Design of Competitive Paging Algorithms with Good Behaviour in Practice

Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften

vorgelegt beim Fachbereich Informatik und Mathematik
der Johann Wolfgang Goethe Universität
Frankfurt am Main

von
Andrei Laurian Negoescu
aus Bukarest

Frankfurt am Main 2013
D(30)

vom Fachbereich Informatik und Mathematik

der Johann Wolfgang Goethe Universität als Dissertation angenommen.

Dekan: Prof. Dr. Thorsten Theobald

Gutachter:

1. Prof. Dr. Ulrich Meyer
2. Prof. Dr. Georg Schnitger
3. Prof. Dr. Alejandro López-Ortiz

Datum der Disputation: *14.10.2013*

Abstract

Paging is one of the most prominent problems in the field of online algorithms. We have to serve a sequence of page requests using a cache that can hold up to k pages. If the currently requested page is in cache we have a *cache hit*, otherwise we say that a *cache miss* occurs, and the requested page needs to be loaded into the cache. The goal is to minimize the number of cache misses by providing a good page-replacement strategy. This problem is part of memory-management when data is stored in a two-level memory hierarchy, more precisely a small and fast memory (cache) and a slow but large memory (disk). The most important application area is the virtual memory management of operating systems. Accessed pages are either already in the RAM or need to be loaded from the hard disk into the RAM using expensive I/O. The time needed to access the RAM is insignificant compared to an I/O operation which takes several milliseconds.

The traditional evaluation framework for online algorithms is *competitive analysis* where the online algorithm is compared to the optimal offline solution. A shortcoming of competitive analysis consists of its too pessimistic worst-case guarantees. For example LRU has a theoretical competitive ratio of k but in practice this ratio rarely exceeds the value 4. Reducing the gap between theory and practice has been a hot research issue during the last years. More recent evaluation models have been used to prove that LRU is an optimal online algorithm or part of a class of optimal algorithms respectively, which was motivated by the assumption that LRU is one of the best algorithms in practice. Most of the newer models make LRU-friendly assumptions regarding the input, thus not leaving much room for new algorithms. Only few works in the field of online paging have introduced new algorithms which can compete with LRU as regards the small number of cache misses.

In the first part of this thesis we study *strongly competitive* randomized paging algorithms, i.e. algorithms with optimal competitive guarantees. Although the tight bound for the competitive ratio has been known for decades, current algorithms matching this bound are complex and have high running times and memory requirements. We propose the algorithm OnlineMin which processes a page request in $O(\log k / \log \log k)$ time in the worst case. The best previously known solution requires $O(k^2)$ time.

Usually the memory requirement of a paging algorithm is measured by the maximum number of pages that the algorithm keeps track of. Any algorithm stores information about the k pages in the cache. In addition it can also store information about pages not in cache, denoted *bookmarks*. We answer the open question of Bein et al. [9] whether strongly competitive randomized paging algorithms using only $o(k)$ bookmarks exist or not. To do so we modify the PARTITION algorithm of McGeoch and Sleator [60] which has an unbounded bookmark complexity, and obtain PARTITION2 which uses $\Theta(k/\log k)$ bookmarks.

In the second part we extract ideas from theoretical analysis of randomized paging algorithms in order to design deterministic algorithms that perform well in practice. We refine competitive analysis by introducing the *attack rate* parameter r, which ranges between 1 and k. We show that r is a tight bound on the competitive ratio of deterministic algorithms. We give empirical evidence that r is usually much smaller than k and thus r-competitive algorithms have a reasonable performance on real-world traces. By introducing the r-competitive priority-based algorithm class ONOPT we obtain a collection of promising algorithms to beat the LRU-standard. We single out the new algorithm RDM and show that it outperforms LRU and some of its variants on a wide range of real-world traces.

Since RDM is more complex than LRU one may think at first sight that the gain in terms of lowering the number of cache misses is ruined by high runtime for processing pages. We engineer a fast implementation of RDM, and compare it to LRU and the very fast FIFO algorithm in an overall evaluation scheme, where we measure the runtime of the algorithms and add penalties for each cache miss. Experimental results show that for realistic penalties RDM still outperforms these two algorithms even if we grant the competitors an idealistic runtime of 0.

Acknowledgements

After years of work I have managed to polish the last details of my Ph.D. Thesis. In achieving this I was not alone. A lot of people supported me in this engaging, yet difficult endeavour, and I would like to take this opportunity to show my gratitude to all of them.

First and foremost thank you, Ulrich Meyer, my Doktorvater, for being an excellent professor, and for the freedom you gave me in choosing the topic of my liking. As a result, I was able to pursue a theme that is both dear and appealing to me, which made the research process more interesting and engaging.

Gabriel Moruz, thank you for the brainstorming sessions and for the research done together, and also for your input in the studies we have carried out over the years.

This thesis was co-funded by MADALGO (Center for Massive Data Algorithmics) and DFG (Deutsche Forschungsgemeinschaft), and I would like to thank both organisations for their support.

I would also like to thank my other co-authors: Annamária Kovács, Gerth Stølting Brodal, Volker Weichert, Christian Neumann for a great collaboration and bright ideas. My appreciations also go to my colleagues: Matthias Poloczek, Bert Besser, Georg Schnitger, Andreas Beckmann, Claudia Schwarz, Claudia Heinemann, David Sabel and Conrad Rau.

I am grateful to Scott F.H. Kaplan (Department of Computer and Science at Amherst College), Alejandro López-Ortiz and Reza Dorrigiv (University of Waterloo) for providing us with experimental data.

I also owe a debt of gratitude to Edgardo Deza, for his insights on subjects relevant to my papers, and to David Veith, my colleague who made the office a work-fit environment and who has reviewed my thesis, providing me with a very valuable feedback.

My thanks also go to my family and my girlfriend Anda for always believing in me, for their unconditional love, understanding and support in all the decisions I have made.

Andrei Negoescu,
Frankfurt, October 28, 2013.

Contents

Chapter 1

Introduction

This thesis deals with theoretical and practical aspects of the paging problem. Paging is part of the memory management in systems with a hierarchy of memory levels, where the faster the memory the more expensive it is. Paging strategies manage the arrangement of accessed data between two memory levels. The small and fast memory is denoted *cache* and the slow and large one is called *disk*. When data is accessed it is desirable to be contained in the cache in order to avoid time penalties for disk access. Data is considered to be organized in chunks of equal size, called *pages* and we denote by k the number of pages that fit into the cache.

The most prominent application area is the virtual memory management of modern operating systems [18, 24, 59, 63], where the cache corresponds to the limited-size RAM (Random-Access Memory). When the needed memory space exceeds the size of available RAM a part of the data is stored on the much slower hard disk. When a page is accessed and it is in the cache we have a *cache hit* and nothing has to be done. Otherwise we say that a *cache miss* occurs and the page has to be loaded into the cache which involves time-consuming I/O operations. Upon a cache miss a paging algorithm decides which page to replace from the cache. Other examples where paging algorithms are used to manage the same data splitting between RAM and hard disk are database systems [22, 55, 58, 62], web caching [19, 56, 64] and tools for external memory algorithms like the C++ library STXXL [25].

Competitive Analysis. The major difficulty regarding paging algorithms is given by the fact that usually little is known about future data access patterns. Thus paging is one of the most studied problems in the field of online algorithms, namely algorithms which have to make irrevocable decisions without knowing the whole input in advance. The first part of this thesis focuses on theoretical analysis of randomized paging algorithms. We use the classical quality measure for online algorithms, namely the *competitive ratio* [60]. A deterministic online algorithm A is c-competitive if for any input sequence it holds that

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

where $cost(A)$ and $cost(OPT)$ denote the cost of A and the optimal offline cost respectively, and b is a constant. In the case of randomized online algorithms the expected number of misses is taken into consideration. An online algorithm is said to be *strongly competitive* if it has the best possible competitive ratio.

For deterministic algorithms the lower bound on the competitive ratio is k [60]. Three of the most prominent k-competitive paging algorithms are LRU (Least Recently Used), FIFO (First In First Out) and FWF (Flush When Full) [60]. Fiat et al. [32] proved that randomized algorithms cannot perform better than H_k-competitive, where $H_k = \sum_{i=1}^{k} 1/i$ is the k-th harmonic number. In the same work the randomized algorithm MARK was proposed and proven to be $2H_k$-competitive. Although not optimal, MARK has the advantage of being a simple and fast algorithm. Later, Achlioptas et al. [1] determined the exact competitive ratio of $2H_k - 1$ for MARK.

The first strongly competitive randomized algorithm PARTITION was proposed by McGeoch and Slater [48]. The memory requirement and runtime complexity of PARTITION are not bounded by the cache size k. Although the cache miss performance is the primary issue in competitive analysis, memory requirement and runtime also play an important role. The first strongly competitive algorithm improving efficiency was EQUITABLE [1] which can be implemented in $O(k^2)$ time per request and $O(k^2 \log k)$ memory. Borodin and El-Yaniv [14] stated the open question if H_k-competitive algorithms exist with only $O(k)$ memory. Bein et al. [10] show that $O(k)$ memory is possible by providing the algorithm EQUITABLE2, a variant of EQUITABLE which uses only $2k$ bookmarks, i.e. pages not in cache that the algorithm keeps track of. In the same work it was conjectured that $o(k)$ bookmarks are possible. Although EQUITABLE2 uses only $O(k)$ memory it still requires $O(k^2)$ time per request.

Contributions I. In this thesis we propose the strongly competitive randomized algorithm ONLINEMIN which processes a page request in $O(\log k)$ time [20] while preserving the $O(k)$ memory requirement of EQUITABLE2. We further improve the runtime to $O(\log k / \log \log k)$ [21] by exploiting the power of the RAM model. We refine the analysis of EQUITABLE2 and show that it can be implemented using only approximately $0.62k$ bookmarks and show that EQUITABLE2 cannot achieve $o(k)$ bookmarks. Instead we present a variant of PARTITION, which we denote PARTITION2, and show that PARTITION2 requires $\Theta(k/\log k)$ bookmarks, thus proving the $o(k)$ bookmark conjecture [10].

Theory vs. Practice. The second part of this thesis deals with the gap between theory and practice of competitive analysis which was often criticized for its too pessimistic quality guarantees for deterministic paging algorithms. Young [67] investigated the *empirical competitive ratio*, i.e. the ratio between the cost of the online algorithm and the optimal cost on real-world inputs. The empirical com-

petitive ratio of LRU is a small constant, almost independent of the cache size, whereas its competitive ratio is k. Multiplicative factors in the hundreds have been observed between the theoretical worst-case bound and the experimental results. Another criticized issue is the fact that competitive analysis does not separate algorithm LRU from FWF or FIFO, although in practice it is well-known that LRU performs substantially better than the other two algorithms [14].

Recent theoretical research focused on new methods for analyzing online algorithms in general and paging in particular. One line of research is concerned with restricted versions of competitive analysis, such as the *diffuse adversary* [44] or *loose competitiveness* [68]. Other approaches consider comparing algorithms directly, without relating them to an optimal offline algorithm. Relevant examples include the *Max/Max ratio* [13], the *random order ratio* [42], the *relative worst order ratio* [17], and *bijective analysis* and *average analysis* [6]. A detailed survey of alternative performance measures can be found in [29]. Many of these approaches are concerned with separating existing paging algorithms in order to explain the differences noticed in practice. In particular, several approaches (e.g. diffuse adversary, bijective analysis combined with locality of reference [7]) 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) [17] and FARL (Farthest-To-Last-Request) [14, 33] which were designed according to the relative worst order ratio and access graph model [15, 31, 38] respectively.

Contributions II. We propose a refinement of competitive analysis for the paging problem by introducing the attack rate r [51]. The attack rate is an input parameter which ranges between 1 and k and roughly corresponds to the extent of uncertainty regarding which pages the optimal offline algorithm has in cache. In contrast to many other approaches we do not use the LRU-friendly assumption of locality of reference. We prove that r is a lower and upper bound on the competitive ratio of deterministic algorithms. This bound is matched by LRU and FIFO but not by FWF. Experiments on real-world traces reveal that the value of r is in many settings much smaller than k, thus the gap between the observed empirical competitive ratio and r is much smaller for LRU and FIFO compared to classical competitive analysis. Our parametrization also allows us to give an upper bound on the *fault rate* of r-competitive algorithms. We show experimentally that our parametrized bound explains better the low fault rate of LRU than the fault rate predictions using parametrizations based on locality of reference.

Separating known algorithms and/or predicting their performance is one aspect of a cost model which is (at least partially) addressed successfully by many new approaches. Another desirable property is to find new algorithms with a good behaviour in practice. We define the OnOPT algorithm class which con-

tains only r-competitive algorithms, including LRU. From this class we single out the new algorithm RDM and perform experiments on real-world traces. The experimental results show that RDM can outperform LRU and other algorithms thus contradicting newer theoretical models that single out LRU as the best paging algorithm. We note that the attack rate model and the OnOPT class are inspired by the research of strongly competitive paging algorithms. This shows that insights from classical competitive analysis can help design algorithms performing few cache misses on real world inputs.

The main goal of a practical paging algorithm is to minimize the cost in terms of cache misses since cache misses can cause several milliseconds time penalties in the program execution which is up to ten thousand times more than the access time for the main memory. Yet the running time of paging algorithms cannot be ignored. We engineer an efficient implementation of RDM [54]. Further we provide an overall evaluation which simulates page requests from traces and determines a so-called *total cost*. The total cost consists of the runtime of the paging algorithm and a typical additional penalty of 9ms [63, Chapter 1.3.3] for each cache miss. The experimental results show that RDM can outperform LRU and FIFO as regards the total cost.

Publications. The results in this cumulative thesis are based on four peer-reviewed conference publications, two peer-reviewed (invited) journal publications, and one technical report. I was the main contributor of all six publications.

[20] G. S. Brodal, G. Moruz, and A. Negoescu. Onlinemin: A fast strongly competitive randomized paging algorithm. In *Proc. 9th International Workshop on Approximation and Online Algorithms:, WAOA 2011, Revised Selected Papers*, pages 164–175. Springer, 2012

I contributed with the design and and the main part of the competitive analysis of algorithm OnlineMin. The proposed data structures are a contribution which belongs to all authors in the same proportion.

[21] G. S. Brodal, G. Moruz, and A. Negoescu. Onlinemin: A fast strongly competitive randomized paging algorithm. *Journal Theory of Computing Systems, Special issue of the 9th Workshop on Approximation and Online Algorithms*, 2013

My share in this paper is the design and the main part of the competitive analysis of algorithm OnlineMin, including the improvements over the conference version [20]. The pointer-based data structures are a joint contribution of all the authors in the same amount. The input regarding the data structures in the RAM model mainly belongs to my co-author Gerth Stølting Brodal.

[51] G. Moruz and A. Negoescu. Outperforming LRU via competitive analysis on paramterized inputs for paging. In *Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1669–1680, 2012

My contribution contains the attack rate model, the OnOPT class and the RDM algorithm. The experimental evaluation is a joint contribution which belongs to both authors in the same proportion.

[54] G. Moruz, A. Negoescu, C. Neumann, and V. Weichert. Engineering efficient paging algorithms. In *Proc. 11th International Symposium on Experimental Algorithms*, pages 320–331, 2012

The compressed layer representation, the improved implementation of OnlineMin and the overall evaluation method are the part I brought to this paper. The experimental evaluation was performed together with the other authors of the paper: Christian Neumann, Gabriel Moruz, and Volker Weichert.

[53] G. Moruz, A. Negoescu, C. Neumann, and V. Weichert. Engineering efficient paging algorithms. *Journal of Experimental Algorithmics, Special issue of SEA 2012 (to appear)*

The part I bring in this paper is the compressed layer representation, the improved implementation of OnlineMin, the overall evaluation method, and the equivalence proof of the layer representation, which was omitted in the conference paper [54] due to space limitations. The contribution of the more detailed experimental results (compared to the conference paper) belongs to all authors in the same proportion.

[50] G. Moruz and A. Negoescu. Improved space bounds for strongly competitive randomized paging algorithms. In *Proc. 40th International Colloquium on Automata, Languages, and Programming, ICALP 2013 (to appear)*

My share in this work is the design and analysis of algorithm Partition2 (the main result of this paper) and the lower bound on algorithms using deterministic forgiveness methods. The tighter analysis of Equitable2 belongs to both authors in the same proportion.

[52] G. Moruz and A. Negoescu. Improved space bounds for strongly competitive randomized paging algorithms. Technical report, Goethe-Universität Frankfurt am Main, 2013

This technical report is the full version of paper [50]. It contains the same results as the conference paper. Additionally it contains proofs omitted due

to space limitation in the conference paper. My contribution is the design and analysis of algorithm PARTITION2 (the main result of this paper) and the lower bound on algorithms using deterministic forgiveness methods. The tighter analysis of EQUITABLE2 belongs to both authors in the same proportion.

Beside my research on the classical paging problem I have also studied two other topics, which are not covered by this thesis. The first is α-paging, a variation of paging tailored for flash memory devices. The second is about lower bounds on the average-case complexity for classic single-source shortest-paths algorithms. The two resulted peer-reviewed publications are listed next.

[45] A. Kovács, U. Meyer, G. Moruz, and A. Negoescu. Online paging for flash memory devices. In *Proc. 20th International Symposium on Algorithms and Computation,, ISAAC 2009*, pages 352–361, 2009

[49] U. Meyer, A. Negoescu, and V. Weichert. New bounds for old algorithms: On the average-case behavior of classic single-source shortest-paths approaches. In *Proc. First International ICST Conference on Theory and Practice of Algorithms in (Computer) Systems, TAPAS 2011*, pages 217–228, 2011

Outline. The rest of this thesis is structured as follows: In Chapter 2 we introduce relevant concepts of online algorithms in general and the paging problem in particular. This covers a brief description of the most popular paging algorithms in the theoretical framework of competitive analysis. As a preliminary step for our experimental results, we also introduce three known models for analyzing the fault rate. This chapter ends with a survey on the concept of the so-called *valid configurations*, a crucial ingredient for our results.

In Chapter 3 the results of our research are presented. We begin with theoretical improvements of runtime [20, 21] and memory requirement [50] for strongly competitive paging algorithms. Next we introduce a refinement of competitive analysis leading to a new algorithm named RDM which outperforms LRU on real-world traces [51]. At the end of the chapter we engineer fast implementations for RDM [53, 54].

Chapters 4-7 contain our published papers.

Chapter 2

Definitions and Basics

2.1 Online Algorithms

Optimization Problems. A well-known optimization problem we deal with on a daily basis when driving our car, is to find the shortest path from the starting point to the destination. There are many feasible solutions, namely each path that may take us to our destination, and each solution has a cost, in this case - the distance. We are interested in the solution with the minimal distance.

Formally an optimization problem consists of a set $\mathcal{I}$ of possible inputs; for every input I we have a set $F(I)$ of feasible solutions and a quality measure function $f : F(I) \to \mathbb{R}$. Depending on the problem we are dealing with we have either to minimize or to maximize the value of f over the set of all feasible solutions. Thus we call f a *cost* or a *profit* function.

Usually we are provided with the complete input of finite size and the task is to find good solutions for the given instance. In the case of the shortest path problem our input is a weighted graph G which models the street map. We can find an optimal solution with Dijkstras algorithm [28] in a number of computation steps, which is polynomial in the input size. There are also harder problems, which we cannot solve in polynomial time. One prominent example is Clique, where provided with an unweighted graph G we have to find a subgraph G' such that each node in G' is connected to all other nodes in the subgraph. The profit function is the number of nodes in G' and has to be maximized. This kind of problems can also be solved *optimally* although not in reasonable time.

The basic challenge in design and analysis is finding algorithms which are efficient in terms of running time and space consumption. If we cannot compute the optimum in polynomial time we are seeking for good approximations.

Online Optimization Problems. Consider that a person goes skiing, where the duration d of the trip is not known in advance, due to weather conditions for example. There exists the possibility to rent skis for 1 price unit per day or buy them for 10 price units. Each day our vacationist has to decide if he rents the

skis or buys them. If he buys the skis no other expenses need to be done in the remaining days. The goal is to minimize the amount of money spent during the holiday. If $d \geq 10$ he should buy the skis on the first day; otherwise renting (each day) is more profitable. It is easy to see that an optimal decision strategy does not exist since we do not know d in advance. The described problem is known as the *ski rental problem* [40].

We call this kind of problems *online problems*. They differ from offline problems by the property that irrevocable decisions have to be made with only partial knowledge of the input. More precisely an online algorithm A has to process an input sequence $\sigma = (\sigma_1, \ldots, \sigma_n)$, where at each step i the element σ_i is presented. Algorithm A has to base its decisions in step i only on the first i elements $\sigma_1, \ldots, \sigma_i$. Due to the lack of information about the *future* an online algorithm cannot solve the given problem optimally for every request sequence, even without any constraints on computation time or memory space. In the following we assume online problems to be minimization problems, i.e. f is a cost function and the goal is to minimize the value of f over all feasible solutions. We denote by $A(\sigma)$ the cost of A on input σ.

Optimal Offline Algorithm (OPT). Offline algorithms for online optimization problems can base their decisions on the whole request sequence, including the items yet to be presented. Although not implementable for an online problem in practice, they allow for a comparison of the online solution with the best possible solution. The optimal cost for a given input σ is defined as:

$$OPT(\sigma) = \min_A \{A(\sigma)\},$$

where A is an arbitrary algorithm. For the ski rental problem, assume that the input σ is a binary sequence, where the first 1 indicates the ending of the vacation. At the beginning, the optimal offline algorithm identifies the index d such that $\sigma_1 = \sigma_2 = \cdots = \sigma_d = 0$ and $\sigma_{d+1} = 1$. If $d \geq 10$ it decides to buy on the first day; otherwise it rents skis for the entire period. Note that not all online problems have such simple optimal offline solutions.

Competitive Analysis. Let A be a deterministic online algorithm. How can we decide whether A is a good algorithm or not? Competitive analysis is doing this by comparing A to the optimal offline algorithm.

Definition 2.1 *Online algorithm A is c-competitive if there exists a constant b such that:*

$$\forall \sigma : A(\sigma) \leq c \cdot OPT(\sigma) + b.$$

In the case that A is a randomized algorithm, $A(\sigma)$ stands for the *expected cost* of A on σ. As an alternative to A is c-competitive we can say that A has

a *competitive ratio* of c. Algorithm A is said to be *strongly competitive* if its competitive ratio is minimal, and thus the best possible for the given online problem.

Competitive analysis can be seen as a two-player game between the *online player* and the *adversary*. The adversary creates an input which maximizes the ratio between the online and the optimal offline cost. When creating a worst-case input the adversary knows exactly the decisions of a deterministic algorithm. In the case of randomized algorithms it knows only a probability distribution over all possible decisions.

