ requests to unrevealed pages in the support.

<table>
<thead>
<tr>
<th>Application</th>
<th>OS</th>
<th>Collected by</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>espresso</td>
<td>Linux</td>
<td>VMTrace</td>
<td>circuit simulator</td>
</tr>
<tr>
<td>gcc-2.7.2</td>
<td>Linux</td>
<td>VMTrace</td>
<td>GNU C/C++ compiler</td>
</tr>
<tr>
<td>gnuplot</td>
<td>Linux</td>
<td>VMTrace</td>
<td>GNU plotting utility</td>
</tr>
<tr>
<td>grobner</td>
<td>Linux</td>
<td>VMTrace</td>
<td>calculated Grobner basis functions</td>
</tr>
<tr>
<td>gs3.33</td>
<td>Linux</td>
<td>VMTrace</td>
<td>GhostScript (postscript interpreter)</td>
</tr>
<tr>
<td>lindsay</td>
<td>Linux</td>
<td>VMTrace</td>
<td>hypercube simulator</td>
</tr>
<tr>
<td>p2c</td>
<td>Linux</td>
<td>VMTrace</td>
<td>Pascal to C transformer</td>
</tr>
<tr>
<td>acroread</td>
<td>Windows NT</td>
<td>Etch</td>
<td>Acrobat Reader</td>
</tr>
<tr>
<td>cc1</td>
<td>Windows NT</td>
<td>Etch</td>
<td>Compiler core for gcc</td>
</tr>
<tr>
<td>compress</td>
<td>Windows NT</td>
<td>Etch</td>
<td>Compression utility</td>
</tr>
<tr>
<td>go</td>
<td>Windows NT</td>
<td>Etch</td>
<td>AI program playing “Go”</td>
</tr>
<tr>
<td>netscape</td>
<td>Windows NT</td>
<td>Etch</td>
<td>Netscape web browser</td>
</tr>
<tr>
<td>powerpoint</td>
<td>Windows NT</td>
<td>Etch</td>
<td>MS Powerpoint</td>
</tr>
<tr>
<td>vortex</td>
<td>Windows NT</td>
<td>Etch</td>
<td>Database program</td>
</tr>
<tr>
<td>winword</td>
<td>Windows NT</td>
<td>Etch</td>
<td>MS Word</td>
</tr>
</tbody>
</table>

Table 1: The description of the datasets we ran experiments on. The page size was 4KB [19].

**Attack rate in real-world inputs.** We conduct experiments on a collection of cache traces extracted from various applications$^5$ from both Linux and Windows NT operating systems, ranging from gcc compiler to an AI program playing the game “Go”, and from a formula-rewrite program (grobner) to MS Powerpoint (all datasets are described in Table 1). We compare the observed attack rate $r$ against $k$ for all these traces, as $r$ is an upper bound on the competitive ratio for algorithms that are always in a valid configuration, while $k$ is the best upper bound in the classical

---

$^5$We used all the available original reference traces from [http://www.cs.amherst.edu/~sfkaplan/research/trace-reduction/index.html](http://www.cs.amherst.edu/~sfkaplan/research/trace-reduction/index.html).
model. The results in Figure 1 show that in practice $r$ is significantly smaller than the cache size and converges to 1 as the cache sizes increases (and more pages fit in memory). Moreover, for most ranges and applications $r$ is below $0.2k$, meaning that algorithms that are always in a valid configuration should improve the worst case guarantees on the number of cache misses given by standard competitive analysis by 500%, though in many cases the improvement is much larger.

**Fault rate.** The fault rate is a practical measure of the efficiency of paging algorithms and is defined as the ratio between the number of cache misses and the input size. Unfortunately, competitive analysis by itself does not capture this measure, as it is easy to construct inputs and algorithms which achieve the same competitive ratio and very different fault rates. In our setting, for algorithms that are always in a valid configuration the fault rate is at most $\frac{\lambda_r + \lambda_0}{\lambda_u + \lambda_0 + \lambda_r}$; this bound extends trivially to all $r$-competitive algorithms. We stress that this is a guaranteed upper bound on the fault rate, however certain algorithms perform less than this amount. Empirical results showed that in practice the value of $\lambda_r$ is very large, typically amounting to more than 99% of the requests, and thus our upper bound on the fault rate is usually smaller than 1%.