For the ski rental problem consider the following strategy: rent skis for the first 9 days and buy them at the beginning of the 10th day. What value d for the duration would be chosen by the adversary? For sure it would not choose $d \leq 9$ since the online strategy is optimal in this case. The value $d = 10$ is a good one since the online player pays 19 whereas the optimal solution is 10. A higher value for d does not increase any of the two costs. We conclude that the worst-case ratio is 1.9. This online algorithm is known as the *break-even algorithm* [41] and is strongly competitive.

2.2 The Paging Problem

Paging is a prominent, well studied problem in the area of online algorithms. It is a special case of the k-server problem [46], one of the most challenging problems in the field of online algorithms.

Problem Definition. We are provided with a slow but large memory, the so-called *disk*, and a fast but small memory, the so-called *cache*. Data is assumed to be organized in blocks of equal size which we denote *pages*. We assume pages to be represented by numbers $1, \ldots, m$. The size of the cache is k. The input consists of a request sequence $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$, where $\sigma_i \in \{1, \ldots, m\}$.

An online algorithm A for the paging problem has to process the requests online and it is allowed to update its cache content C_A upon each request. We have that C_A is a subset of $\{1, \ldots, m\}$ under the constraint $|C_A| \leq k$. Upon request σ_i we distinguish two possibilities. If $\sigma_i \in C_A$ we say that a *cache hit* occurs and we have cost 0 for processing σ_i. Otherwise we get a *cache miss*, which is also denoted *page fault* and the cost is 1. Algorithm A needs to load σ_i into the cache, where a *page replacement* has to be performed if the cache contains already k pages. More precisely, A has to determine a page $q \in C_A$ and perform the update $C_A = C_A \setminus \{q\} \cup \{\sigma_i\}$. The decision of A is based only on the input seen so far, namely $\sigma_1, \sigma_2, \ldots, \sigma_i$.

Bookmarks. Every paging algorithm needs to keep track of all k pages in its cache. Some more sophisticated algorithms also need to store information about

a part of the pages on the disk. The pieces of information about those pages are called *bookmarks*. If an algorithm does not use bookmarks at all it is denoted *trackless*.

The Optimal Offline Solution. An optimal offline algorithm for paging is known for decades, namely LFD (Longest-Forward-Distance) [12]. Upon a page replacement it evicts the page which is requested farthest in the future. Consider the request σ_i which triggers a cache miss. Assign each page p in the cache the the value $v[p] = j$, where j is the smallest index $j > i$ such that $\sigma_j = p$. In the case that p is never requested again, assign $v[p] = \infty$. Determine q as the page with the highest v-value in the cache and replace it by σ_i.

In the following we refer to LFD as OPT. Although not implementable in real-world scenarios due to its offline character it is important to compare the performance of online algorithms to the best possible result.

Competitive Ratio Lower Bounds. Sleator and Tarjan [60] showed a lower bound of k on the competitive ratio for deterministic algorithms, where k is the cache size. A proof for this lower bound is summarized in the following.

Consider an arbitrary online algorithm A. The adversary requests at the beginning k distinct pages $p_1, p_2, \ldots, p_k$. We assume these pages to be contained in both the cache of OPT and A. Then the adversary requests a page p_{k+1} and A has to evict some page p_{i_1} from the cache. Next he requests p_{i_1} which has to be brought back by A and another page currently in the cache p_{i_2} has to be evicted. This continues until the request of $p_{i_{k-1}}$. Each of the k requests $(p_{k+1}, p_{i_1}, p_{i_2}, \ldots, p_{i_{k-1}})$ triggers one cache miss for A. In this sequence at least one page p_j from the initial cache content $\{p_1, \ldots, p_k\}$ is not requested. Thus upon the first request of p_{k+1} the optimal algorithm can replace page p_j and has a total cost of 1 for this sequence. We request pages from $\{p_1, \ldots, p_{k+1}\} \setminus \{p_j\}$ until A evicts p_j and thus A has the same cache content as OPT. Now we can repeat our attack scheme which incurs cost at least k for A and cost 1 for OPT. We conclude that no deterministic online algorithm can achieve a competitive ratio smaller than k. Algorithms like LRU and FIFO are k-competitive and thus strongly competitive.

When considering *randomized algorithms* life becomes harder for the adversary. When creating a worst-case input, it does not know the precise cache content of the online algorithm A. In the previous lower bound proof the adversary knew exactly the cache content of A and could always request a page not in the cache of A. If A uses randomization the adversary can only determine the probability of a page not being in the cache of A.

Fiat et al. [32] showed that randomized algorithms cannot perform better than H_k-competitive, where H_k is the k-th harmonic number: $1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k}$. A simpler proof in [32] using Yao's principle [66](a powerful technique for proving

lower bounds for randomized algorithms) is given in [14]. It is an adaptation of the lower bound proof for uniform task systems [16].

Empirical Competitive Ratio. Competitive ratio guarantees are worst-case guarantees, which do not necessarily reflect the behaviour of online algorithms in practice and especially not for paging. Empirical competitive ratio is the ratio between the cache misses of an online algorithm divided by the cache misses of the optimal solution on real-world traces. Usually this ratio is plotted on a chart where k varies from 2 to the amount of all distinct pages requested in the trace. LRU is known to have a very small empirical competitive ratio.

Fault Rate. The fault rate is the measure often used in practice in order to evaluate the performance of paging algorithms. Let A be an arbitrary paging algorithm and σ a request sequence of length $|\sigma|$. If $A(\sigma)$ is the number of cache misses of A when processing σ, we define the fault rate of A on σ by the quotient

$$F_A(\sigma) = A(\sigma)/|\sigma|,$$

which ranges between 0 and 1. Reasonable theoretical worst case results for the fault rate are not possible if we do not restrict the class of possible inputs, as every algorithm can be forced to fault on each request by always requesting a page that has never been requested before.

2.2.1 Paging Algorithms

In this section we give a brief description of the most popular paging algorithms. All the following deterministic algorithms are trackless.

LRU. The strategy Least Recently Used maintains the cache content sorted by the timestamp when the pages were requested last time. The more recently a page was requested the higher is its priority to be kept in the cache. Upon a page replacement LRU evicts the least recently accessed page. LRU is k-competitive and thus strongly competitive. Its empirical competitive ratio is usually below 4 and thus it is often regarded as the algorithm with the best performance in practice.

FIFO. The paging algorithm First In First Out evicts upon a page replacement the page which first entered the cache. Although FIFO is also k-competitive it performs worse than LRU in practice as regards the number of cache misses. Nonetheless it has the big advantage that an implementation of this algorithm needs only a simple FIFO Queue, which handles each request in $O(1)$ time. FIFO is one of the fastest paging algorithms.

LIFO, MRU, LFU. Last In First Out evicts upon a cache miss the page which was loaded last in the cache and Most Recently Used replaces the most recently accessed page. These two strategies can be seen as the counterpart of FIFO and LRU. Both have in common that their competitive ratio is not bounded by the cache size k. To see this consider the initial sequence $p_1, \ldots, p_k$. We continue requesting p_{k+1}, p_k arbitrarily often. The optimal solution can evict p_1 and the whole sequence has cost 1 whereas LIFO and MRU have an unbounded cost. On real-world traces both strategies have a disastrous performance. Least Frequently Used has a counter for each page in cache and upon each request of page p it increases the counter of p. Upon a page replacement the page with the lowest value of the counter is chosen to be evicted. The competitive ratio of LFU is also unbounded.

Random. This is the simplest randomized paging algorithm. Whenever a page needs to be evicted, we choose one uniformly at random from the cache. This strategy yields a competitive ratio of k [57]. Although it uses randomization it does not perform better than good deterministic algorithms. RANDOM is a trackless algorithm.

Mark, FWF. A simple randomized paging algorithm which almost matches the lower bound of H_k on the competitiveness of paging algorithms is MARK. By using one bit for each page in the cache, it divides pages into *marked* and *unmarked*. Whenever a page is requested it gets marked. Upon a page replacement, if not all k pages are marked, MARK evicts one unmarked page uniformly at random and the actually accessed page gets marked. In the case where all k pages are marked, they are first unmarked, one is chosen uniformly at random to be evicted and the newly requested page gets marked. Since MARK uses information only about pages in the cache it is also trackless.

A first analysis showed that MARK is $2H_k$-competitive [32]. The analysis was improved later [1] and a tight bound of $2H_k - 1$ was proven. Due to its simplicity, the question arises if another probability distribution on the unmarked pages can improve the competitive ratio. Chrobak et al. [23] addressed this question by proving that, for any constant $\varepsilon > 0$, no randomized marking algorithm can perform better than $(2 - \varepsilon)H_k$-competitive.

Deterministic marking algorithms evict upon a page replacement one unmarked page using a deterministic rule. Algorithms in this class are k-competitive. One special algorithm is FWF (Flush When Full), which evicts all k pages upon a page replacement. It is often used to point out the limits of competitive analysis, since it reaches the optimal bound of k but its behaviour in practice is rather poor compared to LRU and FIFO.

Partition, Equitable. The randomized algorithm PARTITION [48] was the first one that matched the lower bound of H_k. This algorithm is much more complicated than the MARK algorithm. Both memory and runtime are bounded only by the input size n and not by the cache size k. This was improved by Achlioptas et al. [1] who presented the EQUITABLE algorithm which uses $O(k^2 \log k)$ bookmarks. Bein et al. [11] further improved the memory requirement, where a variant of EQUITABLE, denoted EQUITABLE2, uses only $2k$ bookmarks. Both variants of EQUITABLE process a page request in $O(k^2)$ time.

In order to achieve the optimal bound of H_k, PARTITION and EQUITABLE maintain a probability distribution over all *valid* configurations. Valid configurations are all possible cache contents of the optimal offline algorithm after having processed the requests so far. Especially if a page is in none of the valid configurations it has 0 probability to be in their cache. Keeping track of all valid configurations is the main bottleneck for runtime and space of PARTITION and EQUITABLE.

Unlike the strongly competitive randomized algorithms we can easily force MARK to keep a page in the cache with non-zero probability although this page is not contained in any valid configuration. To see this let us look at the following request sequence:

$$(p_1, p_2, \ldots, p_k, p_{k+1}, p_{k+2}, p_1, p_2, \ldots, p_{k-1}).$$

Upon the request of p_{k+1} the optimal offline algorithm evicts p_k and upon the request of p_{k+2} it evicts p_{k+1}. After processing the given sequence we know for sure that the cache of OPT contains $p_1, p_2, \ldots, p_{k-1}$ and p_{k+2}. Thus we have only one valid configuration. Right before the last request MARK has the k marked pages $p_{k+1}, p_{k+2}, p_1, p_2, \ldots, p_{k-2}$ in its cache. After the last request the page p_{k+1} has a probability of $1 - 1/k$ to reside in the cache of MARK.

2.2.2 Fault Rate Analysis

As mentioned before, the *fault rate* is often used in order to evaluate the performance of paging algorithms on real-world traces. Reasonable theoretical worst-case results for the fault rate are not possible if we do not restrict the class of possible inputs. Observations from practice tell us that request sequences have regularities which make them easy to handle. The most important behaviour for the paging problem is called *locality of reference*, which tells us that there is a small set of pages that is requested very often within a certain time interval [24,63]. This partially explains the very good performance of LRU in practice. In the following we present three approaches for modelling locality of reference and analyzing the fault rate under this parametrization. The *Max-Model* and the *Average-Model* were introduced by Albers et al. [2] and a measure for the *non-locality* of reference was proposed by Dorrigiv et al. [30]. Each of them pro-

	Max-Model	Average-Model
Online	$\geq \frac{k-1}{f^{-1}(k+1)-2}$	$\geq \frac{f(k+1)-1}{k}$
LRU	$= \frac{k-1}{f^{-1}(k+1)-2}$	$= \frac{f(k+1)-1}{k}$
FIFO	$\geq \frac{k-1/k}{f^{-1}(k+1)-1}, \leq \frac{k}{f^{-1}(k+1)-1}$	$= \frac{f(k+1)-1}{k}$
Marking	$\leq \frac{k}{f^{-1}(k+1)-1}$	$\leq \frac{4}{3}\frac{f(k)}{k}$

Table 2.1: Fault rates of online algorithms [2].

vides upper bounds on the fault rate which is based on one easy-to-measure input parameter.

Locality of Reference [2]. The underlying characterization of locality of reference is given by a function f which assigns to a window size n the number of distinct pages requested. The function f is assumed to be increasing and concave.

First we summarize the Max-Model. In the Max-Model an input σ is said to be consistent with f if during every subsequence of n requests at most $f(n)$ distinct pages are accessed. The inverse function f^{-1} is defined as follows:

$$f^{-1}(m) = \min\{n \in \mathbb{N} | f(n) \geq m\}.$$

The key parameter in the theoretical results on the fault rate (see Table 2.1) is the value $f^{-1}(k+1)$, which is the minimal window size for $k+1$ distinct page requests. In the Max-Model it was shown that LRU achieves the best possible worst-case guarantee for the fault rate of online algorithms. FIFO and FWF were proven to have worse guarantees than LRU, although the difference is very small.

In the Average-Model an input is consistent with f if the average number of distinct pages in a window of n requests is at most $f(n)$. The crucial parameter for the results (see Table 2.1) is $f(k+1)$, the average number of distinct pages in frames of size $k+1$. Although the Average-Model does not separate LRU from FIFO, the parametrized guarantees are closer to the observed fault rate than in the case of the Max-Model.

Non-Locality of Reference [30]. This parametrization of the input sequences reflects the non-locality of reference. For a given input σ the non-locality parameter $\bar{\lambda}(\sigma)$ is defined as the average of the number of distinct pages between two consecutive requests to the same page. For the i-th requested page $\sigma[i]$ let $d_\sigma[i]$ be $k+1$ if page $\sigma[i]$ was requested for the first time. Otherwise let $d_\sigma[i]$ be the number of distinct pages requested since the last request to $\sigma[i]$.

$$\bar{\lambda}(\sigma) = \frac{1}{|\sigma|} \sum_{1 \leq i \leq |\sigma|} d_\sigma[i]$$

	online alg.	LRU	FIFO	FWF	LFU	LIFO
lower bound	$\frac{\bar{\lambda}}{k+1}$	$\frac{\bar{\lambda}}{k+1}$	$\frac{\bar{\lambda}}{k+1}$	$\frac{2\bar{\lambda}}{k+3}$	$\frac{2\bar{\lambda}}{k+3}$	$\frac{\bar{\lambda}}{2}$
upper bound	$\frac{\bar{\lambda}}{2}$	$\frac{\bar{\lambda}}{k+1}$	$\frac{\bar{\lambda}}{k+1}$	$\frac{2\bar{\lambda}}{k+3}$	-	$\frac{\bar{\lambda}}{2}$

Table 2.2: Upper and lower bounds on the fault rates of deterministic online algorithms [30].

One strength of this model is the simple analysis of standard algorithms. LRU and FIFO both achieve the best possible guarantee on the fault rate. Although it does not separate LRU from FIFO, it manages to clearly separate LRU from FWF and LFU, namely by a multiplicative factor of almost 2. Algorithm LIFO which performs disastrously on inputs with high locality of reference has the worst possible guarantee. A summary of the theoretical results is given in Table 2.2.

2.2.3 Cache Configurations of OPT

Given the request sequence σ seen so far, in an online setting it is of interest to know the actual cache content C_{OPT} of an optimal offline solution, which processes the whole input with minimal cost. Although in an online scenario C_{OPT} is not known since it depends also on the future request sequence τ, we are provided with partial information (from the processed sequence 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 the cache content C of some algorithm A containing k pages is a *valid configuration* iff σ has been processed by A with minimal cost. It was shown that the set of all valid configurations, in the following denoted $\mathcal{V}$, contains all possible configurations C_{OPT} [44] (Lemma 2.3) . This means that an optimal solution for the whole sequence $\sigma\tau$ has minimal cost after processing σ. As we will see in Lemma 2.4, $\mathcal{V}$ contains exactly the set of all possible configurations of the optimal offline algorithm. We give a brief overview of existing representations of $\mathcal{V}$. We start with one of the first characterizations used in the context of work functions which can be seen as a mathematical utility for algorithm analysis. Then we present some changes which lead to representations more favorable for algorithm design and implementation.

Set Inclusion. One of the first representations of $\mathcal{V}$, denoted in the following *set inclusion*, was provided by Koutsoupias and Papadimitriou [44]. It uses a sequence of non-disjoint sets containing pages which is updated after each request. It was used to prove that LRU is optimal in the *diffuse adversary* model.

Lemma 2.1 ([44], Lemma 2.3) *For each request sequence σ there is an increasing sequence of sets $S = (S_1 \subsetneq S_2 \subsetneq \cdots \subsetneq S_k)$, with $S_1 = \{p\}$ the most recent request, such that $\mathcal{V}$ is precisely*

$$\{C : |C \cap S_j| \geq j \text{ for all } j \leq k\}.$$

The sets are initially $S_j = \{p_1, \ldots, p_j\}$ given that $p_1, \ldots, p_k$ is an arbitrary order of the first k pairwise distinct requested pages. Note that this initial set representation is *not unique* and each of the $k!$ permutations of $p_1, \ldots, p_k$ describes the same set of valid configurations. If S is the representation of $\mathcal{V}$ for input σ, let S^p denote the representation of $\mathcal{V}$ for σp, the sequence resulting from the request of page p after processing σ. The sets are updated as follows [44]:

$$S^p = \begin{cases} (\{p\}, \underbrace{S_1 \cup \{p\}}_{S_2^p}, \underbrace{S_2 \cup \{p\}}_{S_3^p}, \ldots, \underbrace{S_{j-1} \cup \{p\}}_{S_j^p}, S_{j+1}, \ldots, S_k) & p \in S_j, p \notin S_{j-1} \\ (\{p\}, \underbrace{S_2 \cup \{p\}}_{S_2^p}, \underbrace{S_3 \cup \{p\}}_{S_3^p}, \ldots, \underbrace{S_k \cup \{p\}}_{S_k^p}), & \text{if } p \notin S_k \end{cases}$$

In the case $p \in S_k$ there exists always an index j such that p belongs to $S_j, \ldots S_k$ but not to $S_1, \ldots, S_{j-1}$ due to the property $S_1 \subsetneq S_2 \subsetneq \cdots \subsetneq S_k$. Since S_k always contains at least k items we conclude that valid configurations never contain pages which are not in S_k.

Layer Representation. Having information about the cache content of the optimal solution is a powerful tool to analyze algorithms in models where the cost of the online algorithm is compared to the optimal cost. However this information can also be tracked and used in the design of online algorithms. The first such algorithms were Partition [48] and Equitable [1]. Both are optimal randomized algorithms as regards the competitive ratio. Equitable uses a space efficient version of the set inclusion representation of $\mathcal{V}$, presented in the previous paragraph. It maintains a sequence of k so-called *layers* $L = (L_1|L_2|\ldots|L_k)$, where $L_1 = S_1$ and $L_j = S_j \setminus S_{j-1}$. Note that the layer representation provides complete information to compute each set S_j since $S_j = L_1 \cup \cdots \cup L_j$.

The initialization and update rule of the layers can be directly adapted from the set inclusion representation. Initially we have $L = (p_1|p_2|\ldots|p_k)$ and upon a request of a page p the following update rule [1] ensues:

$$L^p = \begin{cases} (\{p\}|L_1|\ldots|L_{j-1}|L_j \cup L_{j+1} \setminus \{p\}|L_{j+2}|\ldots|L_k) & \text{if } p \in L_j \text{ and } j < k \\ (\{p\}|L_1|\ldots|\ldots|L_{k-1}) & \text{if } p \in L_k \\ (\{p\}|L_1 \cup L_2|L_3|\ldots|L_k) & \text{if } p \notin L_1 \cup \cdots \cup L_k \end{cases}$$

Lemma 2.2 ([1], Lemma 1) *If $(L_1|\dots|L_k)$ is the layer representation of the actual set of valid configurations $\mathcal{V}$, a set C of k pages is a valid configuration iff:*

$$|C \cap (\cup_{i\leq j} L_i)| \geq j \text{ for all } 1 \leq j \leq k.$$

Lemma 2.2 is a direct adaptation of Lemma 2.1 to the layer representation. Let index r be maximal such that $L_1, \dots, L_r$ are singletons. Achlioptas et al. [1] denote the pages in $L_1, \dots, L_r$ as *revealed*. It follows directly from Lemma 2.2 that revealed pages are always contained in a valid configuration independent of the future request sequence. Thus revealed pages are always in the cache of any optimal offline algorithm. Requests to revealed pages play a crucial role in designing online paging algorithms with few cache misses in practice and low runtime overhead [51, 54].

Inversed Layer Representation. Given a set C of k pages and the layer representation L of $\mathcal{V}$, we can easily check if C is in $\mathcal{V}$ by testing all k conditions from Lemma 2.2. However if we are given only L and want to construct all valid configurations it seems there is no simple and intuitively direct way to do so. The requirement of having to choose *at least* an amount of j pages from $L_1, \dots L_j$ does not help deciding which page to exclude and which not. We recall that the configuration C contains exactly k pages, and the k conditions from Lemma 2.2 can be stated the following way:

$$|C \cap (\cup_{i>j} L_i)| \leq k - j + 1 \text{ for all } 1 \leq j \leq k.$$

By using the reindexation $j = k-j+1$ and by adding a new layer L_0 containing all pages not in $L_1 \cup \dots \cup L_k$ we obtain the representation that we introduced in order to define and analyze OnlineMin [21].

Lemma 2.3 ([21], Lemma 1) *If $L = (L_0|L_1|\dots|L_k)$ is the inversed layer representation of $\mathcal{V}$ then a set C of k pages is a valid configuration, iff*

$$|C \cap (\cup_{i\leq j} L_i)| \leq j \text{ for all } 0 \leq j \leq k.$$

Initially each layer L_i, where $i > 0$, consists of one of the first requested k pairwise distinct pages. The layer L_0 contains all pages not in $L_1, \dots, L_k$. Adding the layer L_0 the update rule from the original layer partition (considering the reindexation) simplifies to only two cases [21]:

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

Note that in the inversed representation the revealed pages are the rightmost singletons. The main strength of this representation is that the layer index j

bounds the amount of pages from $L_0 \cup L_1 \cup \cdots \cup L_j$ in a valid configuration C. We provide a priority-based incremental selection process which constructs valid configurations [21]. The goal is to maintain the cache content of an optimal offline algorithm under the assumption that the priorities reflect the order of future requests, namely the higher the priority of a page p, the sooner we assume p to be requested again in the future. For some set S let $\min_j(S)$ and $\max_j(S)$ denote the subset of S of size j with the smallest and the largest priorities, respectively.

Definition 2.2 ([21], Definition 1) *We construct iteratively $k+1$ selection sets $C_0, \ldots, C_k$ from the layer partition $L = (L_0| \ldots |L_k)$ as follows: we first set $C_0 = \emptyset$ and then for $j = 1, \ldots, k$ we set $C_j = \max_j(C_{j-1} \cup L_j)$.*

By construction the set C_k is a valid configuration and thus contains all revealed pages and no page from L_0. For designing an online algorithm which stays in valid configurations it is desirable to have a simple update rule for C_k. Under the assumption that we change only the priority of the currently requested page, we show that C_k can be updated the following way without recomputing the selection process from the beginning:

Theorem 2.1 ([21], Theorem 1) *Let p be the requested page. Given C_k, we obtain C_k^p as follows:*

1. *$p \in C_k$: $C_k^p = C_k$*
2. *$p \notin C_k$ and $p \in L_0$: $C_k^p = C_k \setminus \min(C_k) \cup \{p\}$*
3. *$p \notin C_k$ and $p \in L_i$, $i > 0$: $C_k^p = C_k \setminus \min(C_j) \cup \{p\}$, and $j \geq i$ is the smallest index with $|C_j \cap C_k| = j$.*

Initially, the update rule from Theorem 2.1 was used with random priority assignment and resulted in the runtime efficient strongly competitive randomized algorithm OnlineMin. The algorithm class using deterministic priority assignments is denoted OnOPT [54]. Algorithms from this class have been shown to adapt to easy inputs regarding the *attack rate* [54]. We use the properties of the OnOPT class to show that optimal offline algorithms perform cache misses only on requests to pages in L_0.

Lemma 2.4 *Let p be the actually requested page and L the inversed layer representation of $\mathcal{V}$. Page p is in the cache of an optimal offline algorithm iff $p \notin L_0$.*

Proof. Let C_{OPT} denote the cache content of an optimal offline algorithm and $L = (L_0|L_1| \ldots |L_k)$ be the layer representation after processing σ. Recall that due to Lemma 2.1 optimal offline algorithms are always in valid configurations. By Lemma 2.3 we immediately conclude that if p is in L_0 it cannot be in C_{OPT}. Now we show that if p is not in L_0 it is in the cache of an optimal algorithm.

We use an (offline) algorithm A from the OnOPT class where we define the priority assignment to be the following: upon request of page q we assign q the negated timestamp of its future request as priority. Recall that OnOPT algorithms use the update rule from Theorem 2.1. Since the cache of an OnOPT algorithm is identical to the outcome C_k of the priority based selection process from Definition 2.2 and p has the highest priority of all pages $L_1, \ldots, L_k$ we conclude that p is in the cache of A. Since each optimal offline algorithm faults on requests from L_0 and A faults *only* on requests from L_0 we get a contradiction to the optimality of OPT if we assume $p \notin C_{OPT}$. □

Compressed Layer Representation. Given a fixed set of valid configurations $\mathcal{V}$, all the three previously introduced approaches do not have an unique representation. The simplest way to see this is the initialization step after the first k pairwise distinct pages $p_1, \ldots, p_k$ have been requested. In this case $\mathcal{V}$ contains only the configuration $C = \{p_1, \ldots, p_k\}$ and does not depend on the request order. The layer representations use an arbitrary permutation and assign each page to one layer. Assigning pages with identical characteristics with respect to $\mathcal{V}$ in different layers or sets seems to be counterintuitive. We propose an unique representation of $\mathcal{V}$. Given the inversed layer representation L, a compressed representation $\mathcal{L}$ [54] is defined which groups all consecutive singletons of L. An algorithmic description of this compression process is given in Algorithm 1.

Algorithm 1 Layer compression

```
procedure LAYER COMPRESSION(Layers L = (L_0|...|L_k))      ▷ Compress L
    L_0 = L_0;
    T = ∅;
    for i = 1 to k − 1 do
        if |L_i| = 1 then                                   ▷ L_i is singleton
            L_i = ∅; T = T ∪ L_i;
        else                                                ▷ L_i is not singleton
            L_i = L_i ∪ T; T = ∅;
        end if
    end for
    L_k = L_k ∪ T;
end procedure
```

Here is an example for $k = 7$:

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

We have shown that $\mathcal{L}$ is consistent with Lemma 2.3 and thus it inherits all results for the inversed layer representation excepting the layer update rule. Like

in the inversed layer representation, $\mathcal{L}_0$ contains the pages which are not in any valid configuration and all revealed pages are grouped in $\mathcal{L}_k$. *Unrevealed* pages are pages whose presence in C_{OPT} depends not only on σ but also on the future request sequence. These pages are in $\mathcal{L}_1, \ldots, \mathcal{L}_{k-1}$. Note that if a page p is requested from such a layer it is for sure in the cache of OPT due to Lemma 2.4.

An update rule maintaining the compressed layer representation is also presented [54]. Initially it assigns $\mathcal{L}_k$ the first k pairwise distinct requested pages; $\mathcal{L}_1, \ldots, \mathcal{L}_{k-1}$ contain no pages. Let $\mathcal{L}$ and $\mathcal{L}^p$ be the compressed representation of L and L^p respectively. $\mathcal{L}^p$ can be obtained directly from $\mathcal{L}$ as follows:

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

The compressed layer representation has the advantage that upon requests to revealed pages nothing changes. Experiments on real-world traces have shown that usually more than 99% of the requests are to revealed pages. This leads to significant runtime improvements of OnOPT algorithms [54]. Another advantage is that the number of non-empty layers is much smaller than k on real-world traces. This allows for more efficient implementations.

Chapter 3

Contributions

In this chapter the most relevant parts of our published papers are summarized. We listed for each subsection in Table 3.1 the corresponding published paper and its location within this thesis.

Subsection	Paper	Corresponding Chapter
3.1.1.	[20], [21], [50]	4,5
3.1.2, 3.1.3	[20], [21]	4
3.1.4	[50], [52]	5
3.2.1, 3.2.2	[51]	6
3.2.3	[54], [53]	7

Table 3.1: Connection between the contents of this chapter and the published work.

3.1 Randomized Algorithms

The first part of our research focuses on strongly competitive randomized paging algorithms. We propose algorithms which match the lower bound of the competitive ratio and improve the previous results in terms of *runtime* [20, 21] and *memory requirement* [50]. The memory requirement is usually analyzed in terms of bookmarks, i.e. the number of pages not in cache that an online algorithm keeps track of.

Although the tight bound of H_k on the competitive ratio of randomized algorithms is known for decades, known algorithms matching this bound are rather complex. It is not an easy task to mathematically capture the simplicity of algorithms. Chrobak et al. [23] proposed to limit the memory requirement and runtime of algorithms in order to make them simpler. In the case of memory consumption the best case occurs if an algorithm is trackless, namely it does not use any bookmarks. Bein et al. [9] proved that no trackless randomized

algorithm can be H_k-competitive. The MARK algorithm is trackless and fast, but it was shown that no variant of this algorithm can perform better than $(2H_k - 1)$-competitive [23]. The memory requirement and runtime per request of the strongly competitive algorithm PARTITION [48] can be linear in the size of the request sequence, and thus not bounded by the cache size k. EQUITABLE [1] uses $O(k^2 \log k)$ memory and processes a page request in $O(k^2)$. A variant of EQUITABLE, denoted EQUITABLE 2 [11], further improves the memory requirement to $O(k)$, while the running time still remains $O(k^2)$. This memory improvement solved an open problem stated by Borodin and El-Yaniv [14]. Bein et al. [11] conjectured that a better result, namely $o(k)$ bookmarks, is possible.

3.1.1 New Results

We propose the algorithm ONLINEMIN [20] that handles each page request in $O(\log k)$ time in the worst case and we improve it to $O(\log k / \log \log k)$ [21]. This is a significant improvement over the fastest known algorithm, EQUITABLE2, which needs $O(k^2)$ time per request.

The key element of our algorithm is a new priority-based incremental selection process which always yields a cache content, that is identical to the cache of the optimal offline algorithm under the assumption that page priorities reflect the order of future requests. The analysis of this process results in a simple cache update rule which is different from the EQUITABLE algorithms [1, 11], but leads to the same probability distribution of the cache content for random priorities. A simple implementation of our update rule requires $O(k)$ time per request. Additionally we design data structures that result in two more efficient implementations: the first implementation uses pointer-based data structures to achieve $O(\log k)$ worst-case time per page request, whereas the second implementation exploits the power of the RAM model to achieve $O(\log k / \log \log k)$ worst-case time per request. The main contribution concerning the data structures in the RAM model belongs to my co-author Gerth Stølting Brodal.

In our paper [50] we address the $o(k)$ bookmark conjecture [10]. We first provide a tighter analysis of algorithm EQUITABLE2 reducing the number of bookmarks from $2k$ to $\approx 0.62k$, which is the first solution using less than k bookmarks. We give a negative result showing that EQUITABLE2 cannot achieve a competitive ratio of H_k using $o(k)$ bookmarks. Nonetheless, we show that it can trade competitiveness for space: if it is allowed to be $(H_k + t)$-competitive, it requires $k/(1+t)$ bookmarks. We propose the new algorithm PARTITION2 which is a variant of the PARTITION algorithm. PARTITION2 improves the bookmark requirements of PARTITION from $\Theta(n)$ to $\Theta(k/\log k)$ where n is the input size. This proves the $o(k)$ bookmark conjecture.

request	valid configurations
$1, 2, 3$	$\{1, 2, 3\}$
4	$\{1, 2, 4\}, \{1, 3, 4\}, \{2, 3, 4\}$
5	$\{1, 2, 5\}, \{1, 3, 5\}, \{2, 3, 5\}, \{1, 4, 5\}, \{2, 4, 5\}, \{3, 4, 5\}$
4	$\{1, 4, 5\}, \{2, 4, 5\}, \{3, 4, 5\}$
2	$\{2, 4, 5\}$

Table 3.2: An example of valid configurations.

3.1.2 Constructing Valid Configurations

A *cache configuration* C is a subset of k pages. The PARTITION and EQUITABLE algorithms maintain a probability distribution over the set of all possible cache contents. Ideally only configurations which might correspond to the cache content of the optimal offline algorithm should have non-zero probability; we call these configurations *valid*. An overview on valid configurations including a formal definition is given in Section 2.2.3. In the following we give an intuitive explanation and restate needed properties from Subsection 2.2.3.

We do not know the precise cache content of OPT since it depends on the future request sequence but, given the sequence so far, we can restrict the possible configurations. We give an intuitive example for cache size $k = 3$ and the pageset $\{1, 2, \ldots, 6\}$. Consider the following requests:

$$\sigma = 1, 2, 3, 4, 5, 4, 2, \ldots$$

Starting with the initial request sequence $1, 2, 3$ we list in Table 3.2 the set of valid configurations $\mathcal{V}$ after each request. If $\mathcal{V}$ contains only one configuration we say that we are in a *cone*. In a cone we know the precise cache content of OPT. Given $\mathcal{V}$, we distinguish between three types of pages. A page p is called OPT-miss if p is not contained in any configuration $C \in \mathcal{V}$, which means that p is for sure not in the cache of OPT (independent of the future request sequence). In the given example page 6 is always OPT-miss. A page p is called *revealed* if it is contained in all configurations $C \in \mathcal{V}$, which means that it is for sure in the cache of OPT. In our example 4, and 5 are pages revealed directly after the second request of 4. All other pages are called *unrevealed*. These pages are contained in at least one configuration but not in all of them.

Inversed Layer Representation. A characterization of the set $\mathcal{V}$ of all valid configurations was first given by Koutsoupias and Papadimitriou [44]. We introduced an equivalent variant of this, denoted *inversed layer representation* [20,21]. It partitions the pageset into $k + 1$ sets $L_0, L_2, \ldots, L_k$ denoted *layers*. Initially each L_i, where $i > 0$, contains one of the first k requested pages and L_0 contains

request	L_0	L_1	L_2	L_3
$1,2,3$	$4,5,6$	1	2	3
4	$5,6$	1	$2,3$	4
5	6	1	$2,3,4$	5
4	6	$1,2,3$	5	4
2	$6,1,3$	5	4	2

Table 3.3: An example of the inversed layer representation.

all the other pages. Upon a request to page p, we use the following update rule:

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

Lemma 3.1 ([20], Lemma 1) *If $L = (L_0|L_1|\ldots|L_k)$ is the inversed layer representation of $\mathcal{V}$ then a set C of k pages is a valid configuration, iff*

$$|C \cap (\cup_{i \leq j} L_i)| \leq j \text{ for all } 0 \leq j \leq k.$$

Let r be minimal such that all layers L_i, with $i > r$, are singletons (contain exactly one page). The set of revealed pages is given by $L_r \cup L_{r+1} \cup \cdots \cup L_k$. The layer L_0 contains all OPT-miss pages and $L_1 \cup L_2 \cup \cdots \cup L_{r-1}$ are the unrevealed pages. The set of pages which are not in L_0 is denoted *support*. Consider $k = 3$, the pageset $\{1, 2, \ldots, 6\}$ and $\sigma = 1, 2, 3, 4, 5, 4, 2, \ldots$. Starting with the initial request sequence $1, 2, 3$, in Table 3.3 we list the inversed layer representation of $\mathcal{V}$ after each request.

Selection Process. Using the inversed layer representation we introduce a priority-based selection process which allows us to construct each valid configuration. We assume that pages have distinct priorities. Let $\max_j(S)$ denote the subset of S containing the j pages with the highest priority.

Definition 3.1 ([20], Definition 1) *We construct iteratively $k+1$ selection sets $C_0, \ldots, C_k$ from the layer partition $L = (L_0|\ldots|L_k)$ as follows. We first set $C_0 = \emptyset$ and then for $j = 1, \ldots, k$ we set $C_j = \max_j(C_{j-1} \cup L_j)$.*

The outcome of the selection process is C_k and it always contains k pages which form a valid configuration. Moreover C_k corresponds to the cache content of an optimal offline algorithm under the assumption that page priorities reflect the order of future requests, namely the higher the priority of a page p, the sooner we assume p to be requested again in the future.

Let p be the requested page and assign p a new priority directly after it has been requested; the priorities of all other pages in the support are not modified.

Since the layers change upon each request, the outcome C_k of the selection process might also change. We have shown in [20] that C_k can be updated without recomputing the selection process from the beginning, as follows:

Theorem 3.1 ([20], Theorem 1) *Let p be the requested page. Given C_k, we obtain C_k^p as follows:*

1. *$p \in C_k$: $C_k^p = C_k$*
2. *$p \notin C_k$ and $p \in L_0$: $C_k^p = C_k \setminus \min(C_k) \cup \{p\}$*
3. *$p \notin C_k$ and $p \in L_i$, $i > 0$: $C_k^p = C_k \setminus \min(C_j) \cup \{p\}$, and $j \geq i$ is the smallest index with $|C_j \cap C_k| = j$.*

3.1.3 OnlineMin

Algorithm OnlineMin is H_k-competitive, requires $O(k)$ space and processes a page in $O(\log k / \log \log k)$ time .

Algorithm. The cache content of OnlineMin corresponds to the outcome C_k of the selection process from Definition 3.1. Upon a request to page p the update rule from Theorem 3.1 is applied followed by the layer update rule. Note that only the layers $L_1, L_2, \ldots, L_k$ need to be stored. Finally we assign p a new priority, such that its rank is uniformly distributed within the set of all support pages.

Competitiveness. The probability distribution of the cache content of OnlineMin can be described the following way: assume that the ranks of priorities of the pages in the support are uniformly distributed; the probability of configuration C is given by the probability that C is the outcome of the selection process. We have proven that this probability distribution over the set of valid configurations is identical to the Equitable algorithms [20,21]. This implies that OnlineMin and Equitable have the same expected cost and thus OnlineMin is H_k-competitive.

Space. The update rule for the layers imply that the support can grow arbitrarily large. We apply the forgiveness rule from [11] for the layers; this bounds the size of the support to $3k$ and works as follows: if the size of the support reaches the threshold of $3k$ and page p is requested from L_0 we first artificially insert p in L_1 and perform the update rule, as p was a page not in the cache requested from L_1. By the layer update rule and the fact that layers are never empty this step does not increase the support above the threshold of $3k$. Doing so, we use an approximation of the actual set of valid functions. Nonetheless Bein et al. [11] showed that the Equitable distribution over this approximated set still leads to an H_k-competitive algorithm.

Runtime. A straightforward implementation can be carried out using an array to store the layers $L_1, L_2, \ldots, L_k$. Performing the layer update and cache update by linear search leads directly to $O(k)$ runtime per page request. In order to achieve a better runtime, we use the observation that $L_1, L_2, \ldots, L_k$ can be seen as consecutive time intervals. A page p is in layer L_i iff the timestamp of its last request corresponds to the time interval of L_i.

The layer update can be done in $O(\log k)$ time using standard data structures, e.g. AVL-trees. The challenging part appears for the cache update rule, namely when a page from L_i, where $i > 0$, is requested and it is not in the cache. We design a tree-based data structure which supports this operation (and all others) in worst-case runtime $O(\log k)$ per request. We refine this data structure using the power of the RAM model in order to improve the runtime to $O(\log k / \log \log k)$ [21]. As already mentioned, this last improvement was brought my co-author Gerth Stølting Brodal.

3.1.4 Improving Space

All known randomized algorithms which are strongly competitive need to store information about the pages in the support, e.g. the layer partition. Since the support can grow arbitrarily large there is no upper bound on the number of bookmarks if we insist that our algorithm maintains a probability distribution over the actual set of valid configurations. Thus PARTITION and the (initial) EQUITABLE algorithm need $\Theta(n)$ bookmarks. The EQUITABLE algorithm was shown to remain H_k-competitive if an approximation of $\mathcal{V}$ with a support size of at most $3k$ is used.

This approximation is achieved by applying a so-called *forgiveness step* in case the support reaches the size of $3k$. We refine the analysis of the EQUITABLE distribution using our priority-based approach and show that this forgiveness step can be applied when the support size is $\approx 1.62k$, which implies $\approx 0.62k$ bookmarks. Moreover, we show that EQUITABLE cannot achieve a support size of $o(k)$ and be H_k-competitive at the same time. In order to confirm the conjecture [10] that there exists a strongly competitive randomized algorithm using $o(k)$ bookmarks we introduce another approximation scheme of the valid configurations that we apply to PARTITION.

Manipulations of $\mathcal{V}$. Note that the set $\mathcal{V}$ determines precisely the cost of the optimal offline algorithms since a request to a page p leads to a cache miss for OPT iff p is not contained in any of the valid configurations. Now consider the layer representation of $\mathcal{V}$ from Lemma 3.1. If we *artificially* insert a page p from L_0 into some layer L_i, with $i > 0$, we extend the set of valid configurations. Each valid configuration remains valid after the modification. This approximation step leads to an overestimation of the optimal offline cost and we are allowed to use it in competitive analysis, but only if we charge OPT the cost 0.

The forgiveness step proposed by Bein et al. [10] does the following: the page p from L_0 is artificially inserted into L_1 and then p is requested as a page from L_1. The cache update is performed using the case where p is a page not in the cache and requested from L_1. This leads to the cost of 1 for EQUITABLE and cost 0 for OPT. This step brings a disadvantage in competitive analysis for the online algorithm but it has the benefit of not increasing the support due to the layer update rule for L_1 and the fact that L_1 always contains at least one page.

Equitable. Based on the actual set of valid configurations, Achlioptas et al. [1] proposed a potential function Φ in order to track the cost of EQUITABLE. The potential Φ is defined as the cost of a *lazy attack sequence*, i.e. a sequence of requests where OPT has 0 cost and we end in a cone. In the same work it was proven that every lazy attack sequence has the same cost and thus Φ is well-defined. In terms of the inversed layer representation, Φ can be obtained as the cost of consecutive requests from L_1 until all layers are singletons. For each request it was shown that the following holds:

$$\Delta cost(Eq) + \Delta\Phi \leq \Delta cost(OPT) \cdot H_k.$$

By the very definition of the potential Φ the inequation holds for lazy requests since $\Delta cost(Eq) + \Delta\Phi = 0$. Otherwise if p is requested from L_0 it was shown that $\Delta\Phi \leq H_k - 1$. If we apply the forgiveness step this analysis does not hold. The main idea is that $\Delta\Phi$ is at most $H_k - 1$ and in the case of growing support this value decreases, thus occurring the so-called *savings*. These savings were tracked by Bein et al. using a second potential Ψ in order to compensate the uncovered cost in the case of forgiveness.

While the definition of Φ makes one's life easy when dealing with requests from the support, no closed form is known for Φ or for its increase upon a request from L_0. The main challenge is a close approximation of $\Delta\Phi$ in order to maximize the saving function Ψ, namely the difference between $H_k - 1$ and $\Delta\Phi$. Using our priority-based characterization of the EQUITABLE distribution we have shown that the following holds:

Theorem 3.2 ([50], Theorem 1) *For a request to a page $p \in L_0$ where no forgiveness is applied, let i be the largest index with $|L_i| > 0$; $i = 0$ if we are in a cone. We have that:*

$$H_{k-i} - H_1 \leq \Delta\Phi \leq H_k - H_1 - i/(k+1).$$

We keep track of the savings $i/(k+1)$ in a second potential Ψ which is always greater than or equal to 0 and depends on the differences $|L_i| - i$, where $0 < i < k$. We show for the new saving potential Ψ that the following holds after each request if we use the forgiveness mechanism already at a support size of $\approx 1.62k$:

$$\Delta cost(Eq) + \Delta\Phi + \Delta\Psi \leq \Delta cost(OPT) \cdot H_k.$$

Req	Layer embedding
-	$L = (7, 8, 9 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6)$
9	$L = (7, 8 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5, 6, \star \mid 9)$
6	$L = (7, 8 \mid 1 \mid 2 \mid 3 \mid 4, 5, \star \mid 9 \mid 6)$
8	$L = (7 \mid 1 \mid 2 \mid 3 \mid 4, 5, \star \mid 9, 6, \star \mid 8)$
1	$L = (7 \mid 2 \mid 3 \mid 4, 5, \star \mid 9, 6, \star \mid 8 \mid 1)$
9	$L = (7 \mid 2 \mid 3 \mid 4, 5, \star, 6, \star \mid 8 \mid 1 \mid 9)$
6	$L = (7 \mid 2 \mid 3, 4, 5, \star, \star \mid 8 \mid 1 \mid 9 \mid 6)$
3	$L = (7 \mid 2, 4, 5, \star, \star \mid 8 \mid 1 \mid 9 \mid 6 \mid 3)$
5	$L = (7, 2, 4 \mid 8 \mid 1 \mid 9 \mid 6 \mid 3 \mid 5)$

Figure 3.1: Example for the layer embedding of the set partition.