We compare our guaranteed upper bound on the fault rate against other approaches using input parametrization, based on locality of reference. We recall that for decades it is known that real-life inputs exhibit locality of reference [12]. To quantify the locality of reference in the input, Albers et al. [2] proposed two ways of dealing with locality of reference by inspecting all subintervals of the input having size $n$ and measuring the maximum and average numbers of distinct pages in these sliding windows respectively. For these settings, denoted Max-model and Average-model respectively, they analyzed the fault rate on which they gave upper bounds for several text-book algorithms, such as LRU, FIFO, and Marking. More recently, Dorrigiv et al. [14] gave a measure quantifying the non-locality existent in the input and also gave upper bounds on the fault rate of many classical algorithms. Perhaps the biggest drawback of the approaches based on (non-)locality of reference is that they do not distinguish between revealed and unrevealed pages and thus include revealed pages in their predicted upper bounds even though algorithms that are always in a valid configuration never fault on such pages (and LRU is one such algorithm). This tends to result in predicting higher upper bounds than necessary, especially for not too large cache sizes.

![Figure 1: The attack rate $r$ in real-world inputs. The x-axis shows the percentage of pages that fit in cache, i.e. $k$, and the y-axis shows the ratio between $r$ and the cache size $k$. The constant line $f(x) = 1$ corresponds to $r = k$ and is the upper bound on the competitive ratio by standard competitive analysis.](image)

We conduct experiments which show the fault rate as predicted by the four approaches, together
with the actual fault rate of LRU. In Figures 2 and 3 we give the results for all datasets. They show that for all inputs and all cache sizes our approach gives more realistic upper bounds on the fault rate of LRU than non-locality of reference and locality of reference in the average model, for some datasets by huge margins, i.e. factors larger than 100. Typically for cache sizes smaller than 1/3 of the pageset our parametrization clearly outperforms locality of reference in the Max setting, in many cases by factors of thousands. Up to 2/3 of the cache size our approach still outperforms it but by smaller margins, whereas for cache sizes exceeding approximately two thirds of the pageset the locality of reference in the Max model gives the best upper bounds, though by very small margins. On the one hand, the Max model allows good theoretical bounds because it is based on a worst case parameter of the input. On the other hand, even small subintervals without locality of reference cause bad predictions for the whole input. The larger the input sequence, the higher the probability to find such an interval. This happens for example if the working set of a program changes. We conclude that overall our parametrization provides tighter bounds than existent locality of reference for \( r \)-competitive algorithms in general and LRU in particular.

3 Input-parametrized competitive ratio

3.1 Priority-based paging algorithms

Most paging algorithms can be viewed as consisting of two components: a predictor and an eviction policy. The predictor assigns priorities to pages in an attempt to guess the order of future requests. Based on the predictor, the strategy decides which page is to be evicted upon a cache miss. Without loss of generality we assume the smaller the priority of a page, the more in the future its next request is predicted. For instance, LRU may assign as priority for the current page the current timestamp and evict the page having the smallest priority. Depending on the eviction policy, we consider three classes of algorithms introduced below, namely CacheMin, Marking, and OnOPT.

CacheMin. Upon a cache miss, an algorithm in this class evicts the page in cache that is predicted to occur the farthest in the future, i.e. that has the smallest priority. Most text-book deterministic algorithms belong to this class. Setting for each request the current timestamp as priority yields LRU; if we set the priority to the negated current timestamp we obtain MRU. Similarly, setting the priority of a page to the last timestamp it faulted we obtain FIFO, and the negated of this value yields LIFO. Assigning for a page the request frequency as priority results in LFU.

Marking. The marking algorithms assign marks to pages and work in phases as follows. A phase begins when all pages in cache are marked and a cache miss occurs. In this case all pages are unmarked, the page in cache predicted to be requested farthest in the future is evicted, and the new page is loaded in cache and marked. For each request to some page \( p \) within a phase, if \( p \) is a cache hit it gets marked and if it’s a cache miss the unmarked page in cache predicted to be requested farthest in the future is evicted, after which \( p \) is loaded in the cache and marked.