The saving potential from [11] allows the use of forgiveness at the threshold of $3k$. We conclude that EQUITABLE2 can be implemented using $\approx 0.62k$ bookmarks. The following questions arise: how much better can we approximate the savings and especially can EQUITABLE2 be implemented with $o(k)$ bookmarks? We construct an input which shows that if EQUITABLE2 uses less than $0.25k$ bookmarks, it is no longer H_k-competitive.

Partition2. The algorithm PARTITION uses a different approach in order to keep track of the set $\mathcal{V}$ of valid configurations which we denote *set partition*. Like the layer partition the set partition can also have different representations of the same set $\mathcal{V}$. In contrast to EQUITABLE the probability distribution of which depends only on the set $\mathcal{V}$, the distribution of PARTITION depends on the specific representation of $\mathcal{V}$. We have shown that the set partition can be embedded in the inversed layer representation the following way. The layers become ordered sets, which also contain, in addition to pages, a special *marker* $\star$. The initialization remains the same. The update rule changes mainly in the case where p is requested from L_0:

$$L_{k-1} = (L_{k-1}, L_k, \star),\ L_k = \{p\}.$$

For the merging operation $L_{i-1} \cup L_i \setminus \{p\}$ in the case $p \in L_i$ we take p out of L_i and concatenate L_{i-1} with L_i without removing any marker. Upon merging L_1 with L_0 we delete all markers from the resulting layer L_0. An example is given in Figure 3.1.

The distribution of the cache content C of PARTITION can be described in the following way: go from left to right through all layers $L_1, \ldots, L_k$. If the element is a page, add it to C; if it is the symbol $\star$ evict one page from C uniformly at random.

Lemma 3.2 ([50], Lemma 9) *If p is requested from L_i, where $i > 0$, the probability that p is not in the cache of* Partition *is at most:*

$$\sum_{j \geq i} \frac{x_j}{j+1}.$$

Provided with the cache miss probability bound from Lemma 3.2 we use the following potential:

$$\Phi = \sum_{j=1}^{k-1} x_j \cdot (H_{j+1} - 1).$$

Our new potential depends only on the size of the layers and not on the additional information given by the layer embedding. Potential Φ is an upper bound on the maximal cost of a lazy attack sequence. Note that in the case of Partition it does not hold that all lazy attack sequences have the same cost as it holds for the Equitable distribution. Unlike for Equitable we do not need a second potential for the savings in order to apply approximations to $\mathcal{V}$.

We introduce algorithm Partition2 which works identically to Partition except when the support reaches the threshold of $k + 3t$ where $t = \Theta(k/\log k)$ and a page $p \in L_0$ is requested. We distinguish two forgiveness modes: the *regular forgiveness* and the *extreme forgiveness*. If the first t layers contain more than $2t$ pages we apply the regular forgiveness, which is an adaptation of the Equitable2 forgiveness step. We choose the leftmost page from L_1 not in the cache and swap it with p. Then we request p as if it was q. The cost of 1 for Partition2 is shown to be totally covered by the potential decrease $\Delta\Phi$. The extreme forgiveness mode is applied if $|L_1| + |L_2| + \cdots + |L_t| \leq 2t$. We apply regular forgiveness for each of the following page requests in L_0 until we reach a cone. The main idea in the analysis is that in this case we have at least t elements in high index layers, which leads to a potential of at least $k - 1$. This allows Partition2 to perform $k - 1$ cache misses even if OPT pays the cost 0. Together with a potential argument for the cases where no forgiveness is applied, we obtain our main result in Theorem 3.3.

Theorem 3.3 ([50] Theorem 5) Partition2 *uses $\Theta(\frac{k}{\log_k})$ bookmarks and is H_k-competitive.*

As long as a randomized algorithm keeps a bookmark for every page with non-zero probability, we have shown that it needs $\Omega(k/\log k)$ bookmarks. Under this constraint the memory requirement of Partition2 is asymptotically optimal.

3.2 Deterministic Algorithms

In spite of being frequently criticized, competitive analysis is a natural and simple model to analyze online algorithms. Recently, many alternatives to competitive

analysis have been proposed, and most of them led to the conclusion that LRU is the best or at least among the best online algorithms. Few of these models were able to provide new ideas in algorithm design, more precisely to point out *new* algorithms with few cache misses on real-world inputs. Our research on the competitive ratio of randomized algorithms encouraged us to look for new deterministic algorithms in order to outperform LRU. The basic idea we use is simple: we look at the randomized algorithm OnlineMin as a big collection of reasonable deterministic algorithms which may contain candidates able to outperform LRU. We denote this class of deterministic algorithms OnOPT [51]. Algorithms within this class use a deterministic priority assignment, and keep the cache content of the optimal offline algorithm under the assumption that the priority assignment reflects the order of future page requests. A pseudo-code is provided in Algorithm 2.

Algorithm 2 OnOPT class

```
procedure OnOPT(Page p, Cache M)                    ▷ Processes page p
    Assign p its priority (deterministically)
    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 j such that j ≥ i and |(L_1 ∪ ··· ∪ L_j) ∩ M| = j
        Evict page in L_1 ∪ ··· ∪ L_j having smallest priority
    end if
    Update the layers                               ▷ Layers update
end procedure
```

3.2.1 Input Parametrization

We assume that the reader is familiar with the inversed layer representation of the set of all valid configurations from Subsection 3.1.2. Consider the request sequence generated by an adversary who wants to maximize the cost of an online algorithm and minimize the cost of OPT. The following question arises: which are evil requests and which are not? Requests from L_0 always incur the cost of 1 for OPT and thus the adversary should minimize the number of these requests. Requests to revealed pages have no cost for the optimal algorithm, but the online player also knows that these pages are for sure in the cache of OPT. Thus if the online player is smart and stays within the valid configurations this type of requests is harmless. Requests to unrevealed pages have no cost for OPT and the online player might fault on them, because the information whether they are in the cache of OPT or not depends on future requests.

Class	Comp. ratio
CacheMin	∞
Mark	$[2r-1, 2r]$
OnOPT	r

Algorithm	Comp. ratio
LFU, MRU, LIFO	∞
FWF	$[2r-1, 2r]$
LRU, FIFO	r

Table 3.4: The guaranteed competitive ratio for the generic classes (left) and for classic algorithms (right) from [51].

Definition 3.2 *Given some input sequence σ let $\lambda_r(\sigma)$, $\lambda_u(\sigma)$, and $\lambda_0(\sigma)$ denote the number of requests in σ to revealed pages, unrevealed pages in the support, and pages that are not in the support, respectively.*

Attack Rate. We use the input parametrization from Definition 3.2 to define the *attack rate*.

Definition 3.3 ([51], Definition 2.1) *For some input σ, the attack rate $r(\sigma)$ is defined as $r(\sigma) = \frac{\lambda_0(\sigma)+\lambda_u(\sigma)}{\lambda_0(\sigma)}$. Also, we denote by $\mathcal{I}(r)$ the set of inputs having an attack rate at most r, i.e. $\mathcal{I}(r) = \{\sigma | r(\sigma) \leq r\}$.*

Since between two requests from L_0 there are at most $k-1$ consecutive unrevealed requests we have that the attack rate ranges from 1 to k. The attack rate can be seen as a measure of the evilness of the adversary. We can perform competitive analysis under the restriction of the parameter r, which in real-world traces is much smaller than k (see Figure 3.2). The best possible competitiveness guarantee for inputs from $\mathcal{I}(r)$ is r. We apply competitive analysis on 6 standard paging algorithms based on parameter r. Moreover we investigate three priority-based classes of algorithms. We assume pages to have priorities which should reflect the guessed order of future requests. The simplest one is the CacheMin class, which evicts the page with the lowest priority whereas the Mark class replaces the unmarked page with the lowest priority. OnOPT algorithms use the cache update rule from the randomized algorithm OnlineMin, which always leads to r-competitive algorithms. The results are summarized in Table 3.4. The optimal competitive ratio r is achieved by LRU and FIFO but not by FWF. Like in classical competitive analysis algorithms LFU, MRU, and LIFO have no bounded competitive ratio.

Fault Rate. The defined attack rate r is useful for competitive analysis, but it cannot be employed for the analysis of the fault rate. If we request only pages from L_0 we obtain $r = 1$ and although we have an easy input in terms of competitive analysis, we have the worst possible fault rate of 1. Nonetheless, if we also consider the parameter λ_r (the number of revealed requests) we obtain the bound from Observation 3.1 on the fault rate of r-competitive algorithms like LRU, FIFO and all algorithms from the OnOPT class.

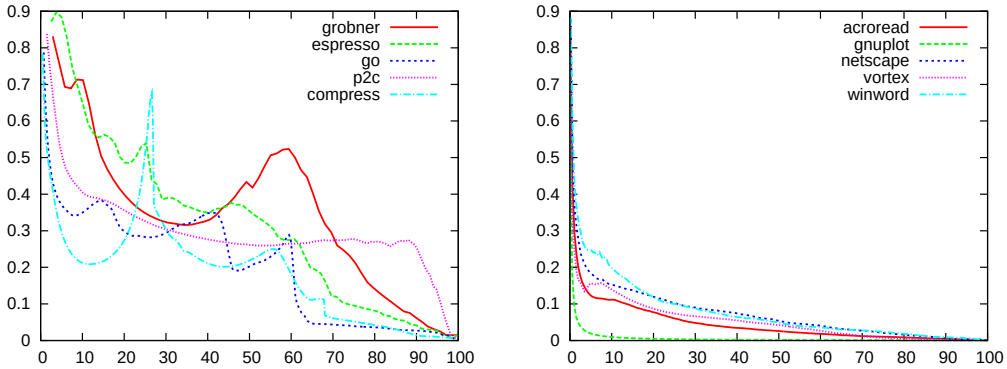


Figure 3.2: The attack rate r in real-world inputs. The x-axis shows the percentage of pages that fit in cache, i.e. k, and the y-axis shows the ratio between r and the cache size k.

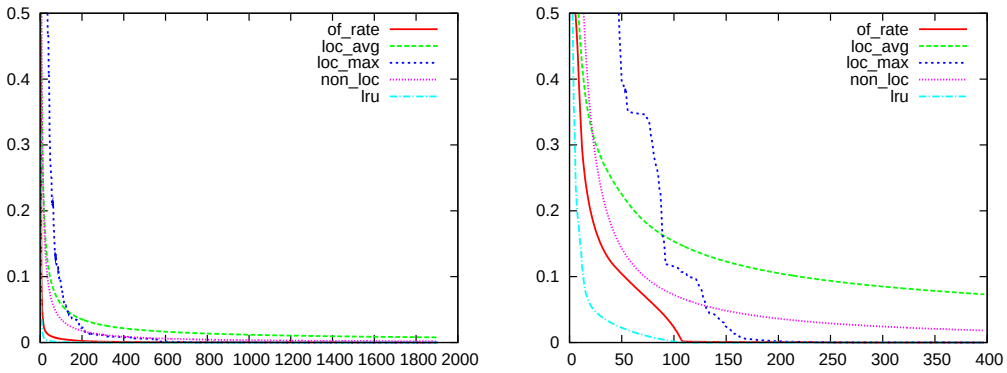
Observation 3.1 *If A is an r-competitive algorithm, the fault rate is at most*

$$\frac{\lambda_u + \lambda_0}{\lambda_u + \lambda_0 + \lambda_r}.$$

This parameterized bound on the fault rate allows us to compare our approach to the fault-rate analysis based on locality of reference. The *Max-Model*, *Average-Model* [2] and the *non-locality* model [30] point out that LRU has the best possible guarantee for online algorithms in the respective model. For a brief overview of the three models refer to Subsection 2.2.2. We compare the three guarantees to our fault rate guarantee from Observation 3.1 which we denote *of_rate*. The results for two data sets are given in Figure 3.3. 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 2/3 of the pageset our parametrization outperforms locality of reference in the Max-Model, in many cases by factors of thousands. For larger cache sizes the Max-Model gives the closest upper bounds.

3.2.2 Recency Duration Mix

As previously mentioned, choosing an OnOPT algorithm means always choosing an r-competitive algorithm staying in valid configurations, no matter its priority assignment. As a result, we are given the chance to experiment with a wide range of priority policies. We propose the priority assignment Recency Duration Mix (RDM) [51], the use of which leads to good performance on real-world inputs.



acroread compress

Figure 3.3: Our fault rate guarantee of_rate $= \frac{\lambda_0+\lambda_u}{\lambda_0+\lambda_u+\lambda_r}$, and the three models based on locality of reference with the actual performance of LRU. The x-axis shows the cache size and the y-axis shows the fault rate.

Priority Assignment. We use a global counter t, which keeps track of the number of requests to pages in L_0 and unrevealed pages. Thus, before assigning a priority to the requested page p, we increment t only if p is not revealed. Also, for each page p in the support we store a variable t_0 which keeps the value of t at the time that p entered the support. More exactly, for any request p from L_0 we set $t_0(p) = t$. The RDM priority (see Definition 3.4) for a page p is a mix between the timespan p spent in the support $t - t_0$ (duration) and the timestamp t of its actual request (recency).

Definition 3.4 *If page p is requested, RDM assigns p the priority*

$$0.8t + 0.1(t - t_0(p)).$$

Empirical Competitive Ratio. In order to compare the performance of RDM with the performance of LRU and two of its variants, namely RLRU [17] and EELRU [61] which in practice were shown to behave better than LRU, we have conducted a series of experiments.

For this purpose, we used all the available traces (16) from [39]. The memory access traces were extracted during the computation of applications running on Linux and Windows NT operating systems. The 16 applications cover standard software such as Acrobat Reader, MS PowerPoint or GNU C/C++ compiler. The page request sequences were generated using 4KB-sized pages while removing consecutive requests to the same page.

For every dataset and cache size, we determined the empirical competitive ratio for each of the four algorithms considered, i.e. the number of cache misses performed, normalized by the performance of OPT. In Figure 3.4 we show the

results for four datasets. On all traces the performance of RLRU is similar to LRU, though it outperforms it consistently by very small margins. For almost all traces and cache sizes EELRU performs at least as good as LRU and RLRU. Our algorithm RDM 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 and more than 10% on most of the remaining datasets if the cache size is not too large. On cache sizes larger than 75% of the pageset the performance of all four algorithms is almost identical. It rarely happens that RDM has an empirical competitive ratio higher than 2. Finally, we note that, except for gnuplot, RDM outperforms EELRU as well on most cache sizes, in many cases by significant margins.

We conclude that RDM outperforms LRU, and can even compete with improved variants of LRU, such as RLRU and EELRU.

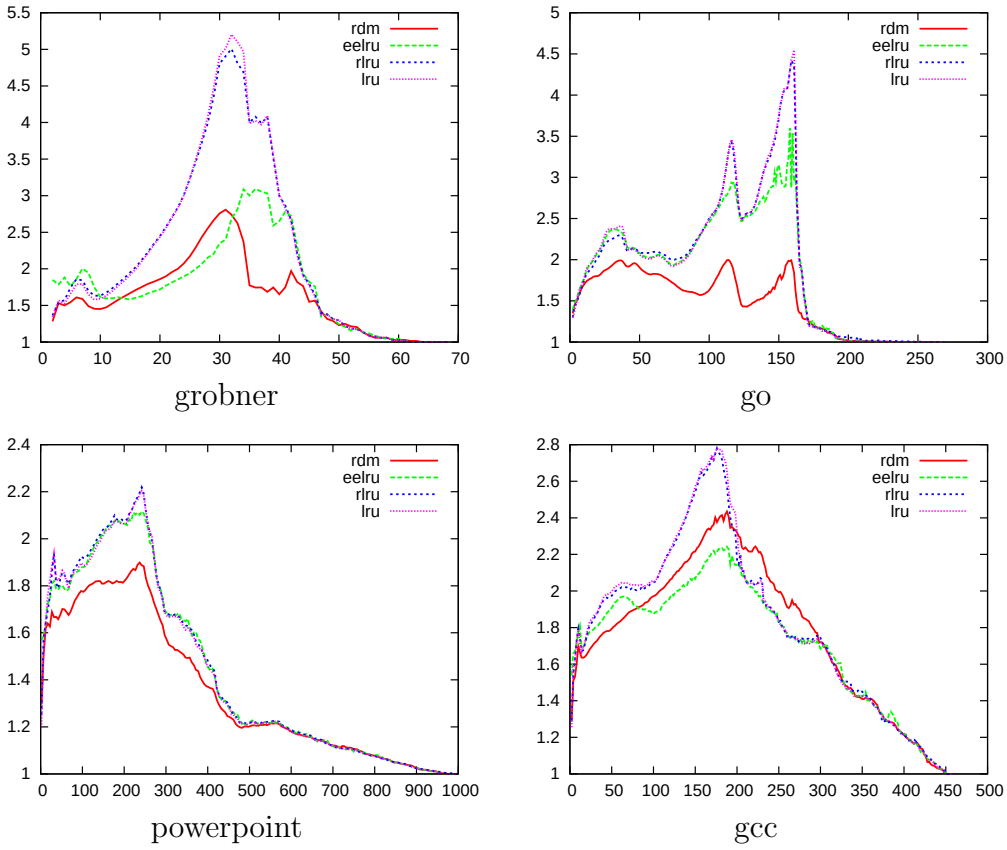


Figure 3.4: The empirical competitive ratio on selected datasets for RDM, LRU, RLRU, and EELRU. The x-axis shows the cache size and the y-axis shows the competitive ratio.

3.2.3 An Overall Evaluation

Each cache miss performed leads to a delay in the execution of the program code, thus the main goal for paging algorithms is to minimize the number of cache misses. Another necessary property of a good paging algorithm is a fast processing time of page requests since the gain in cache misses can be annihilated by a very slow runtime of the paging algorithm itself. LRU is a simple paging algorithm which incurs few cache misses on real-world inputs and supports efficient implementations. FIFO performs more cache misses than LRU but it is faster than LRU. Algorithms with a potential in reducing cache misses in comparison to LRU were proposed, among these also our algorithm RDM. All these solutions have in common that they are more complex than LRU, and it is not known if there exist implementations with reasonable runtimes. Our work [53] focused on finding an implementation for RDM such that the processing time is in the league of LRU and FIFO.

Compressed Layers. One bottleneck when implementing RDM is the layer update, that needs to be done after each request. Especially the observation that most of the requests are to revealed pages, which ensures that no cache update needs to be done, motivated us to find another characterization of the set of valid configurations. We prove in [53] that the layer update rule from Theorem 3.4 is equivalent to the update rule of the inversed layer representation. We use again $k+1$ layers $\mathcal{L}_0, \mathcal{L}_1, \ldots, \mathcal{L}_k$ and initially $\mathcal{L}_k$ contains the first k pages, $\mathcal{L}_0$ contains all other pages and the remaining layers are empty. We denote this representation the *compressed layer representation*.

Theorem 3.4 ([53], Theorem 1) *Let $\mathcal{L}$ be the compressed layer representation. We obtain $\mathcal{L}^p$, the compressed layer representation after the request of page p, as follows:*

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

The big advantage of the compressed representation is that it groups all revealed pages in layer $\mathcal{L}_k$ and upon a request to a revealed page, no update is needed. Another point is that we have empty layers in this representation. Since we can store layers as time intervals, it suffices to store only the sequence of the delimiters of non-empty layers together with a variable which represents the number of preceding empty layers. Thus the needed merging operation of layers $\mathcal{L}_i$ and $\mathcal{L}_{i-1}$ boils down to a decreasing of this variable in the case that $\mathcal{L}_{i-1}$ is an empty set. Our measurements give evidence that the number of non-empty layers is far below k (see Figure 3.5).

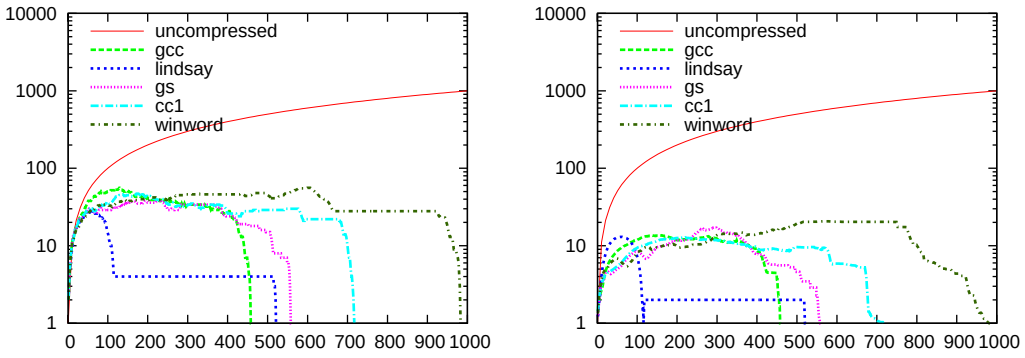


Figure 3.5: Maximum (left) and average (right) number of non-empty layers in the compressed layer partition. The x-axis is the cache size k.

A Simple RDM Implementation. The data structures proposed by us [51] ensure $O(\log k)$ runtime guarantee per request. Although this runtime is (asymptotically) reasonable, the constants are rather high. Our experiments show that most requests (a ratio of about $1 - 1/k$) are to revealed pages [53]. In order to engineer a fast implementation, we give up the general asymptotic bound of $O(\log k)$ per request and tune our implementation to ensure an $O(1)$ time for revealed requests at the price of an $O(k)$ update time for the other two request types. To do so we use the compressed layer representation. For each page we store the timestamp of its last request and its priority. The non-empty layers are stored in an array as consecutive time intervals, each with a counter for the number of preceding empty layers. If p is requested we can compare it to the layer interval corresponding to L_k. In the case that p is revealed, we just update its priority. If p is not revealed, the update of the layers (and eventually the cache) is performed using linear array traversals which takes $O(k)$ time.

Runtime Experiments. This experimental part was intended to measure the runtime of paging algorithms needed for processing the input sequence of page requests. Note that cache misses do not cause runtime penalties in this simulation. In order to do so, we first loaded the traces (with a size of up to several GB) into the main memory and then started the runtime measurement.

We compared the runtime of the new simple RDM implementation to that of the tree-based implementations of RDM [51]. For the tree-based implementation we achieved a significant speedup using the compressed layer representation instead of the uncompressed one. Overall, the simple implementation clearly outperforms the tree-based implementations. We conclude that the simple implementation is the fastest RDM implementation on the given data sets.

The next step was to compare the fastest RDM implementation with other

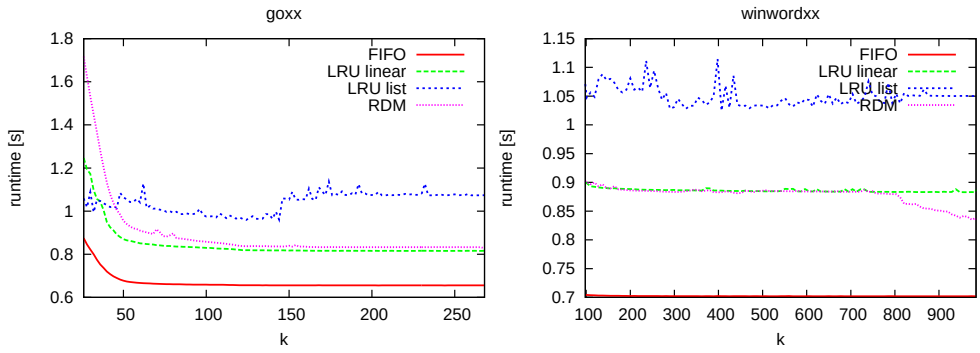


Figure 3.6: The runtime for RDM, two implementations of LRU, and FIFO.

algorithms. We decided to do this using FIFO and two implementations of LRU. 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.

It is easy to see that FIFO is one of the fastest possible algorithms. On the other hand it is well-known that LRU is a simple algorithm with a good fault rate. A first attempt to include algorithms like EELRU or RLRU was discarded since our implementations were hopelessly slow. Other (maybe trickier and thus faster) implementations of these algorithms were not available. The fastest algorithm was in all cases FIFO followed by LRU. Our algorithm RDM was slower than the fastest LRU implementation by a small margin. The runtime charts for two traces is given in Figure 3.6.

Overall Cost. We introduced a new approach to evaluate the experimental performance of paging algorithms. The cost of the algorithm is given by its runtime plus a penalty for each cache miss. In this scenario a good paging algorithm needs to have a low fault rate and a reasonable runtime. We have chosen a typical additional penalty for a cache miss of 9ms [63, Chapter 1.3.3] leading to the following overall cost: $total = runtime + \#misses \cdot 9ms$. Since it is not possible to prove that our implementation of FIFO or LRU is the fastest possible in practice we decided to set their runtime to 0. Despite this handicap RDM succeeds to outperform LRU for about 48% of the experiments. Further LRU and RDM outperform FIFO consistently. A detailed trace-by-trace comparison between LRU and RDM is given in Figure 3.7.

Trace	LRU better (in %)	RDM better (in %)
acroread	63.7	36.3
cc1	60.5	39.5
compress	31.5	68.5
espresso	52	48
gcc	57.7	42.3
gnuplot	90.1	9.9
go	39.5	60.5
grobner	34.3	65.7
gs	69.5	30.5
lindsay	90.4	9.6
netscape	41.9	58.1
p2c	5.3	94.7
powerpoint	55.3	44.7
vortex	48.5	51.5
winword	54	46

Figure 3.7: The relative performance of LRU vs RDM for each trace.

3.3 Conclusions

In this thesis we studied paging, one of the most famous online problems, by combining theoretical analysis, algorithm design, and evaluation on real-world data.

In our work we have managed to improve the runtime and memory requirement of strongly competitive paging algorithms. Although our algorithm OnlineMin can be implemented in $O(\log k/ \log\log k)$ time per request and uses only $O(k)$ memory it cannot compete with the Mark algorithm in terms of simplicity and efficiency. Our second algorithm Partition2 beats OnlineMin in terms of bookmarks, but a straightforward implementation needs $O(k^2)$ time per request. It is an interesting research issue, whether Partition2 can be implemented in $O(k)$ time or even better. All known strongly competitive algorithms keep track of pages with non-zero probability of being in the cache. We have shown that this approach has its limits as regards the bookmark complexity. One possibility to overcome this limitation, is to choose randomly the approximation of the set of valid configurations.

Although none of the strongly competitive randomized algorithms seem to be interesting for practical use, analyzing them had a significant impact on the development of the attack rate parametrization, the OnOPT class and the RDM algorithm.

Our input parametrization leads to more realistic bounds on the competitive ratio and fault rate. The analysis of deterministic paging algorithms for attack rate r inherits the simplicity of classical competitive analysis. A natural extension of this line of research is the analysis of randomized algorithms on inputs with attack rate r. We assume the competitiveness bounds for randomized algorithms

to be much closer to the observed performance than in the case of deterministic algorithms.

A new insight regarding the good performance of LRU is that it can be partially explained by its property of being r-competitive. However this holds for the whole OnOPT class, and motivates the search for other practical algorithms herein. We provided an algorithm RDM which clearly outperforms LRU and two of its improved variants on the tested inputs. We think that it is worth further investigating the OnOPT class to find other algorithms with an even better performance. Although RDM performs few cache misses it is more complex than LRU. We provided fast implementations for RDM which prove it to be a realistic candidate to use in practice.

We conclude that competitive analysis, although often criticized, can be successfully employed in designing algorithms not only having theoretical guarantees but also performing well on realistic inputs.

Chapter 4

OnlineMin: A Fast Strongly Competitive Randomized Paging Algorithm

The work **OnlineMin: A Fast Strongly Competitive Randomized Paging Algorithm** was published as a conference paper [20] and as a journal paper [21] (invited for the special issue of the journal *Theory of Computing Systems* dedicated to WAOA 2011).

[20] G. S. Brodal, G. Moruz, and A. Negoescu. Onlinemin: A fast strongly competitive randomized paging algorithm. In *Proc. 9th International Workshop on Approximation and Online Algorithms:, WAOA 2011, Revised Selected Papers*, pages 164–175. Springer, 2012

[21] G. S. Brodal, G. Moruz, and A. Negoescu. Onlinemin: A fast strongly competitive randomized paging algorithm. *Journal Theory of Computing Systems, Special issue of the 9th Workshop on Approximation and Online Algorithms*, 2013

The contents of this chapter correspond to the journal version [21] which includes all results of the conference paper [20]. One of the additions consist of RAM data structures for the further improvement of the runtime of algorithm OnlineMin.

OnlineMin: A Fast Strongly Competitive Randomized Paging Algorithm

Gerth Stølting Brodal* , Gabriel Moruz† ‡, and Andrei Negoescu† §

Abstract

In the field of online algorithms paging is one of the most studied problems. For randomized paging algorithms a tight bound of H_k on the competitive ratio has been known for decades, yet existing algorithms matching this bound have high running times. We present a new randomized paging algorithm OnlineMin that has optimal competitiveness and allows fast implementations. In fact, if k pages fit in internal memory the best previous solution required $O(k^2)$ time per request and $O(k)$ space. We present two implementations of OnlineMin which use $O(k)$ space, but only $O(\log k)$ worst case time and $O(\log k/\log\log k)$ worst case time per page request respectively.

4.1 Introduction

Online algorithms are algorithms for which the input is not provided beforehand, but is instead revealed item by item. The input is to be processed sequentially, without assuming any knowledge of future requests. The performance of an online algorithm is usually measured by comparing its cost against the cost of an optimal offline algorithm, i.e. an algorithm that is provided all the input beforehand and processes it optimally. This measure, denoted *competitive ratio* [41, 60], states that an online algorithm A has competitive ratio c if for any input sequence its cost satisfies $cost(A) \leq c \cdot cost(OPT) + b$, where $cost(OPT)$ is the cost of an optimal offline algorithm and b is a constant. If A is a randomized algorithm, $cost(A)$ denotes the expected cost. In particular, an online algorithm is denoted *strongly competitive* if its competitive ratio is optimal. While the competitive ratio is a quality guarantee for the cost of the solution computed by an online

*Department of Computer Science, Aarhus University. Åbogade 34, 8200 Aarhus N, Denmark. Email: gerth@cs.au.dk

†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 grants ME 3250/1-3 and MO 2057/1-1, and by MADALGO.

§Partially supported by the DFG grants ME 3250/1-3, and by MADALGO.

algorithm, factors such as space complexity, running time, or simplicity are also important.

In this paper we study *paging algorithms*, a prominent and well studied example of online algorithms. We are provided with a two-level memory hierarchy, consisting of a cache and a disk, where the cache can hold up to k pages and the disk size is infinite. When a page is requested, if it is in the cache a *cache hit* occurs and the algorithm proceeds to the next page. Otherwise, a *cache miss* occurs and the algorithm has to load the page from the disk; if the cache was full, a page must be evicted to accommodate the new one. The cost is given by the number of cache misses performed.

Related Work. Paging has been extensively studied over the last decades. In [12] an optimal offline algorithm, denoted MIN, was given. In [60] a lower bound of k on the competitive ratio for deterministic paging algorithms was shown. Several algorithms, such as LRU and FIFO, meet this bound and are thus strongly competitive. For randomized algorithms, Fiat et al. [32] proved a lower bound of H_k on the competitive ratio, where $H_k = \sum_{i=1}^{k} 1/i$ is the k-th harmonic number. They also gave an algorithm, named Mark, which has a competitive ratio of $(2H_k - 1)$. The first strongly competitive randomized algorithm being H_k-competitive was Partition [48]. For Partition, the memory requirement and runtime per request can reach $\Theta(n)$, where n is the number of page requests, and n can be far greater than k. Partition was characterized in [1] as counter-intuitive and difficult to understand. The natural question arises if there exist simpler and more efficient strongly competitive randomized algorithms. The Mark algorithm can be easily implemented using $O(k)$ memory and has very fast running time ($O(1)$ dictionary operations) per request, but it is not strongly competitive. Furthermore, in [23] it was shown that no Mark-like algorithm can be better than $(2H_k - 1)$-competitive. The strongly competitive randomized algorithm Equitable [1] was a first breakthrough towards efficiency, improving the memory complexity to $O(k^2 \log k)$ and the running time to $O(k^2)$ per page request. In [11] a modification of Equitable, denoted Equitable2[1], improved the space complexity to $O(k)$. Both Equitable algorithms are based on a characterization in [44] in the context of *work functions*. The main idea is to define a probability distribution on the set of all possible configurations of the cache and ensure that the cache configuration obeys this distribution. For each request, it requires k probability computations, each taking $O(k)$ time. For a detailed view on paging algorithms, we refer the interested reader to the comprehensive surveys [2, 14, 34].

[1] In [11] Equitable2 is denoted A_k. Due to its similarity to Equitable we use its original name of Equitable2.

Our contributions. In this paper we propose a strongly competitive randomized paging algorithm, denoted OnlineMin. We first propose an implementation for it which handles a page request in $O(\log k)$ worst case time, and then we improve this implementation to achieve $O(\log k / \log\log k)$ time in the worst case for processing a page request. This is a significant improvement over the fastest known strongly competitive algorithm, Equitable, which needs $O(k^2)$ time per request[2]. The space requirement of both our implementations is $O(k)$, due to the forgiveness technique used in Equitable2.

The main building block of our algorithm is a priority based incremental selection process starting from the same characterization of an optimal solution in [44] as the Equitable algorithms. The analysis of this process yields a simple cache update rule which is different from the one in [1, 11], but leads to the same probability distribution of the cache content. A straightforward implementation of our update rule requires $O(k)$ time per request. Additionally we design appropriate data structures that result in two more efficient implementations: the first implementation uses simple pointer-based data structures to achieve $O(\log k)$ worst case time per page request, whereas the second implementation exploits the power of the RAM model to achieve $O(\log k / \log\log k)$ worst case time per request.

4.2 Randomized Selection Process

In this section we recall the notions of *offset functions* for paging algorithms introduced in [44]. We then describe in Section 4.2.2 a new priority based selection process which is the basis of our algorithm OnlineMin. We analyze the selection process in order to obtain a simple page replacement rule which remains at all times consistent with the outcome of the selection process. Finally, in Section 4.2.3 we prove equivalences between the cache distribution of our selection process and the Equitable algorithms [1, 11], which implies that OnlineMin is H_k-competitive.

4.2.1 Preliminaries

Let σ be the request sequence so far. For the construction of a competitive paging algorithm it is of interest to know the possible cache configurations if σ has been processed with minimal cost. We call these configurations *valid*.

For fixed σ and an arbitrary cache configuration C (a set of k pages), the *offset function* ω for σ assigns C the difference between the minimal cost of processing σ ending in configuration C and the minimal cost of processing σ. Thus C is a valid configuration after processing σ iff $\omega(C) = 0$. In [44] it was shown that the class

[2]Since no explicit implementation of Equitable2 is provided, due to their similarity we assume it to be the same as for Equitable.

of valid configurations $\mathcal{V}$ determines the value of ω on any configuration C by $\omega(C) = \min_{X \in \mathcal{V}}\{|C \setminus X|\}$.

Koutsoupias and Papadimitriou [44] showed that ω can be represented by a sequence of $k+1$ disjoint page sets $(L_0, L_1, \ldots, L_k)$, denoted *layers*, which can be constructed as follows[3]. Initially each layer L_i, where $i > 0$, consists of one of the first requested k pairwise distinct pages. The layer L_0 contains all pages not in $L_1, \ldots, L_k$. Since the offset function ω depends on the input sequence it has to be updated after each request. If ω is the offset function for input σ and page p is requested next, we denote by ω^p the offset function which results for the input σp and update the layers as follows[4]:

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

In [44] the relationship in Lemma 4.1 between the layer representation of ω and the class of valid configurations $\mathcal{V}$ was given.

Lemma 4.1 *If $(L_0, \ldots, L_k)$ is a layer representation of an offset function ω, then a set C of k pages is a valid configuration, i.e. $\omega(C) = 0$, iff $|C \cap (\cup_{i \le j} L_i)| \le j$ for all $0 \le j \le k$.*

We give an example of an offset function for $k = 3$ in Figure 4.1. The *support* of ω is defined as $S(\omega) = L_1 \cup \cdots \cup L_k$. In the remainder of the paper, we call a set with a single element *singleton*. Also, let u be the smallest index such that $L_{u+1}, \ldots, L_k$ are all singletons. We distinguish two sets: the set of *revealed* pages $R(\omega) = L_{u+1} \cup \cdots \cup L_k$, and the set of *unrevealed* pages $N(\omega) = L_1 \cup \cdots \cup L_u$. A valid configuration contains all revealed pages and no page from L_0. Note that when requesting some unrevealed page p in the support, we have $R(\omega^p) = R(\omega) \cup \{p\}$ and the number of layers containing unrevealed items decreases by one. Moreover, if $p \notin L_1$ then $N(\omega^p) = N(\omega) \setminus \{p\}$ and otherwise $N(\omega^p) = N(\omega) \setminus L_1$. Also, the layer representation is not unique and especially each permutation of the layers containing revealed items describes the same offset function.

Equitable, Equitable2 and the Forgiveness Technique. Given the layer representation of ω by the sequence requested so far, a probability distribution over all possible actual cache configurations was proposed in [1]. The probability that C is the cache content is defined as the probability of being obtained at the end of the following random process: Starting with $C = R(\omega)$ a page p is selected uniformly at random from $N(\omega)$, p is added to C, and ω is set to ω^p. This process

[3] We use a slightly modified, yet equivalent, version of the layer representation in [44].

[4] For easiness of exposition we refer by ω to both the offset function and its corresponding layer representation.

is iterated until C has k pages. A page replacement strategy that maintains this probability distribution under the constraints that 1) it replaces one page only upon a cache fault and 2) the cache content does not change upon a cache hit, was shown to be H_k-competitive [1] . The authors also provide the randomized algorithm EQUITABLE which handles a page request in $O(k^2)$ time and achieves the desired distribution under both constraints.

Note that by repeatedly requesting pages from L_0 the amount of pages in the support increases. In order to reduce the space requirements *forgiveness* techniques can be applied, which use an approximation of the offset function in order to cap the support size. The intuition behind these techniques is that a large support implies that the adversary did not play optimally and there is a large gap between the actual ratio and the worst case ratio of H_k. This gap cannot be closed by the adversary with future requests, and thus allows the online algorithm to deviate from the original layer update rule when tracking the offset function, while still preserving the H_k-competitiveness.

Given an offset function ω, both EQUITABLE and EQUITABLE2 have identical cache distributions. The difference consists in the forgiveness steps, more precisely they have different update rules for the offset function ω, when the support becomes too large. If the support size reaches a threshold EQUITABLE uses an approximation of the current offset function in order to bound the support size by $O(k^2 \log k)$. EQUITABLE2 uses an improved forgiveness step leading to space requirements of $O(k)$. More precisely whenever the support contains $3k$ pages and a page p is requested from L_0, EQUITABLE2 adds p to L_1 and handles the update of ω as p would have been requested from L_1.

Definition 4.1 *Given the current offset function $\omega = (L_0, \ldots, L_k)$ and the page request p, the update rule for ω including the forgiveness step of* EQUITABLE2 *is as follows*

$$\omega^p = \begin{cases} (L_0 \setminus \{p\}, L_1, \ldots, L_{k-2}, L_{k-1} \cup L_k, \{p\}) & \text{if } p \in L_0, |S(\omega)| < 3k, \\ (L_0 \setminus \{p\} \cup L_1, \ldots, L_{k-2}, L_{k-1}, L_k, \{p\}) & \text{if } p \in L_0, |S(\omega)| = 3k, \\ (L_0, \ldots, L_{i-2}, L_{i-1} \cup L_i \setminus \{p\}, L_{i+1}, \ldots, L_k, \{p\}) & \text{if } p \in L_i, i > 0. \end{cases}$$

Note that by the given update rule the layers $L_1, \ldots, L_k$ contain each at least one element. Thus in the case $|S(\omega)| = 3k$ the support size decreases by $|L_1| \geq 1$ and increases by 1 (the requested page p), and therefore we always have $|S(\omega)| \leq 3k$. In [11] it was shown that applying this update rule for ω still leads to a competitive ratio of H_k.

4.2.2 Selection Process for OnlineMin

If ω is the offset function for the input requested so far, an online algorithm should have a configuration similar to the cache C_{OPT} of an optimal strategy.

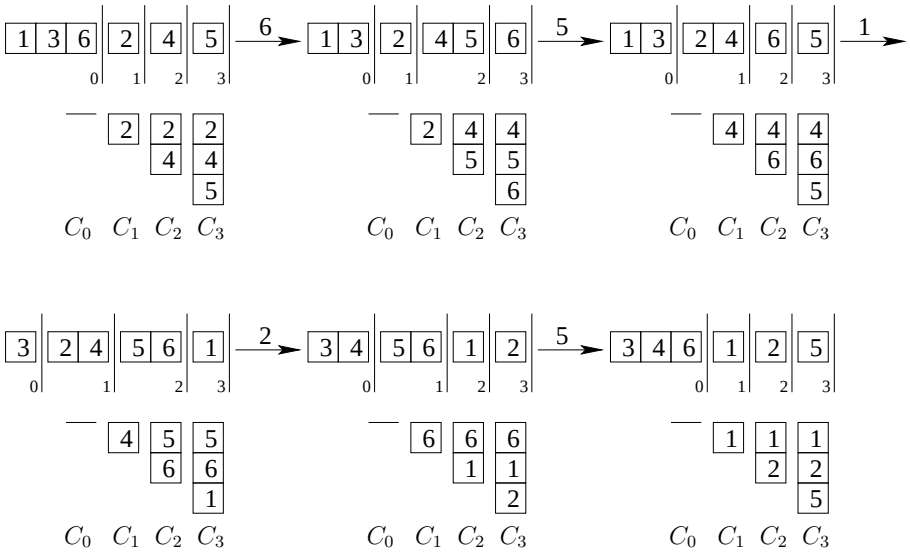


Figure 4.1: Example for updating the layers $L_0, \ldots, L_k$ and the selection sets $C_0, \ldots, C_k$ for $k = 3$. The initial cache configuration is $\{2, 4, 5\}$. The request sequence is $(6, 5, 1, 2, 5)$ and the priority of a page is its number.

We know that C_{OPT} contains all revealed items and no item from L_0. Which non-revealed items are in the cache depends on future requests. To guess the order of future requests of non-revealed items OnlineMin assigns priorities to pages when they are requested. It maintains the cache content of an optimal offline algorithm under the assumption that the priorities reflect the order of future requests. We introduce a priority based selection process for the layer representation of ω. Assuming that each order of priorities has equal probability, we prove that the outcome of the selection process has the same probability distribution as the Equitable algorithms. Our approach allows an efficient and easy-to-implement update method for the cache of OnlineMin, which is consistent with our selection process.

In the following we assume that pages from $L_1, \ldots, L_k$ have pairwise distinct priorities. For some set S we denote by $\min_j(S)$ and $\max_j(S)$ the subset of S of size j having the smallest and largest priorities respectively. Furthermore, $\min(S) = \min_1(S)$ and $\max(S) = \max_1(S)$.

Definition 4.2 *We construct iteratively* $k+1$ selection sets $C_0(\omega), \ldots, C_k(\omega)$ *from the layer partition* $\omega = (L_0, \ldots, L_k)$ *as follows. We set* $C_0(\omega) = \emptyset$ *and for* $j = 1, \ldots, k$ *we set* $C_j(\omega) = \max_j(C_{j-1}(\omega) \cup L_j)$.

When ω is clear from the context, we let $C_i = C_i(\omega)$. For a page request p and offset function $\omega = (L_0, \ldots, L_k)$, denote $\omega^p = (L'_0, \ldots, L'_k)$ and let C'_k be the

result of the selection process on ω^p. By the layer update rule each layer contains at least one element and the following result follows immediately.

Fact 4.1 *$|C_j| = j$ for all $j \in \{0, \ldots, k\}$. If $|L_j|$ is singleton then $C_j = C_{j-1} \cup L_j$. Moreover, all revealed pages are in C_k.*

Updating C_k. We analyze how C_k changes upon a request. First we give an auxiliary result in Lemma 4.2 and then show in Theorem 4.1 that C'_k can be obtained from C_k by at most one page replacement. We get how C'_k can be directly constructed from C_k and the layers, without executing the whole selection process.