\(^6\)For all datasets we considered the full input to compute the parameters for locality of reference in the Max model, as opposed to [2] where they truncated inputs longer than \( 10^7 \) requests; in our experiments the input size ranges from \( 7 \cdot 10^5 \) to \( 5 \cdot 10^8 \), hence the slightly different behavior compared to [2] for the same application.
Figure 2: The offset function-based predicted fault rate \( of = \frac{\lambda_0 + \lambda_u}{\lambda_0 + \lambda_u + \lambda_r} \), the locality of reference in the Max- and Average-model, and the non-locality of reference, together with the actual performance of LRU for the first twelve datasets. The x-axis shows the cache size and the y-axis shows the fault ratio.
OnOPT. The algorithms in this class are based on the layer partition in [22] previously described. They always have a cache configuration identical to LFD if the priority assignment reflects future requests. This implies that they are always in a valid configuration according to the current work function. These algorithms maintain the layer partition and process some page \( p \) by first applying an eviction policy in the case of a cache miss followed by updating the layers, as shown in [11]. The eviction policy is implemented as follows. If \( p \) is in the cache then nothing needs to be done. If \( p \) is not in the cache we distinguish between two cases: \( p \in L_0 \) and \( p \in L_i \) with \( i > 0 \). If \( p \in L_0 \) then the page in cache having the smallest priority is evicted. If \( p \in L_i \) and \( p \) triggers a cache fault, we first identify the layer \( L_j \) with \( j \geq i \) such that the cache contains exactly \( j \) pages in \( L_1 \cup \cdots \cup L_j \), i.e. \( |M \cap (\cup_{l=1}^j L_l)| = j \). The page in cache from \( L_1 \cup \cdots \cup L_j \) having the smallest priority is evicted. This eviction policy ensures that in the case that the priority assignment reflects the future requests, the cache contents of the online algorithm and LFD are identical.

We note that, since implementations are given, each of the three classes can be viewed as a framework which, provided with a priority assignment, results in a paging algorithm. Assuming that the only priority change happens for the current request, algorithms in all three classes support very fast implementations and thus are not prohibitively expensive in practice. Algorithms in the CACHEMIN and MARKING classes can be easily implemented using a dictionary and a priority queue, which take \( O(k) \) space and \( O(\log k) \) time per page request. For the algorithms in the ONOPT class we showed in [11] how to implement them in \( O(m) \) space and \( O(\log m) \) time per request where \( m \) is the size of the pageset. A variant with similar behavior and supporting a faster implementation can be achieved by using the forgiveness mechanism introduced in [5]. The resulted implementation uses \( O(k) \) space and \( O(\log k) \) time per request as well, however the theoretical guarantees are compromised. Nonetheless, experimental results show that the number of cache misses done by the two implementations is virtually identical. However the bounds provided are generic and apply to all algorithms in a given framework, but certain algorithms can be implemented significantly faster, e.g. FIFO takes \( O(1) \) time per request.
### 3.2 Competitive analysis

In this section we give lower and upper bounds on the competitive ratio for deterministic paging algorithms, as a function of the attack rate \( r \). The results are summarized in Table 2.

**Lemma 1** The competitive ratio for any deterministic paging algorithm on an input in \( \mathcal{I}(r) \), for any arbitrary rational \( r \in [1...k] \), is at least \( r \).

**Proof.** Recall that \( \mathcal{I}(r) \) contains all inputs having the attack rate at most \( r \). Consider some arbitrary deterministic algorithm \( A \). To prove the claimed bound we build an input sequence on which \( A \) is guaranteed to perform \( r \) times more cache misses than LFD. We consider a set containing \( k + 1 \) pages, on which we build a subsequence which starts in a cone and ends in a cone. We first use the standard lower bound construction from classical competitive analysis and request \( k \) pages such that for each request \( A \) does a cache miss and \( \lambda_0 = 1 \). We then request as many unrevealed pages as necessary until we end in a cone. Since the only first request is in \( L_0 \) and we end in a cone, for each such subsequence we have \( \lambda_0 = 1 \) and \( \lambda_u = k - 1 \). Also, by construction \( A \) does at least \( k \) cache misses.

We request this subsequence \( n_1 \) times using the same set of \( k + 1 \) pairwise distinct pages, followed by \( n_2 \) requests to pages in \( L_0 \) that were never requested. For such an input, we have \( \lambda_0 = n_1 + n_2 \) and \( \lambda_u = (k - 1)n_1 \), which leads to an attack ratio \( r = \frac{k(n_1 + n_2)}{n_1 + n_2} \). Using the fact that LFD faults only on requests in \( L_0 \), the competitive ratio is at least \( \frac{k(n_1 + n_2)}{n_1 + n_2} = r \). Combining different values for \( n_1 \) and \( n_2 \) we obtain any possible rational value for \( r \in [1...k] \) and the proof concludes.