Lemma 4.2 *Let p be the requested page from layer L_i, where $0 < i < k$. If for some j, with $i \leq j < k$ we have $q \in C_j$ and $C'_{j-1} = C_j \setminus \{q\}$, then we get*

$$C'_j = \begin{cases} C_{j+1} \setminus \{q\} & \text{if } q \in C_{j+1} \ , \\ C_{j+1} \setminus \min\{C_{j+1}\} & \text{otherwise} \ . \end{cases}$$

Proof. We have:

$$C'_j = \max_j (L'_j \cup C'_{j-1}) = \max_j (L_{j+1} \cup C_j \setminus \{q\}) = C_{j+1} \setminus \{q\} \quad \text{(case: } q \in C_{j+1}\text{)}$$
$$C'_j = \max_j (L'_j \cup C'_{j-1}) = \max_j (L_{j+1} \cup C_j \setminus \{q\}) = \max_j (C_{j+1}) \quad \text{(case: } q \notin C_{j+1}\text{)}$$

In both cases, we first use the assumption $C'_{j-1} = C_j \setminus \{q\}$ and the partition update rule $L'_j = L_{j+1}$. In the case $q \in C_{j+1}$ we use $C_{j+1} = \max_{j+1}(L_{j+1} \cup C_j) = \max_j (L_{j+1} \cup C_j \setminus \{q\}) \cup \{q\}$, which holds as $q \in C_j$ implies $q \notin L_{j+1}$. If $q \notin C_{j+1}$, we use $C_{j+1} = \max_{j+1}(L_{j+1} \cup C_j) = \max_{j+1}(L_{j+1} \cup C_j \setminus \{q\})$. We have $q \in C_j$, $q \notin C_{j+1}$ and $|C_{j+1}| = j+1$, which leads to $C'_j = \max_j (C_{j+1}) = C_{j+1} \setminus \min\{C_{j+1}\}$. □

Theorem 4.1 *Let p be the requested page. Given C_k, we obtain C'_k as follows:*

1. *$p \in C_k$: $C'_k = C_k$*
2. *$p \notin C_k$ and $p \in L_0$: $C'_k = C_k \setminus \min(C_k) \cup \{p\}$*
3. *$p \notin C_k$ and $p \in L_i$, $i > 0$: $C'_k = C_k \setminus \min(C_j) \cup \{p\}$, and $j \geq i$ is the smallest index with $|C_j \cap C_k| = j$.*

Before the proof, we note for the third case that the constraint $|C_j \cap C_k| = j$ means that all pages in C_j are also in C_k. While in general this constraint does not hold for all j, it is satisfied for all layers containing revealed pages (and the rightmost layer containing unrevealed pages) and thus such a j always exists.

Moreover, $|C_j \cap C_k| = j$ is equivalent to $|(L_1 \cup \cdots \cup L_j) \cap C_k| = j$, since C_j has elements only in $L_1 \cup \cdots \cup L_j$ and $C_j \subseteq C_k$.

Proof. First assume that $p \in L_0$. In this case, by construction p is not in C_k. The only layers that change are L_{k-1} and L_k: $L'_{k-1} = L_{k-1} \cup L_k$ and $L'_k = \{p\}$. Applying the definition of C'_k, the fact that $C_k = \max_{k-1}(C_{k-2} \cup L_{k-1}) \cup L_k$, and L_k is singleton, we get

$$C'_k = C'_{k-1} \cup \{p\} = \max_{k-1}(C_{k-2} \cup L_{k-1} \cup L_k) \cup \{p\} = C_k \setminus \min(C_k) \cup \{p\} .$$

Now we consider the case when $p \in L_i$. We distinguish two cases: $p \in C_k$ and $p \notin C_k$. If $p \in C_k$, we have by construction that p is in all sets $C_i, \ldots, C_k$ and we get $C_i = \max_i(C_{i-1} \cup L_i) = \max_{i-1}(C_{i-1} \cup L_i \setminus \{p\}) \cup \{p\}$. Based on this observation we show that $C'_{i-1} = C_i \setminus \{p\}$. It obviously holds for $i = 1$ since C'_0 is empty. If $i > 1$ we get

$$C'_{i-1} = \max_{i-1}(C_{i-2} \cup L_{i-1} \cup L_i \setminus \{p\}) = \max_{i-1}(C_{i-1} \cup L_i \setminus \{p\}) = C_i \setminus \{p\} .$$

Using $C'_{i-1} = C_i \setminus \{p\}$ and $p \in C_i$, applying Lemma 4.2 we get $C'_i = C_{i+1} \setminus \{p\}$. Furthermore, using that p is in all sets $C_{i+1}, \ldots, C_k$, we apply Lemma 4.2 for all these sets which leads to $C'_{k-1} = C_k \setminus \{p\}$ and we obtain $C'_k = C'_{k-1} \cup \{p\} = C_k$.

Now we assume that $p \notin C_k$. This implies that p is a non-revealed page. First we analyze the structure of C'_{i-1} which will serve as starting point for applying Lemma 4.2. If $p \in C_i$ we argued before that $C'_{i-1} = C_i \setminus \{p\}$. Otherwise, we show that $C'_{i-1} = C_i \setminus \min(C_i)$. It holds for $i = 1$ since C_0 is always empty and by Fact 4.1 we have $|C_1| = 1$. If $i > 1$ we get:

$$C'_{i-1} = \max_{i-1}(C_{i-2} \cup L_{i-1} \cup L_i \setminus \{p\}) = \max_{i-1}(C_{i-1} \cup L_i \setminus \{p\}) = C_i \setminus \min(C_i) .$$

Let $j \geq i$ be the smallest index such that $|C_j \cap C_k| = j$. By construction, we have $C_j \subseteq C_k$. Applying Lemma 4.2 for sets $C'_{i-1}, \ldots, C'_{j-1}$ we get $C'_{j-1} = C_j \setminus \{s\}$, where $s \in C_j$ and either $s = p$, $s = \min C_j$, or s is a page with minimal priority from a set C_l, with $i \leq l < j$. Note that page s is also in C_k by the definition of C_j and thus $s = p$ can be excluded since p is not in C_k. If s is a page with minimal priority from some set C_l then all the other pages in C_l are also in C_j and thus in C_k because all of them have higher priorities than s. This leads to $C_l \subset C_k$ which contradicts the minimality of j. Thus we have $s = \min C_j$. Since the page $s = \min(C_j)$ is in all sets $C_j, \ldots, C_k$ by Lemma 4.2 we get $C'_{k-1} = C_k \setminus \min(C_j)$ and it follows that $C'_k = C_k \setminus \min(C_j) \cup \{p\}$. □

4.2.3 Probability Distribution of C_k

Theorem 4.2 *Assume that non-revealed pages are assigned priorities such that the order of the priorities is distributed uniformly at random. For any offset*

function ω, the distribution of C_k over all possible cache configurations is the same as the distribution of the cache configurations for the Equitable *algorithms.*

Proof. Let u be the index of the last non-revealed layer, more precisely $|L_u| > 1$ and $|L_i| = 1$ for all $i > u$. The set of non-revealed items is $N(\omega) = L_1 \cup \cdots \cup L_u$ and the singletons $L_{u+1}, \ldots, L_k$ contain the revealed items $R(\omega)$.

The following selection process is used by both Equitable and Equitable2 to obtain the probability distribution of the cache M. Initially M contains all $k - u$ revealed items $R(\omega)$. Then u elements $x_1, \ldots, x_u$ are added to M, where x_i is chosen uniformly at random from the set of non-revealed items of $\omega^{x_1,\ldots,x_{i-1}}$, the offset function obtained from ω after requesting the sequence $x_1, \ldots, x_{i-1}$.

We define an auxiliary selection $C_k^*(\omega)$ which is a priority based version of Equitable's random process and then prove for every fixed priority assignment that $C_k(\omega) = C_k^*(\omega)$ holds.

Assume that pages in $N(\omega)$ have pairwise distinct priorities, with a uniformly distributed priority order. Initialize $C_k^*(\omega)$ to $R(\omega)$ and add elements $x_1^*, \ldots, x_u^*$ to $C_k^*(\omega)$, where x_i^* is the page with maximal priority from the non-revealed items of $\omega^{x_1^*,\ldots,x_{i-1}^*}$. Obviously all pages from $N(\omega)$ have the same probability to possess the maximal priority and thus x_1^* and x_1 have the same distribution. Since x_1^* is a revealed item in $\omega^{x_1^*}$, the priority order of pages in $N(\omega^{x_1^*})$ remains uniformly distributed. This implies inductively that $C_k^*(\omega)$ has the same distribution as Equitable. Note that by the definition of C_k^* we have $C_k^*(\omega) = C_k^*(\omega^{x_1^*})$ because x_1^* becomes a revealed item in $\omega^{x_1^*}$.

Now we prove for each fixed priority assignment that $C_k(\omega) = C_k^*(\omega)$ by induction on u. For $u = 0$ both C_k^* and C_k contain all k revealed items. For $u \geq 1$, let x_1^* be the non-revealed page with the largest priority in ω. For the auxiliary process, we have already shown that $C_k^*(\omega) = C_k^*(\omega^{x_1^*})$. Also, the index u for $\omega^{x_1^*}$ is smaller by one than for ω, which by inductive hypothesis leads to $C_k^*(\omega) = C_k^*(\omega^{x_1^*}) = C_k(\omega^{x_1^*})$. It remains to prove that $C_k(\omega^{x_1^*}) = C_k(\omega)$. By the definition of the selection process for $C_1, \ldots, C_k$ we have $C_k(\omega) = C_u(\omega) \cup R(\omega)$. Page x_1^* has the highest priority from $N(\omega) = L_1 \cup \cdots \cup L_u$ and thus it is a member of $C_u(\omega)$ and hence also in $C_k(\omega)$. Applying the update rule from Theorem 4.1 we get $C_k(\omega) = C_k(\omega^{x_1^*})$, and this concludes the proof. □

4.3 Algorithm OnlineMin

4.3.1 Algorithm

OnlineMin initially holds in its cache M the first k pairwise distinct pages. Note that the timestamp of the last request for any page in L_i is smaller than the timestamp of the last request for any page in L_{i+1}.

Page Replacement. The algorithm maintains the invariant that $M = C_k$ after each request. To do so, it keeps track of the layer partition $\omega = (L_0, \ldots, L_k)$ according to Definition 4.1 (including the forgiveness step of Equitable2), where it suffices to store only the support layers $(L_1, \ldots, L_k)$. The cache update is performed according to Theorem 4.1[5]. More precisely, if the requested page p is in the cache, M remains unchanged. If a cache miss occurs and p is from L_0 the page with minimal priority from M is replaced by p. If p is from L_i with $i > 0$, and $p \notin M$ we first identify the set C_j in Theorem 4.1 satisfying $|C_j \cap M| = j$. This can be done as follows. Let $p_1, \ldots, p_k$ be the pages in M sorted in increasing order by their layer index. We search the minimal index $j \geq i$, such that the condition that the layer index of p_j is j, i.e. $p_j \in L_j$, is satisfied (index j is guaranteed to exist, since the condition holds for all revealed pages and the rightmost unrevealed page). We evict the page with minimal priority from $p_1, \ldots, p_j$. The layers are updated after the cache update.

Priorities. To assign priorities, we develop a data structure which maintains a dynamic random *ordered* set P of integers. We require at all times that the ranks of numbers in P correspond to an (equally distributed) random permutation of $\{1, \ldots, |P|\}$. Under the assumption that the size of P is bounded by a number u, we require $|P|$ to support two operations: *expand*, which adds a new element to P, and *delete*(x) which removes element x from P.

We use the universe $U = \{1, \ldots, u\}$ for numbers in P. We start with $P = \emptyset$. Upon an expand operation we choose an element uniformly at random from $U \setminus P$ and insert it in P. Upon a delete operation, the element to delete is simply removed from P. In particular, the expand operation corresponds to a step in the Fisher-Yates shuffle algorithm (original method) [35]. They showed that applying u expand operation results in a random permutation of U. In Lemma 4.3 we show that the two operations can be implemented efficiently and that the ranks of elements in P form a random permutation.

Lemma 4.3 *The ranks of the elements in P represent a random permutation of $\{1, \ldots, |P|\}$. The data structure can be implemented in $O(1)$ time per request and uses $O(u)$ space.*

Proof. Let $e_x = (e_1, \ldots, e_x)$ be a random variable describing the ranks in the sequence after x consecutive expand operations. Since after u expand operation the Fisher-Yates shuffle yields a random permutation. Thus, e_u is a random permutation of U and it follows that e_x is a random permutation of $\{1, \ldots, x\}$. If e_x describes the ranks in P, we get upon an expand operation the distribution e_{x+1} and upon a delete operation the distribution e_{x-1}. We conclude that ranks in P correspond to $e_{|P|}$ and thus a random permutation of $\{1, \ldots, |P|\}$.

[5]Theorem 4.1 does not explicitly take into account the forgiveness step. According to Definition 4.1, if $p \in L_0$ and forgiveness is applied we treat p as if it was requested in L_1.

The data structure can be implemented using an array A of size u and a list L of size $|P|$. We store the elements from P in the first $|P|$ locations of A and the rest of A contains all elements from $U \setminus P$. The element order is given by the list L. We further assume direct access to the position of any element $i \in U$ in A and in L using additional $O(u)$ space. Initially we set $A[i] = i$ and $L = \emptyset$. Upon an expand operation we choose a random index r in the range $[|P| + 1, u]$ and append $A[r]$ to L. To reflect this in A we switch the contents of $A[|P| + 1]$ and $A[r]$. The deletion of element x is similar, we first look up the index i_x of x in A and swap $A[i_x]$ and $A[|P|]$. We further delete x from L. □

By the forgiveness mechanism the support size is at all times $O(k)$ and thus priorities can be maintained using the data structure previously introduced. When a page enters the support we assign it a priority using the expand operation, and when it leaves the support we use the delete operation. This takes $O(k)$ space and $O(1)$ time per page request (at most one page is assigned a new priority at each request) by setting u to the maximal support size which is guaranteed by the forgiveness technique to be at most $3k$.

Time and Space Complexity. Storing the layer partition together with the page priorities needs $O(k)$ space by applying the forgiveness mechanism of Equitable2 [11]. A naive implementation storing the layers in an array processes a page request in $O(k)$ time. In the remainder of the paper we will improve this naive bound first to $O(\log k)$ worst case time per request using simple pointer-based data structures and then to $O(\log k / \log \log k)$ time per request using data structures in the RAM model.

Competitive Ratio. We showed in Theorem 4.2 that the probability distribution over the cache configurations for OnlineMin and Equitable2 are the same. This holds also when using the forgiveness step, and thus the two algorithms have the same expected cost. This leads to the result in Lemma 4.4.

Lemma 4.4 OnlineMin *is H_k-competitive.*

4.3.2 Algorithm Implementation

In this section we show that OnlineMin can be implemented using a sorted list augmented with a series of specific operations. We will later focus only on giving data structures supporting these operations.

Basic Structure. In the following we represent each page in the support by the timestamp of its last request. Consider a list $L = (l_1, \ldots, l_t)$, with $t \leq 4k$, where L has two types of elements: k *layer delimiters* and at most $3k$ *page elements*. Furthermore, we distinguish two types of page elements: *cache elements* which

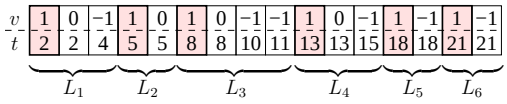


Figure 4.2: Example for list L: representing pages by timestamps of last requests, we have $L_1 = \{2, 4\}$, $L_2 = \{5\}$, $L_3 = \{8, 10, 11\}$, $L_4 = \{13, 15\}$, $L_5 = \{18\}$, and $L_6 = \{21\}$. Layer delimiters are emphasized and the memory content is $M = \{4, 10, 11, 15, 18, 21\}$.

are the pages in the cache and *support elements* which are pages in the support but not in the cache. We store in L the layers $L_1, \ldots, L_k$ from left to right, separated by k layer delimiters. For each layer L_i we store its layer delimiter, followed by the pages in L_i. For each list element l_i, be it page element or layer delimiter, we store a timestamp t_i and a *v-value* v_i with $v_i \in \{-1, 0, 1\}$; for page elements we also store the priority. For some element l_i, if it is a layer delimiter for some layer L_j, we set $v_i = 1$ and t_i to the minimum of all page timestamps in L_j. If l_i is a page element, then t_i is set to the timestamp corresponding to the last request of the page; we set $v_i = -1$ for cache elements and $v_i = 0$ for support elements. Note that the layer delimiters always have t_i values matching the first page in their layer. As described before, layer delimiters always precede page elements. An example is given in Figure 4.2.

Note that the v-values have the property that $|C_k \cap (L_1 \cup \cdots \cup L_i)| = i$ iff the prefix sum of the v-values for the last element in L_i is zero. Furthermore, since $|C_k \cap (L_1 \cup \cdots \cup L_i)| \leq i$ the prefix sum cannot be negative. This property will be used when dealing with a cache miss caused by a page from L_i, with $i > 0$.

We show how to implement OnlineMin using the following operations on L:

- `find-layer`(l_p). For some page l_p, find its layer delimiter.
- `search-page`(l_p). Check whether l_p is a page in L.
- `insert`(l_p), `delete`(l_p). The item l_p is inserted (or deleted) in L.
- `find-min-prio`(l_p). Find the cache element $l_q \in (l_1, \ldots, l_p)$ with minimum priority.
- `find-zero`(l_p). Find the smallest j, with $p \leq j$ such that $\sum_{l=1}^{j} v_l = 0$, and return l_j.

We note that the prefix sum cannot be negative, and thus for the `find-zero` operation it suffices to find the first element to the right having the minimum prefix sum. For this reason, we refer to `find-zero` also as `find-pref-min`.

We describe how to update the list L upon a request for some page p. OnlineMin keeps in memory at all times the elements in L having the v-value equal to -1.

If $p \notin M$, we must identify a page to be evicted from M. To evict a page we set its v-value to zero and to load a page we set its v-value to -1. We first find the layer delimiter for p. We can have $p \in L_i$ with $0 < i \leq k$ or $p \in L_0$. If $p \in L_i$, the page to be evicted is the cache element in $L_1 \cup \cdots \cup L_j$ having the minimum priority, where $j \geq i$ is the minimal index satisfying $|M \cap (L_1 \cup \cdots \cup L_j)| = j$. This is done using `find-zero`(l_{L_i}), where l_{L_i} is the layer delimiter of L_i, and the page to be evicted is identified using `find-min-prio` applied to the value returned by `find-zero`. If $p \in L_0$ and forgiveness need not be applied, the page having the smallest priority in M is to be evicted. We identify this page in L using `find-min-prio` on the last element in L. If we must apply forgiveness, we treat p as being a support page in L_1.

After updating the cache, we perform in L the layer updates as follows. If $p \in L_i$ with $i > 0$, the layers are updated as follows: $L_{i-1} = L_{i-1} \cup L_i \setminus \{p\}$, $L_j = L_{j+1}$ for all $j \in \{i, \ldots, k-1\}$, and $L_k = \{p\}$. We first delete the layer delimiter for L_i and the page element for p, which triggers not only the merge of L_{i-1} and $L_i \setminus \{p\}$, but also shifts all the remaining layers, i.e. $L_j = L_{j+1}$ for all $j \geq i$. If we deleted the layer delimiter for L_1, we also delete all pages in L_1 because in this case L_1 is merged with L_0. To create $L_k = \{p\}$, we simply insert at the end a new layer delimiter followed by p, both items having as timestamp the current timestamp.

If $p \in L_0$, we first check whether we must apply the forgiveness step, and if so we apply it by simulating the insertion of p in L_1 and then requesting it, as described in Definition 4.1. If forgiveness need not be applied, we update the layers $L_{k-1} = L_{k-1} \cup L_k$ and $L_k = \{p\}$ as follows. We first delete the layer delimiter of L_k, which translates into merging L_{k-1} and L_k. Then, we insert a new layer delimiter having the timestamp of the current request, i.e. create L_k, and insert p with the same timestamp.

We note that while the priority of each page takes $O(\log k)$ space, the timestamp for its last request takes $O(\log n)$ space. We reduce the space requirement to $O(\log k)$ by simply resetting the timestamps for pages and layers in support after $O(k)$ operations, setting the new timestamps to $1, \ldots, |S|$, where $|S|$ is the support size. The new timestamps are assigned in a left-to-right manner, thus ensuring that the relative order of the new timestamps reflect the old order. Since by the forgiveness mechanism we have at all times $|S| = O(k)$, it follows that $O(1)$ amortized time per page request is required. We further deamortize this by resetting only c timestamps per page request, in a left-to-right manner, where c is a constant with $c > 2$; it is necessary to have $c > 2$ because at each request two elements (i.e. the rightmost set delimiter and page) receive new timestamps according to the current (large) timestamp, and we need to ensure that more pages receive updated (small) timestamps. At the end of the day, in $O(1)$ worst case time per request we ensure that the timestamps take also $O(\log k)$ space.

4.3.3 Pointer-Based Structures

We implement all the operations previously introduced using two data structures: a *set structure* and a *page-set* structure. The set structure focuses only on the `find-layer` operation, and the page-set data structure deals with the remaining operations. While most operations can be implemented using standard data structures, i.e. balanced binary search trees, the key operation for the page-set structure is `find-zero`. That is because we need to find in sublinear time the first item to the right of an arbitrary given element having the prefix sum zero in the presence of updates, and the item that is to be returned can be as far as $\Theta(k)$ positions in L.

Set Structure. The set structure is in charge only for the `find-layer` operation. To do so, it must also support updating the layers. It is a classical balanced binary search tree, e.g. an AVL tree, built on top of the layer delimiters in L having as keys the timestamps of the delimiters. Whenever a layer delimiter is inserted or deleted from L, the set structure is updated accordingly. Each operation takes $O(\log k)$ time in the worst case.

Page-set Structure. It contains all elements of L and supports all the remaining operations required on L. We store the elements of L in the leaves of a regular leaf oriented balanced binary search tree indexed by the timestamps. For some node u, denote by $\mathcal{T}(u)$ the subtree rooted at u and by $\mathcal{L}(u)$ the leaves of $\mathcal{T}(u)$. To deal with the priorities, at each node u we store the minimum priority $u.min\_p$ of the cache pages in $\mathcal{L}(u)$ and a pointer $u.idx\_min\_p$ to the leaf having this priority; if no cache pages exist, $u.min\_p$ is set to infinity and $u.idx\_min\_p$ to null. For handling the v-values we store at each node u the sum $u.sum$ of the v-values stored in $\mathcal{L}(u)$ and the minimum prefix sum $u.pref\_min$ of the v-values restricted on the elements of $\mathcal{L}(u)$. More precisely, if $\mathcal{L}(u) = (p_1, \ldots, p_m)$, we have $u.pref\_min = \min_{l=1}^{m}(\sum_{j=1}^{l} p_j.v)$. Also, we store a pointer $u.idx\_pref\_min$ to the leaf having the prefix sum $u.pref\_min$. All the data stored at each node is shown in Figure 4.3.

In the following fact it is shown that all values stored at each node can be computed bottom up in $O(1)$ time per node.

Fact 4.2 *For each node u with children $u.left$ and $u.right$, the following hold:*

- $u.min\_p = \min\{u.left.min\_p, u.right.min\_p\}$ *and $u.idx\_min\_p$ is either $u.left.idx\_min\_p$ or $u.right.idx\_min\_p$ depending on the origin of $u.min\_p$.*
- $u.pref\_min = \min\{u.left.pref\_min, u.left.sum + u.right.pref\_min\}$ *and $u.sum = u.left.sum + u.right.sum$. We also have that $u.idx\_pref\_min$ is either $u.left.idx\_pref\_min$ or $u.right.idx\_pref\_min$ depending on the origin of the minimum prefix sum computed.*

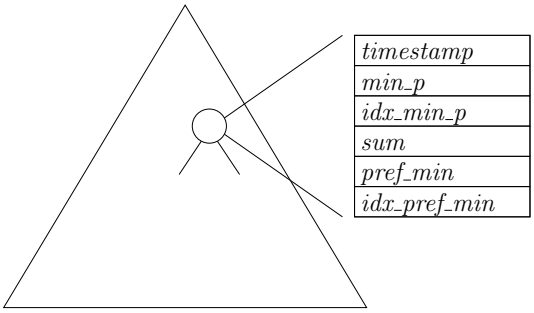


Figure 4.3: The additional information stored at each node in the page-set structure.

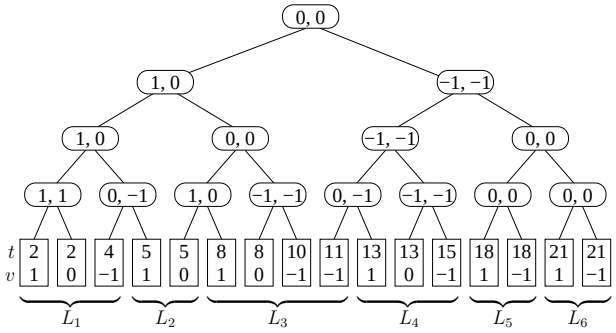


Figure 4.4: The page-set data structure for $L_1 = \{2, 4\}$, $L_2 = \{5\}$, $L_3 = \{8, 10, 11\}$, $L_4 = \{13, 15\}$, $L_5 = \{18\}$, and $L_6 = \{21\}$, and the memory image $M = \{4, 10, 11, 15, 18, 21\}$. For each internal node u the $(u.sum, u.pref\_min)$ values are shown.

Updates. We discuss how to perform insertions and deletions in the page-set structure. To insert an element, we create a leaf for the new element and an internal node, and update the information stored at each internal node on the path to the root as described in Fact 4.2. For the $O(1)$ nodes per level involved in rotations due to rebalancing we also recompute these values. Deleting an element in the page-set structure is done analogously to insertion. We note however that when requesting a page in L_1 we must delete both the layer delimiter and all page elements in L_1 from the data structure which leads to $O(\log k)$ amortized time. We will show later how to improve this bound to $O(\log k)$ worst case time for deletions as well.

Queries. We turn to queries supported by the page-set structure, which are the queries required on L. The `search-page` operation is implemented using a standard search in a leaf-oriented binary search tree.

For `find-min-prio` we find the page element having the minimum priority in $l_1, \ldots, l_p$ by traversing the path from l_p to the root in the page-set structure. For each node u on the path where l_p is in the right-subtree, consider its left child w. For all such nodes w identified and the leaf l_p, take the minimum over all $w.min\_p$, and return the corresponding $w.idx\_min\_p$ index. Since it does a bottom-up traversal, this operation takes $O(\log k)$ time in the worst case.

It remains to deal with the `find-zero` operation, where we are given some leaf storing l_p and must return the first leaf to the right which has the prefix sum of the v-values zero. We do so by traversing the path from l_p to the root. For each visited node u let s denote the sum of v-values of all elements in $\mathcal{L}(u)$ to the right of l_p. Also, let ps be the best prefix sum found so far and s_idx the corresponding leaf. Initially $s\_idx = l_p$ and $s = ps = l_p.v$. When advancing to the parent from the right children no action is required. When advancing to the parent node u from the left child we first check if $s + u.right.pref\_min < ps$ and if so we have found a better prefix sum, and update $ps = s + u.right.pref\_min$ and $s\_idx = u.right.idx\_pref\_min$. Finally we update $s = s + u.right.sum$. We return the identified leaf s_idx. This operation requires a bottom-up traversal of the tree and thus takes $O(\log k)$ time in the worst case.

Worst-case Bounds. The only operation taking $\omega(\log k)$ time is page deletion, more precisely when a page in L_1 is requested all pages in L_1 are moved to L_0 and thus should be removed from the support. Instead of deleting the set delimiter and all the pages corresponding to L_1, we delete only the set delimiter. With the leading set delimiter removed, the list L no longer starts with a set delimiter, but with at most $O(k)$ elements having the v-value set to 0, since all of these pages belong to L_0 and thus by Definition 4.2 cannot be cache elements. Also, these pages do not influence the prefix sums for the v-values. When we process a page, we simply start by checking if the leftmost element in the tree has a v-value of 0, and if so we delete it. Since each page request adds at most one new element to the support, the space complexity is still $O(k)$. This way deletions can be done in $O(\log k)$ time in the worst case.

Each page request uses $O(1)$ operations in both data structures. Theorem 4.3 summarizes the time and space complexities for this implementation of OnlineMin.

Theorem 4.3 OnlineMin *uses $O(k)$ space and processes a request in $O(\log k)$ time in the worst case.*

4.3.4 RAM Model Structures

In this section we provide an alternative implementation for OnlineMin which handles a page request in $O(\log k / \log\log k)$ time. In particular,we describe how the page-set structure operations can be implemented in $O(\log k / \log\log k)$ time

in the RAM model and argue that this is the best possible for an approach using the page-set interface defined in Section 4.3.2.

For the set structure, we use the data structure in [5] which supports updates and predecessor queries in $O(\sqrt{\log k / \log\log k})$ time while using $O(k)$ space. A data structure which supports the `search-page`, `insert`, `delete`, and `find-min-prio` operations in $O(\log k / \log\log k)$ time can be found in [65, Section 5], which adapts fusion tree ideas [36] to priority search trees [47].

In the following we modify the page-set structure previously introduced to support both `find-min-prio` and `find-zero` operations in $O(\log k / \log\log k)$ time. In particular, for the `find-min-prio` operation we borrow ideas from [65]. Instead of the balanced binary search tree in Section 4.3.3 we use a B-tree [8], where each node has degree at most $\Delta = \log^{\varepsilon} k$ for some $0 < \varepsilon \leq 1/4$.

Again, each node u stores $u.min\_p$, $u.idx\_min\_p$, $u.sum$ and $u.idx\_pref\_min$. Furthermore, to support the `find-min-prio` operation and to be able to update $u.min\_p$, each node u stores additionally the following:

- A Q-heap $u.Q$ over the $w.min\_p$ priorities at the children of u.
- A word $u.\pi$ storing for all the Δ children the rank of $u.child[i].min\_p$ among $u.child[1].min\_p, ..., u.child[\Delta].min\_p$. Therefore the number of bits required by $u.\pi$ is $\Delta \log \Delta = o(\log k)$.

The Q-heap of Fredman and Willard [37, Theorem, page 550], supports in $O(1)$ time insertions, deletions, predecessor queries (in particular min-queries), and rank queries (how many elements are smaller than a query element) on sets having at most $(\log k)^{1/4}$ elements. The data structure requires word size at least $\log k$ and time $O(k)$ to preprocess some global tables.

The Q-heap $u.Q$ allows us to update $u.min\_p$ in $O(1)$ time when the i'th child w gets a new $w.min\_p$ value, by deleting the old $w.min\_p$ value from $u.Q$, inserting the new $w.min\_p$ into $u.Q$, and querying $u.Q$ for the new minimum. When updating $u.min\_p$ we can also compute the new rank of $w.min\_p$ among the children of node u using $u.Q$, and update $u.\pi$ using a precomputed table: $T[old\_\pi, i, new\_rank] = new\_\pi$. Note that old_π, i, and new_rank in total only require $\Delta \log \Delta + \log \Delta + \log \Delta = o(\log k)$ bits, i.e. the table needs $k^{o(1)}$ precomputed entries. To perform `find-min-prio` we similarly traverse a leaf-to-root path. At each node u where we now come from the i'th child w, we identify the minimum of $u.child[1].min\_p, \ldots, u.child[i-1].min\_p$ by considering π only, which again can be answered using a precomputed table T, where $T[\pi, i]$ stores the index of the child having the minimum min_p value.

To support the `find-zero` operation we need to efficiently update and query the prefix sum fields. To this end, we consider the updates in $\mathcal{T}(u)$ in *phases*, where each phase consists of Δ updates below u. At each node u we store the following:

- A counter $u.count$ in the range $0 \dots \Delta - 1$, counting the number of updates below u since the start of the phase.
- A word $u.\tau$ storing an array of size Δ with $M[i] \in [0, \dots, \Delta(\Delta+2)]$ for all i; this means that $u.\tau$ can be stored in $\Delta(1 + 2\log\Delta) = o(\log k)$ bits.
- Two arrays m and PS, each of size Δ.
- A word $u.\varsigma$ storing an array dm of size Δ with $dm[i] \in [-\Delta, \dots, \Delta]$, i.e. $u.\varsigma$ can be stored in $\Delta \log(2\Delta + 1) = o(\log k)$ bits.

At the beginning of a phase for some node u we reset the information stored at u as follows. The arrays PS and m store the prefix sum and the minimum prefix sum for the children of u respectively, that is $PS[i] = \sum_{j=1}^{i-1} u.child[j].sum$ and $m[i] = u.child[i].pref\_min + PS[i]$. Also, we set $dm[i] = 0$ for all i. After a number of updates in a phase for $\mathcal{T}(u)$, $dm[i]$ is the value to be added to $PS[i]$ to get the correct values for $PS[i]$ and $m[i]$.

We let M be an approximation of m which maintains at all times during the phase the following invariant: for all i, if $m[j] + dm[j] = \min(m[i] + dm[i], \dots, m[\Delta] + dm[\Delta])$ then we have that $M[j] = \min(M[i], \dots, M[\Delta])$.

At the beginning of a phase we construct M for decreasing index i, while keeping track of the minimum $m[min]$ where $min > i$. Initially $min = u.degree$ and $M[u.degree] = \Delta^2$. For $i = u.degree - 1$ `downto` 1, we compute $M[i]$ as follows: if $m[i] \geq m[min] + \Delta$ then $M[i] = M[min] + \Delta$, and if $m[i] \leq m[min] - \Delta$, then $M[i] = M[min] - \Delta$; otherwise, $M[i] = M[min] + m[i] - m[min]$. If after computing $M[i]$ we have $M[i] < M[min]$ then we set $min = i$, see e.g. Figure 4.5. The key idea is that any update can only change the $pref\_min$ value of a node by at most one, since the v-values are in $\{-1, 0, 1\}$. Therefore, if for some i and j it holds that $|u.child[i].pref\_min - u.child[j].pref\_min| > \Delta$ at the beginning of the phase, then their relative order does not change during the Δ updates in the phase.

To analyze the range of the $M[i]$ values, we note that since $M[j]$ is decreasing during the construction any assigned value can be at most $M[u.degree] + \Delta$, i.e. $\Delta(\Delta + 1)$. Similarly, each $M[i] \geq M[i+1] - \Delta$, i.e. all $M[i] \geq M[u.degree] - (\Delta - 1)\Delta = \Delta^2 - \Delta^2 + \Delta = \Delta$. Since each update below u during a phase can change $M[i]$ by at most ± 1, it follows that during the Δ operations in a phase all $M[i]$ values are within the range $[0 \dots \Delta(\Delta + 2)]$.

During an update where $u.child[i].sum$ is incremented we increment all $M[j]$ and $dm[j]$ for $i < j \leq u.degree$; also, we increment $M[i]$ and $m[i]$ whenever $u.child[i].pref\_min$ is incremented. The case when $u.child[i].sum$ is decremented is treated similarly. The updating of M and dm can be done in $O(1)$ time using table lookup, i.e. $T1[old\_\tau, i] = new\_\tau$ and $T2[old\_\varsigma, i] = new\_\varsigma$. Since M is reconstructed only at the beginning of a phase, and M and dm can be constructed

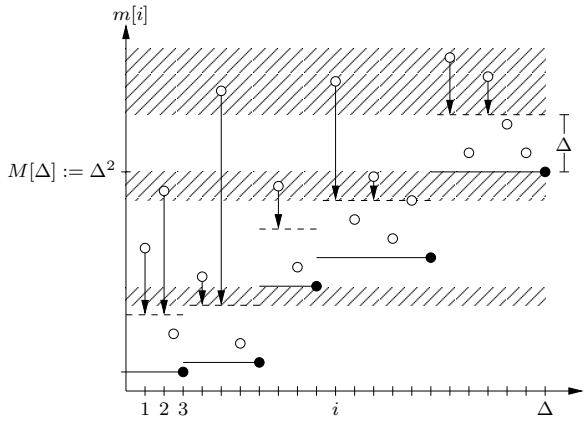


Figure 4.5: Illustration of the construction of M from m. Circles are the elements of m. For each i the solid line/circle shows the minimum m to the right of i. Elements that cannot become the answer during the next Δ updates are replaced by the elements pointed to by an arrow. The shaded area are the ranges of the domain removed in the transformation from m to M.

in $O(\Delta)$ time, it follows that the amortized cost to update M and dm for an update below u is amortized $O(1)$.

Updating the $u.sum$ values bottom-up during updates is done by adding (subtracting) the inserted (deleted) value along the leaf-to-root path. The child i storing the new $u.idx\_pref\_min = u.child[i].idx\_pref\_min$ can be extracted from $u.M$ (i.e. $u.\tau$) using a table lookup, and $v.pref\_min = m[i] + dm[i]$ (we compute the index of the minimum child before actually knowing the exact value).

A query, i.e. `find-zero`, is performed bottom-up as in Section 4.3.3, where we keep track of the minimum prefix sum s so far, except that when reaching node u from child i, we need to find the child j with minimum prefix sum among children $i+1, \ldots, degree(u)$. This can be extracted from $u.M$ using table lookup in $O(1)$ time. The minimum prefix sum for $u.child[j]$ is $PS[j] = u.m[j]+u.dm[j]$, which is compared to the prefix sum from $u.child[i]$ which is $PS[i] = s+u.PS[i]+v.dm[i]$; s becomes the minimum of $PS[i]$ and $PS[j]$. If this minimum is $PS[j]$, then idx_pref_min is updated to $u.child[j].idx\_pref\_min$.

Inserting a new leaf into the tree might cause a node to get more than Δ children, in which case we split the affected node into two nodes of degree at most $\Delta/2 + 1$ each. Whenever a node is split, gets a new child, or loses a child, we recompute all the information at the node in $O(\Delta)$ time. This way the total number of node splits is bounded by $O(\#insertions/\Delta)$ and it follows that the total cause for splitting and inserting/deleting leaves is $O(\#insertions \cdot \Delta)$, i.e. amortized $O(\Delta)$ per update. To avoid the height of the tree to exceed $O(\log_\Delta k)$, we globally rebuild the tree and all the associated information from scratch in $O(k)$ time whenever half of the leaves inserted into the tree have been deleted.

To achieve worst case bounds we use standard deamortization techniques. We perform the node splitting and the computation of the appropriate values at the beginning of each phase incrementally. We simply save the state of the node and at each update we perform $O(1)$ computations such that after Δ updates the updated values in the given node are computed. Similarly, the global rebuilding can be done incrementally as well, which yields $O(1)$ at each level of the tree for all operations. Since the tree has height $O(\log_\Delta k)$ where $\Delta = \log^\varepsilon k$, it follows that all operations can be implemented in $O(\log k / \log\log k)$ worst case time.

The following results stems from the fact that each page request uses a constant number of operations on the previously introduced data structure.

Theorem 4.4 *A page request can be done in $O(\log k / \log\log k)$ time while using $O(k)$ space.*

Lower Bounds The following cell-probe (RAM) lower bounds (using words of $\log k$ bits) state that for the page-set structure we cannot achieve better than $\Omega(\log k / \log\log k)$ query time with polylogarithmic deletion bounds. According to [4, Proposition 2] (note after proposition about decremental priority searching) any data structure supporting `delete` and `find-min-prio` requires $\Omega(\log k / \log\log k)$ time for polylogarithmic deletion time. We note that the given lower bound applies to the page-set structure in particular and not to the paging problem in general, not even to the approach taken by OnlineMin. Nonetheless, they show that to process a page request in $o(\log k / \log\log k)$ time any implementation must exploit some particular characteristics of OnlineMin.

Acknowledgements

We would like to thank previous anonymous reviewers for very insightful comments and suggestions. Also, we would like to thank Annamária Kovács for useful advice on improving the presentation of the paper.

Chapter 5

Improved Space Bounds for Strongly Competitive Randomized Paging Algorithms

The work **Improved Space Bounds for Strongly Competitive Randomized Paging Algorithms** was published as a technical report [52] and was accepted as a conference paper [50] at ICALP 2013.

[52] G. Moruz and A. Negoescu. Improved space bounds for strongly competitive randomized paging algorithms. Technical report, Goethe-Universität Frankfurt am Main, 2013

[50] G. Moruz and A. Negoescu. Improved space bounds for strongly competitive randomized paging algorithms. In *Proc. 40th International Colloquium on Automata, Languages, and Programming, ICALP 2013 (to appear)*

The contents of this chapter correspond to the technical report [52] which is the full version of the conference paper [50]. The technical report contains the same results as the conference paper. In addition to the contents of the conference paper it contains proofs for:

- Lemmas 5.1-5.4, 5.7, 5.9
- Theorems 5.3, 5.4

which were omitted due to space limitation.

Improved Space Bounds for Strongly Competitive Randomized Paging Algorithms

Gabriel Moruz, Andrei Negoescu

Abstract

Paging is one of the prominent problems in the field of online algorithms. While in the deterministic setting there exist simple and efficient strongly competitive algorithms, in the randomized setting a tradeoff between competitiveness and memory is still not settled. Bein et al. [10] conjectured that there exist strongly competitive randomized paging algorithms, using $o(k)$ bookmarks, i.e. pages not in cache that the algorithm keeps track of. Also in [10] the first algorithm using $O(k)$ bookmarks ($2k$ more precisely), EQUITABLE2, was introduced, proving in the affirmative a conjecture in [14].

We prove tighter bounds for EQUITABLE2, showing that it requires less than k bookmarks, more precisely $\approx 0.62k$. We then give a lower bound for EQUITABLE2 showing that it cannot both be strongly competitive and use $o(k)$ bookmarks. Nonetheless, we show that it can trade competitiveness for space. More precisely, if its competitive ratio is allowed to be $(H_k + t)$, then it requires $k/(1 + t)$ bookmarks.

Our main result proves the conjecture that there exist strongly competitive paging algorithms using $o(k)$ bookmarks. We propose an algorithm, denoted PARTITION2, which is a variant of the PARTITION algorithm by McGeoch and Sleator [48]. While classical PARTITION is unbounded in its space requirements, PARTITION2 uses $\Theta(k/\log k)$ bookmarks. Furthermore, we show that this result is asymptotically tight when the forgiveness steps are deterministic.

5.1 Introduction

Paging is a prominent and well studied problem in the field of online algorithms. We are provided with a two-level memory hierarchy consisting of a fast cache which can accommodate k pages, and a slow memory of infinite size. The input consists of requests to pages which are processed as follows. If the currently requested page is in the cache, we say that a *cache hit* occurs and the algorithm proceeds to the next page. Otherwise, a *cache miss* occurs and the requested page must be brought into cache. Additionally, if the cache was full, a page in

cache must be evicted to accommodate the new one. The cost of the algorithm is given by the number of cache misses incurred.

Online algorithms in general and paging algorithms in particular are typically analyzed in the framework of *competitive analysis* [41,60]. An algorithm A is said to have a *competitive ratio* of c (or c-competitive) if its cost satisfies for any input $cost(A) \leq c \cdot cost(OPT) + O(1)$, where $cost(OPT)$ is the cost of an optimal offline algorithm, i.e. an algorithm which is presented with the input in advance and processes it optimally; for randomized algorithms, $cost(A)$ is the expected cost of A. An algorithm achieving an optimal competitive ratio is said to be *strongly competitive*. For paging, an optimal offline algorithm was proposed decades ago; upon a cache miss, it evicts the page in cache whose request occurs the furthest in the future [12]. In the remainder of the paper, we refer to this algorithm as OPT. For comprehensive surveys on online algorithms in general and paging algorithms in particular, we refer the interested reader to [2, 14].

Competitive ratio has been often criticized for its too pessimistic quality guarantees. Especially in the deterministic setting, the empirically measured performance for practical algorithms is far below the theoretical guarantee of k provided by competitive analysis [68]. This gap is significantly smaller for randomized algorithms, since the best possible competitive ratio is H_k. Although using only the quality guarantees provided by competitive analysis is a naive way to distinguish good paging algorithms from bad ones, we have shown in [51] that ideas from competitive analysis for randomized algorithms can be successfully employed to design algorithms with good performance on real-world inputs. That is because an optimal randomized algorithm can be viewed as a collection of reasonable deterministic algorithms, and the algorithm designer can simply look for suitable algorithms in this collection.

Related Work. Randomized competitive paging algorithms have been extensively studied over the past two decades. In [32] a lower bound of H_k on the competitive ratio of randomized paging algorithms has been given[1]. Also in [32], a simple $(2H_k - 1)$-competitive algorithm, denoted MARK, has been proposed. In [23] it was shown that no randomized Marking algorithm can achieve a competitive ratio better than $(2 - \varepsilon)H_k$ for any $\varepsilon > 0$, meaning that MARK is essentially optimal.

The first strongly competitive paging algorithm, denoted PARTITION, was proposed in [48]. While it achieves the optimal competitive ratio of H_k, its time and space requirements are in the worst case proportional to the input size independently of the cache size, which makes them hopelessly high. More recent research has focused on improving these bounds, especially the space requirements. In the literature, a *bookmark* refers to a page outside the cache that the algorithm keeps track of; in particular, an algorithm is denoted *trackless* if it stores no

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

bookmarks at all. In [1], an H_k-competitive algorithm, denoted EQUITABLE, was proposed, using only $O(k^2 \log k)$ bookmarks. Using a better version of EQUITABLE, denoted EQUITABLE2, this bound was further improved in [11] to $2k$ bookmarks. This solved the open question in [14] that there exist H_k-competitive paging algorithms using $O(k)$ space. In [21] we proposed an algorithm, denoted ONLINEMIN, which further improved EQUITABLE2 by reducing its runtime for processing a page from $O(k^2)$ to $O(\log k / \log\log k)$ while maintaining its space requirements.

A distinct line of research for randomized paging algorithms consists of considering fixed small cache sizes ($k = 2$ and $k = 3$ to our best knowledge) to obtain tighter bounds than for general k. In [9], for $k = 2$, a $\frac{3}{2}$-competitive algorithm using only one bookmark was proposed. Also in [9], for trackless randomized algorithms a lower bound on the competitive ratio of $\frac{37}{24} \approx 1.5416$ was given. Still for $k = 2$, a trackless algorithm having an upper bound of ≈ 1.6514 was introduced in [23]. Finally, in [11], strongly competitive randomized paging algorithms were proposed for $k = 2$ and $k = 3$, using 1 and 2 bookmarks respectively.