**Fact 2** Any algorithm is \( 1 \)-competitive on inputs in \( \mathcal{I}(1) \).

**CacheMin; LIFO, MRU, and LFU.** Both LIFO and LFU belong to the CacheMin class, and for both of them the arguments from the standard competitive analysis carry on to our parametrized inputs. For LIFO, after the first \( k \)-pairwise distinct pages we request two new pages \( x \) and \( y \) alternately and infinitely, i.e. the input sequence \( \sigma = p_1, \ldots, p_k, (xy)^\infty \). LIFO does a cache miss on each request while OPT does only 2 cache misses (we exclude the first \( k \) pairwise distinct pages).

We note that we used an input having attack ratio of \( 3/2 \), but it can be easily extended to any value \( r > 1 \). The same argument holds for MRU. For LFU, we request the first \( k \) pairwise distinct items \( n \) times each and then we cyclically request two new pairwise distinct pages \( n - 1 \) times each, i.e. the input is \( \sigma = (p_1, \ldots, p_k)^n(p_{k+1}, p_{k+2})^{n-1} \). Similarly to LIFO and MRU, LFU faults on each page while OPT incurs 2 misses. For infinitely large \( n \) the competitive ratio is unbounded. Similarly to LIFO, the attack rate is \( 3/2 \) but can be extended to any value in \( [1...k] \).

**Marking algorithms.** For the marking algorithms, we first show that they are \( 2r \)-competitive and then we show that there exist priority assignments which are very close to this bound. Although
FWF is not in our MARKING framework, the following result applies to it as well, both for the lower and upper bounds.

**Lemma 2**  The competitive ratio for any marking algorithm on an input in $I(r)$ is at most $\min(2r, k)$; there exist marking algorithms which are at least $\min(2r - 1, k)$-competitive for any value of $r$.

**Proof.** For the upper bound we recall a property of marking algorithms, namely that for a sequence of $k$ pairwise distinct pages there can be at most two cache misses on any given page $p$. We divide the request sequence in consecutive phases which start with a request from $L_0$ and contain all following consecutive requests in the support until the next request in $L_0$. Since by Fact 1 at most $k$ pairwise distinct pages are requested during a phase, a page $p$ requested in this phase causes at most two cache misses. If page $p$ triggers one or two cache misses, it implies that it was requested in this phase either from $L_0$ or from an unrevealed layer, since the phase starts with a request from $L_0$ which unreveals all pages in the support. Mapping the at most two cache misses on $p$ to its request from either $L_0$ or an unrevealed layer leads to the upper bound of $2r$. The upper bound of $k$ comes from classical competitive analysis.

For the lower bound, we consider Mark having MRU as priority assignment, i.e. when a page is requested we assign as priority the negated of the current timestamp and construct inputs which achieve the bounds. We consider three types of inputs which we will combine to show the claimed bounds. For each of them we count the number of cache misses done by Mark (MK) and OPT ($\lambda_0$), and the number of requests $\lambda_u$ to unrevealed pages. Type I input performs a request to an item in $L_0$ and we have $MK = \lambda_0 = 1$ and $\lambda_u = 0$. The type II input is a classical attack starting and ending in a cone with all pages marked and it proceeds as follows. We first request a page in $L_0$ and then request $k - 1$ support pages in reverse order of their last requests so that the MRU assignment faults on each request, which yields $MK = k$. Also, we have $\lambda_u = k - 1$ because the first request to the page in $L_0$ unreveals all pages in the support. Also, we have only one request in $L_0$ meaning $\lambda_0 = 1$. The type III input starts and ends in a cone and Mark has all pages marked. Let $\{p_1, \ldots, p_k\}$ be the pages in the cone, which are also the (marked) pages in the cache of Mark. We request the sequence $(p_{k+1}, p_{k+2}, p_k, p_{k-1}, \ldots, p_3, p_2, p_3, \ldots, p_k, p_{k+2})$, where $p_{k+1}$ and $p_{k+2}$ are new pages. On this input Mark does a cache miss on each request and thus $MK = 2k$. We now analyze the offset function $\omega$. Initially, we have $\omega = (p_1, \ldots, p_k)$ and after the first request to $p_{k+2}$ we have $\omega = (p_1, \ldots, p_{k+1}, p_k, p_3, \ldots, p_k, p_{k+2})$. After the request to $p_2$ we have $\omega = (p_{k+2}, p_k, \ldots, p_3, p_2)$, and all further requests are to revealed pages. We thus have two requests in $L_0$ and $k - 1$ to unrevealed pages in the support, which yields $\lambda_u = k - 1$ and $\lambda_0 = 2$.

We now combine the three types of inputs to obtain the lower bound for any value of $r$. In case $2r - 1 < k$ the input is a sequence of $n_3$ type III inputs followed by $n_1$ type I inputs. We have $\lambda_u = (k - 1)n_3$ and $\lambda_0 = n_1 + 2n_3$, which means the attack rate is $r = \frac{n_1 + (k+1)n_3}{n_1 + 2n_3}$. The number of cache misses done by Mark and OPT is $n_1 + 2kn_3$ and $n_1 + 2n_3$ respectively and we obtain that the competitive ratio is $\frac{n_1 + 2kn_3}{n_1 + 2n_3} = 2r - 1$.

If $2r - 1 \geq k$ we build the input as a sequence of $n_3$ type III inputs followed by $n_2$ type II inputs. We have $\lambda_u = (k - 1)n_2 + (k - 1)n_3$ and $\lambda_0 = n_2 + 2n_3$, which yields an attack rate $r = \frac{kn_2 + (k+1)n_3}{n_2 + 2n_3}$. For the competitive ratio, OPT does $n_2 + 2n_3$ cache misses and Mark faults $kn_2 + 2kn_3$ times, leading to a competitive ratio of $\frac{kn_2 + 2kn_3}{n_2 + 2n_3} = k$. Since by choosing various values for $n_1$, $n_2$, and $n_3$ we obtain arbitrary values of $r$, the bound holds for any $r$. 

23
OnOPT and FIFO. By construction, the algorithms in the OnOPT class never fault on revealed items and are thus $r$-competitive on inputs in $\mathcal{I}(r)$. Since LRU is in OnOPT, it is also $r$-competitive. In what concerns FIFO, we show that it is $r$-competitive in spite of the fact that it is not always in a valid configuration. It is indeed possible to build input sequences for which FIFO faults on revealed page.

Lemma 3. FIFO is $r$-competitive on any input in $\mathcal{I}(r)$.

Proof. Similarly to MARKING algorithms, we split the input in phases where each phase starts with a request in $L_0$ and finishes just before the next request in $L_0$. By Fact 1 each phase consists of at most $k$ pairwise distinct pages. We note that a page can fault at most once during a phase, since $k$ more pairwise distinct pages are required until the same page faults again. Since at the beginning of a phase all pages, except for the request in $L_0$ starting the phase, are unrevealed, this immediately implies that each page in this phase is requested exactly once from $L_0$ or from an unrevealed layer. We thus can charge each cache miss on a page to a request to the same page in $L_0$ or an unrevealed layer. We obtain that overall FIFO does $\lambda_u + \lambda_0$ cache misses, which combined to the $\lambda_0$ done by OPT concludes the proof.

4 An algorithm better than LRU

In this section we first give a priority assignment to be used in the framework of OnOPT algorithms previously introduced, which leads to an algorithm that we denote Recency Duration Mix (RDM). As its name implies, it combines two priority policies, one based on recency and the other on the time-frame that pages spend in support. We then conduct experiments which demonstrate that for most inputs and cache sizes our algorithm outperforms not only LRU, but also two of its variants shown to behave well in practice.

4.1 RDM

We recall that the framework of OnOPT algorithms ensures that regardless of the priority assignment we get an $r$-competitive algorithm which is always in a valid configuration. This gives us the freedom to explore various priority policies. Furthermore, this framework can be implemented efficiently with respect to both space and running time to give it practical value. We use a global counter $t$, which keeps track of the amount of requests to pages in $L_0$ and unrevealed layers. Thus before assigning a priority to the requested page $p$, we increment $t$ only if $p$ is not revealed. We do so because requests to revealed pages trigger only a permutation of the revealed layers. More precisely, only the layer representation of the offset function changes, but not the function itself. Thus, such requests do not provide any new information about the possible states of an optimal solution and consequently should not affect the priority assignment. Also, for each page $p$ in the support we store a value $t_0$ which stores the value of $t$ at the time that $p$ entered the support. More exactly, for any request $p$ from $L_0$ we set $t_0(p) = t$. We describe the two priority assignment strategies that we will later combine into a new priority assignment which we plug into the OnOPT framework to obtain RDM.
Recency. We assign each page upon request the current counter $t$ as priority. It is inspired by LRU in that it assigns for each page $p$ the current counter as priority, but unlike LRU our counter ignores requests to revealed pages.

Duration. A major drawback of LRU is that it performs very bad when repeatedly requesting the same sequence having more than $k$ pages, e.g. repeatedly scanning an array. This priority policy addresses this drawback taking into account the time that a page spent in the support. When requested, each page is assigned as priority the value $t - t_0$. The intuition behind this strategy is that if a page is frequently requested during a period, it remains in OPT's cache during this period and gets a high priority. A particular strength of this strategy is the fact that it adapts to repeatedly requesting the same sequence of more than $k$ pages. After the first iteration all the pages are in the support, at the second iteration the first $k - 1$ requests become revealed and get their priorities increased while the remaining ones are evicted from the support and at their next request they are assigned a new $t_0$ value which gets them low priorities, thus avoiding an LRU-like behavior.

We have empirically determined that assigning priorities according to the duration policy alone outperforms LRU for certain datasets and cache sizes. However, using a linear combination of recency and duration the performance improves significantly. Overall, we have achieved the best results when assigning for each page upon request the value $0.8t + 0.1(t - t_0)$ as priority, and this priority is used in the experimental results.

4.2 RDM on real-world traces

We conduct experimental to compare the performance of RDM against the performance of LRU and two of its variants which were shown to behave better than LRU in practice, namely RLRU [10] and EELRU [25].

RLRU (Retrospective LRU) was proposed in [10], where it was also proven to be better than LRU with respect to the relative worst order ratio. It is a marking-like algorithm which assigns marks based on what OPT would have in cache and evicts unmarked pages using a LRU strategy. Empirical results over various datasets showed RLRU to perform fewer cache misses than LRU, though the differences observed were small (mostly up to 5% improvement). EELRU (Early Eviction LRU) is an adaptive paging algorithm from a less theoretical direction. It simulates a large collection of about 256 parametrized instances of an algorithm which is a mix of LRU and MRU (Most Recently Used). To decide which page to evict EELRU consults the results of these 256 instances for the recent past, and the most promising is simulated on the actual request. If none is promising, it switches to LRU and it is guaranteed by construction that it can never be worse than a factor of three compared to LRU. In [25] it was shown that EELRU achieves good performance compared to LRU in practice, outperforming LRU on many datasets, at times by significant amounts.

For each dataset and cache size, we measure for each of the four algorithms considered the competitive ratio, i.e. the number of cache misses performed normalized by the performance of OPT. In Figures 4 and 5 we give the results for all datasets. The results show that on all datasets and for all cache sizes RLRU has a similar performance to LRU, though it outperforms it consistently by small margins. For EELRU, we note that gnuplot is the only dataset on which it outperforms all other algorithms by large margins. For all the remaining inputs, except for certain cache sizes on the espresso and lindsay datasets, EELRU is at least as good as LRU and RLRU; however, on several datasets (e.g. compress, gcc, grobner) there are cache sizes for which it outperforms LRU by factors
ranging from two to four. In what concerns RDM, it outperforms LRU and RLRU on all datasets and for all cache sizes, except for a narrow range on the gcc dataset. The margins vary among datasets, with improvements by more than a factor of 100% on three datasets (compress, grobner, and go) and more than 10% on most of the remaining datasets. Moreover, it rarely happens that RDM has a competitive ratio of more than two. Finally, we note that, except for gnuplot, RDM outperforms EELRU as well on most cache sizes, in many cases by significant margins.

5 Conclusions

The parametrization using a characterization of the optimal solution leads to more realistic predictions for bounds on the competitive ratio and the fault rate. OnOPT algorithms adapt optimally to the “easiness” of the input. Marking algorithms profit from easy inputs, though not optimally, while algorithms like LIFO or LFU do not profit at all. It is interesting that the good performance of LRU can be partially explained by its property of always being in optimal cache configurations. However this holds for the whole OnOPT class, and this motivates searching for other practical algorithms in this class. We provided an algorithm (RDM) among these which clearly outperforms LRU on the tested inputs. Our algorithm can even compete with improved variants of LRU, such as RLRU and EELRU. It is interesting whether other priority assignments or using adaptiveness like EELRU can further improve the fault rate.
Figure 4: The empirical competitive ratio on various inputs for RDM, LRU, RLRU, and EELRU. The x-axis shows the cache size and the y-axis shows the competitive ratio.
Figure 5: The empirical competitive ratio on various inputs for RDM, LRU, RLRU, and EELRU. The x-axis shows the cache size and the y-axis shows the competitive ratio.