Our Contributions. This work focuses on the number of bookmarks needed by randomized algorithms to achieve the optimal competitive ratio of H_k. The best previously known result is $2k$ [11]. In [11] it was conjectured that there exist algorithms that use $o(k)$ bookmarks and are H_k-competitive.

We first give a tighter analysis for EQUITABLE2 improving the amount of bookmarks from $2k$ to $\approx 0.62k$, which is the first solution using less than k bookmarks. We give a negative result showing that EQUITABLE2 cannot achieve a competitive ratio of H_k using $o(k)$ bookmarks. Nonetheless, we show that it can trade competitiveness for space: if it is allowed to be $(H_k + t)$-competitive, it requires $k/(1 + t)$ bookmarks.

We propose an algorithm PARTITION2 which is a modification of the PARTITION algorithm. PARTITION2 improves the bookmarks requirements from proportional to input size to $\Theta(k/\log k)$ and thus proves the $o(k)$ conjecture. For our analysis we provide a constructive equivalent between the two representations of the *offset functions* in [48] and [43]. Since offset functions are the key ingredient in the design and analysis of optimal competitive algorithms for paging, this may be of independent interest. Finally, we show that k/H_k is a lower bound on the number of bookmarks for any strongly competitive algorithm which uses a deterministic approximation of the offset function. This makes PARTITION2 asymptotically optimal within this class.

5.2 Preliminaries

In this section we give a brief introduction concerning offset functions for paging, the EQUITABLE algorithms, and forgiveness as a space bounding technique.

Offset Functions. In competitive analysis the cost approximation of the optimal offline algorithm plays an important role. For the paging problem it is possible to track online the exact minimal cost using *offset functions*. For a fixed input sequence σ and an arbitrary cache configuration C (i.e., a set of k pages), the *offset function* ω assigns to C the difference between the minimal cost of processing σ ending in configuration C and the minimal cost of processing σ. A configuration is called *valid* iff $\omega(C) = 0$. In [43] it was shown that the class of valid configurations $\mathcal{V}$ determines the value of ω on any configuration C by $\omega(C) = \min_{X \in \mathcal{V}}\{|C \setminus X|\}$. We can assume that OPT is always in a valid configuration. More precisely, if p is requested and there exists a valid configuration containing p, then the cost of OPT is 0; otherwise OPT pays 1 to process p.

Layer Representation. In [43] it was shown for the paging problem that the actual offset function can be represented as a partitioning of the pageset in $k+1$ disjoint sets $L = (L_0|L_1|\ldots|L_k)$, denoted layers. An update rule for the layers when processing a page was also provided. Initially, the first k pairwise distinct requested pages are stored in layers $L_1, \ldots, L_k$, one page per layer, and L_0 contains the remaining pages. Upon processing page p, let $L^p = (L_0^p|L_1^p\ldots|L_k^p)$ be the partitioning after processing p; L^p is obtained from L as follows[2]:

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

This layer representation can be used to keep track of all valid configurations. More specifically, a set C of k pages is valid iff $|C \cap L_i| \leq i$ holds for all $0 \leq i \leq k$ [43].

For a given L, denote by *support* $S(L) = L_1 \cup \cdots \cup L_k$. Also, we call a layer containing a single page a *singleton*. Let r be the smallest index such that $L_r, \ldots, L_k$ are singletons. The pages in $L_r, \ldots, L_k$ are denoted *revealed*, the pages in support which are not revealed are *unrevealed*, and the pages in L_0 are denoted *Opt-miss*. OPT faults on a request to p iff $p \in L_0$ and all revealed pages are (independent of the current request) in OPTs cache. If L consists only of revealed pages it is denoted a *cone* and we know the content of OPT's cache. Although the layer representation is not unique it has an unique *signature*. The signature $\chi(L)$ is defined as a k-dimensional vector $\chi = (x_1, \ldots, x_k)$, with $x_i = |L_i| - 1$ for each $i = 1, \ldots, k$.

Selection Process. In [21] we defined a priority based selection process on L which is guaranteed to construct a valid configuration. Assume that pages in the support have pairwise distinct priorities. Our selection process builds a hierarchy of sets $C_0, \ldots, C_k$ as follows:

[2]We use the layer representation introduced in [21], which is equivalent to the ones in [1,43].

- $C_0 = \emptyset$
- C_i consists of the i pages in $C_{i-1} \cup L_i$ having the highest priorities, for all $i > 0$.

Note that, by definition, when cunstructing C_i there are $i + x_i$ candidates and i slots. Also, if L_i is singleton we have $x_i = 0$ and $C_i = C_{i-1} \cup L_i$; for singleton layers and only for singleton layers, all elements in both C_{i-1} and L_i make it to C_i and we say that no competition occurs. The outcome C_k contains k pages and is always a valid configuration. In particular, if the priorities are the negated timestamps of the next requests (in the future) for the support pages, then C_k is identical to OPT's cache.

Equitable and OnlineMin. The cache content of the EQUITABLE algorithms [1, 11] is defined by a probability distribution over the set of all valid configurations. The cache configuration depends solely on the current offset function. This distribution is achieved by the ONLINEMIN algorithm using the previously introduced priority-based selection process, when priorities are assigned to support pages such that each permutation of the ranks of these pages is equally likely. The cache content of ONLINEMIN is at all times the outcome C_k of the selection process. Since we use in the remainder of the paper only this selection process, we do not describe the selection process for EQUITABLE. Nonetheless, the resulting probability distribution on cache configurations is the same as for EQUITABLE, and in the rest of the paper we refer to this distribution and the associated algorithm as EQUITABLE.

Forgiveness. Note that the support size increases only when pages in L_0 are requested, and may decrease only when pages in L_1 are requested. As the amount of Opt-miss requests may be very large, the support size and together with it the space usage of algorithms, such as EQUITABLE, using it to decide their cache content may also be arbitrarily large. To circumvent this problem, the *forgiveness* mechanism is used. Intuitively, if the support size exceeds a given threshold, then the adversary did not play optimally and we can afford to use an approximation of the offset function which is bounded in size.

5.3 Better Bounds for Equitable2

There are two EQUITABLE algorithms, EQUITABLE [1] and EQUITABLE2 [11][3]. For a fixed offset function (both use an approximation of the actual offset function), they have the same distribution as previously introduced. The difference between them is given by the forgiveness mechanism used. In this section we

[3]In [11] EQUITABLE2 is denoted K_EQUITABLE. In this paper we use its original name.

focus on the EQUITABLE2 algorithm using the forgiveness mechanism described in [11] which works as follows. Whenever the support size reaches a threshold value and an Opt-miss page is requested, the requested page is artificially inserted in L_1 and processed as a L_1 page. This way, all pages in L_1 move to L_0 and the support size never exceeds the designated threshold. The threshold in [11] is set to $3k$, i.e. the algorithm uses $2k$ bookmarks. We give a tighter analysis and show that using the same forgiveness the algorithm uses less than k bookmarks. We also give lower bounds showing that it can not achieve $o(k)$ bookmarks while preserving its H_k competitive ratio. Finally, we show that it can trade competitiveness for space. More specifically, if the algorithm is allowed to be $(H_k + t)$-competitive, it can be implemented using $k/(1 + t)$ bookmarks, where t is an arbitrary non-negative value, e.g a function in k.

To accommodate the selection process for ONLINEMIN previously introduced, all pages in support have pairwise distinct priorities, such that each priority ordering of the support pages is equally likely. We say that some page p has *rank* i in a set if its priority is the i'th largest among the elements in the given set.

Potential. In [1] an elegant potential function, based only on the current offset function, was introduced. Given the layer representation L, the potential $\Phi(L)$ is defined to be the cost of a so-called *lazy attack sequence*, that is, a sequence of consecutive requests to unrevealed pages until reaching a cone. The potential Φ is well defined because in the case of the EQUITABLE distribution, all lazy attack sequences have the same overall cost for a given offset function [1].

Initially, we are in a cone and thus $\Phi = 0$. Upon a request to a page p in support, having cache miss probability $pb(p)$, by definition we have that $\Delta\Phi = -pb(p)$. On lazy requests OPT does not fault and thus $\Delta cost + \Delta\Phi = \Delta cost_{OPT} = 0$. Upon a request from L_0 both EQUITABLE and OPT have cost 1 and it was shown that $\Delta\Phi \leq H_k - 1$ [1, 11]. Since upon revealed requests both algorithms never fault and the offset function does not change we have:

$$\Delta cost + \Delta\Phi \leq H_k \cdot \Delta cost_{OPT}.$$

If L is a cone, it is easy to verify that a request in L_0 leads to $\Delta\Phi = H_k - 1$. If the support size is strictly larger than k the difference in potential is smaller, i.e. $\Delta\Phi < H_k - 1$. This means that the algorithm pays less than its allowed cost and thus it can make *savings*. These savings can be tracked by a second potential function and are used to pay for the forgiveness step when the support size becomes large enough. While Φ is very comfortable to use for requests in support, for arbitrary offset functions there is no known closed form for its exact actual value or for its exact change upon a request in L_0.

5.3.1 Approximation of Φ

The key ingredient to our analysis is to get a bound as tight as possible for $\Delta\Phi$ on requests in L_0. That is because a tighter bound for this value implies larger savings, which in turn means that these savings can pay earlier (i.e. for a smaller support size) for a forgiveness step, which in the end means fewer bookmarks. We therefore analyze $\Delta\Phi$ for requests to pages in L_0 when no forgiveness step is applied. Note that Φ depends only on the signature $\chi = (x_1, \ldots, x_k)$ of the layer representation. We use $\chi = 0$ for the cone signature $(0|0|\ldots|0)$ and $\chi = e_i$ for the i-th unit vector $(0|\ldots|x_i = 1|\ldots|0)$. If $\chi = 0$ we have $\Phi = 0$. Otherwise, let i be the largest index such that $x_i > 0$. Since all lazy attack sequences have the same cost, we get that Φ is the cost of i consecutive requests, each of them to a page in the (current) first layer. For the layer representation L of the current offset function, we let $cost_1(L)$ denote the probability of cache miss for a page p in L_1, i.e. $pb(p \notin C_k)$ in the selection process.

We start with a simple case, where all layers are singletons except some layer L_i. The potential Φ for this particular case is easy to calculate and is given in Lemma 5.1.

Lemma 5.1 *Let $\chi = n \cdot e_i$ be the signature of L, where $n > 0$ and $0 < i < k$. We have $\Phi(\chi) = n \cdot (H_{i+n} - H_n)$.*

Proof. Let p be a page in L_1. Since there is no competition in the selection process for C_j, where $j \neq i$, we have that $p \in C_{i-1}$ independent of its priority and $p \in C_k$ iff $p \in C_i$. For the selection in C_i we have i slots and $i+n$ candidates. All these candidates have the same probability to be selected in C_i, since all layers $L_1, \ldots, L_i$ are singleton and thus no competition steps happened; note that this argument holds only if $x_1 = \cdots = x_{i-1} = 0$. This means that the probability of a cache miss is $\frac{n}{i+n}$. Updating the layers leads to $\chi = n \cdot e_{i-1}$. Repeating the argument we obtain:

$$\Phi = \frac{n}{i+n} + \frac{n}{i-1+n} + \cdots + \frac{n}{1+n} = n(H_{i+n} - H_n) \ ,$$

and the claim holds. □

For some arbitrary values i, n, and κ, where $0 < i < \kappa \leq k$ consider the signatures $\chi = n \cdot e_i$ and $\chi' = n \cdot e_i + e_{\kappa-1}$; let L and L' be their corresponding layer representations. We define the difference in the cost for a request in L_1:

$$f(i, n, \kappa) = cost_1(\chi') - cost_1(\chi) \ .$$

In the special case $\kappa = k$ it represents $\Delta cost_1$ upon a request in L_0. The value for $f(i, n, \kappa)$ can be computed exactly and is given in Lemma 5.2.

Lemma 5.2 $f(i, n, \kappa) = \frac{1}{n+\kappa} \prod_{j=i}^{\kappa-1} \frac{j}{n+j}$.

Proof. If $i = \kappa-1$, we have $cost_1(\chi) = n/(\kappa-1+n)$ and $cost_1(\chi') = (n+1)/(\kappa+n)$, and the result immediately follows. For the remainder of the proof we assume $i < \kappa - 1$. Consider a priority assignment for χ' and a request to some page p in L'_1. By the selection process for OnlineMIN, the value of $f(i, n, \kappa)$ is given by the probability that $p \in C'_i$ and $p \notin C'_{\kappa-1}$, since if $p \in C'_{\kappa-1}$ then $p \in C'_k$ and the probability that $p \in C'_i$ is the probability of a cache hit in χ, i.e. if $p \in C_i$ then $p \in C_k$. The scenario $p \in C'_i$ and $p \notin C'_{k-1}$ happens when p has rank i (i.e. has the i'th highest priority) among the $n + i$ pages in $L'_1 \cup \cdots \cup L'_i$ and all pages in $L'_{i+1}, \ldots, L'_{\kappa-1}$ have greater priorities than p. There are $(n + i - 1)!$ possibilities that p has rank i among the $n + i$ pages in $L'_1 \cup \cdots \cup L'_i$. For each of these, there are $i \cdot (i + 1) \cdot \ldots \cdot (\kappa - 1)$ possibilities that all the $\kappa - i$ pages in $L'_{i+1}, \ldots, L'_{\kappa-1}$ have priorities higher than p. We get that:

$$f(i, n, \kappa) = \frac{(n + i - 1)! \prod_{j=i}^{\kappa-1} j}{(n + \kappa)!} = \frac{1}{n + \kappa} \prod_{j=i}^{\kappa-1} \frac{j}{n + j} ,$$

which concludes the proof. □

We are now ready to move to a more general case. In Lemma 5.3 we show that $f(i, n, \kappa)$ is an upper bound on $\Delta cost_1$ for a whole class of signatures.

Lemma 5.3 *Consider a signature $\chi = (x_1|\ldots|x_k)$, and let i be the minimal index with $x_j = 0$ for all $j > i$. Also, let $\chi' = \chi + e_{\kappa-1}$, $i < \kappa \leq k$. For $n = x_1 + \cdots + x_i$, we have*

$$cost_1(\chi') - cost_1(\chi) \leq f(i, n, \kappa).$$

Proof. Let $g(i, n, \kappa) = cost_1(\chi') - cost_1(\chi)$. Similar to the proof of Lemma 5.2, the value of $g(i, n, \kappa)$ is given by the probability that a request $p \in L'_1$ is in C'_i and not in $C'_{\kappa-1}$. Intuitively, the proof is based on the observation that the fact that p must have exactly rank i among the $n+i$ pages in $L'_1 \cup \cdots \cup L'_i$ is necessary but not sufficient, whereas in the proof of Lemma 5.2 this fact was necessary and sufficient.

Assume $i < \kappa - 1$. By the definition of the selection process, if $p \in C'_i$ then the priority of p is compared against the priorities of all pages in $L'_1 \cup \cdots \cup L'_i$, because $p \in L'_1$; note that this doesn't necessarily hold if $p \in L'_j$, with $j > 1$. This immediately means that p must necessarily have rank i in $L'_1 \cup \cdots \cup L'_i$. The number of permutations where p has rank i among the $n+i$ pages is $(n+i-1)!$. However, it may not hold that for all of them we have $p \in C'_i$. Let $j_1, \ldots, j_t$ be indices smaller than i such that $x'_{j_l} \neq 0$ for all j_l. To have $p \in C'_i$, the priority of p must also be among the largest j_1 in $L'_1 \cup \cdots \cup L'_{j_1}$, among the largest j_2 in $C'_{j_1} \cup L'_{j_1+1} \cdots \cup L'_{j_2}$ and so on; in short, p must overcome t selection processes, instead of one as in the proof of Lemma 5.2. The set P of permutations on the $(n + i)$ pages in the first i layers where p has rank i and $p \in C'_i$ has size at most

$(n+i-1)!$. Recall that all $\kappa - i$ elements in $L'_{i+1} \cup \cdots \cup L'_{\kappa-1}$ must have higher priorities than p. For each permutation in P there are $i \cdot (i+1) \cdot \ldots \cdot (\kappa - i)$ possibilities to do so. In total, we get that:

$$g(i,n,\kappa) = \frac{|P| \prod_{j=i}^{\kappa-1} j}{(n+\kappa)!} \leq \frac{(n+i-1)! \prod_{j=i}^{\kappa-1} j}{(n+\kappa)!} = f(i,n,\kappa) \; .$$

If $i = \kappa - 1$, we have that $g(i,n,\kappa)$ is the probability that $p \in C_i$ and $p \notin C'_i$. Let q be an arbitrary page in L'_i. Then $g(i,n,\kappa)$ is bounded by the probability that q has rank $(i+1)$ in $L'_1 \cup \cdots \cup L'_i$ and rank i in $L'_1 \cup \cdots \cup L'_i \setminus \{q\}$. Using a similar reasoning, there are $(n+i-1)! \cdot i$ possibilities for this scenario to occur, which concludes the proof. □

Lemma 5.4 provides a useful identity for approximating $\Delta\Phi$ for a request in L_0.

Lemma 5.4 *For any i and κ with $i < \kappa$, it holds that $\sum_{j=1}^{i} f(i-j+1, 1, \kappa - j + 1) = H_\kappa - H_{\kappa-i} - \frac{i}{\kappa+1}$.*

Proof. We first note that:

$$f(i,1,\kappa) = \frac{1}{\kappa+1} \cdot \frac{i}{i+1} \cdot \frac{i+1}{i+2} \cdots \frac{\kappa-1}{\kappa} = \frac{i}{\kappa(\kappa+1)} \; .$$

Denoting by $S(i,\kappa) = \sum_{j=1}^{i} f(i-j+1, 1, \kappa-j+1)$ and using that $i/(\kappa(\kappa+1)) = i/\kappa - i/(\kappa+1)$, we have:

$$\begin{aligned} S(i,k) &= f(i,1,\kappa) + \cdots + f(1,1,\kappa-(i-1)) \\ &= \frac{i}{\kappa} - \frac{i}{\kappa+1} + \cdots + \frac{1}{\kappa-(i-1)} - \frac{1}{\kappa-(i-2)} \\ &= \frac{1}{\kappa} + \cdots + \frac{1}{\kappa-(i-2)} + \frac{1}{\kappa-(i-1)} - \frac{i}{\kappa+1} \; , \end{aligned}$$

which concludes the proof. □

Theorem 5.1 *For a request to a page $p \in L_0$ where no forgiveness is applied, let i be the largest index with $x_i > 0$; $i = 0$ if we are in a cone. We have that:*

$$H_{k-i} - H_1 \leq \Delta\Phi \leq H_k - H_1 - i/(k+1).$$

Proof. For $i = 0$, in a cone we have $\Delta\Phi = H_k - 1$ by Lemma 5.1. If $i > 0$, let L and L', and χ and $\chi' = \chi + e_{k-1}$ denote the layers and their corresponding signatures before and after the request to p respectively. We consider the cost of a sequence of i consecutive requests $p_1, \ldots, p_i$, each of these to pages in the current L_1. For each $j = 1, \ldots, i$ let χ^j and χ'^j denote the signatures before

processing p_j. After the whole sequence is processed, we have $\chi = 0$ with $\Phi = 0$ and $\chi' = e_{k-i-1}$ with $\Phi' = H_{k-i} - H_1$ by Lemma 5.1. We get:

$$\Delta\Phi = H_{k-i} - H_1 + \sum_{j=1}^{i} \Big(cost_1(\chi'^j) - cost_1(\chi^j) \Big)$$

Since $cost_1(\chi'^j) - cost_1(\chi^j)$ is non-negative, the left inequation holds.

Now we bound $cost_1(\chi'^j) - cost_1(\chi^j)$ using Lemma 5.3. Before processing page p_j we have $x^j_{i-j+1} > 0$, $x^j_l = 0$ for all indices $l > i-j+1$ and $\chi'^j = \chi^j + e_{\kappa-1}$ with $\kappa = k - j + 1$. Denoting $n^j = x^j_1 + \cdots + x^j_{i-j+1}$, we get:

$$\begin{aligned} \Delta\Phi &\leq \sum_{j=1}^{i} f(i-j+1, n_j, k-j+1) + H_{k-i} - H_1 \\ &\leq \sum_{j=1}^{i} f(i-j+1, 1, k-j+1) + H_{k-i} - H_1 \\ &= H_k - H_{k-i} - \frac{i}{k+1} + H_{k-i} - H_1 \ . \end{aligned}$$

The inequations stem from the fact that f is decreasing in n and $n^j > 0$ for all $j \leq i$, and the equality is the result in Lemma 5.4. □

5.3.2 Competitiveness and Bookmarks

Having obtained a tighter bound on $\Delta\Phi$ for requests in L_0, we get improved savings using a second potential Ψ. To define $\Psi(L)$, we first introduce the concept of *chopped signature*. For some signature $\chi = (x_1|\ldots|x_k)$, let i be the largest index such that $x_i > 0$. The chopped signature corresponding to χ is $\overline{\chi} = (\overline{x_1}|\ldots|\overline{x_k})$, where $\overline{x_i} = x_i - 1$ and $\overline{x_j} = x_j$ for all $j \neq i$. If we are in a cone and $\chi = 0$ we define $\overline{\chi} = \chi$. Ψ is defined as:

$$\Psi(L) = \frac{1}{k+1} \sum_{i=1}^{k-1} i \cdot \overline{x_i} \ .$$

Note that $\Psi(L) = 0$ if $\chi = 0$ or $\chi = e_i$ and otherwise we have $\Psi(L) > 0$.

Fact 5.1 *For a request to page $p \in L_i$, $i > 0$, it holds:*

$$\Delta\Psi = -\frac{1}{k+1} \sum_{j=i}^{k-1} \overline{x_j}.$$

To prove that EQUITABLE2 is H_k-competitive, it suffices to show that for each request $cost + \Phi + \Psi \leq H_k \cdot cost_{OPT}$, as both Φ and Ψ are non-negative. We do so by proving for each step the inequation is preserved by considering the differences in costs and potentials.

Lemma 5.5 *If no forgiveness is applied it holds,*

$$\Delta cost + \Delta\Phi + \Delta\Psi \leq H_k \cdot \Delta cost_{OPT}.$$

Proof. We first analyze the case for a request $p \in L_i$, with $i > 0$. We have $\Delta cost + \Delta\Phi = 0$ by the definition of Φ and $\Delta cost_{OPT} = 0$. By Fact 5.1 $\Delta\Psi \leq 0$ and we are done.

For requests to pages in L_0, both the algorithm and OPT incur a cost of one, and thus $\Delta cost = 1$ and $\Delta cost_{OPT} = 1$. It remains to show that $\Delta\Psi + \Delta\Phi \leq H_k - 1$. We analyze separately the case when we are in a cone. In this case, by definition $\Delta\Psi = 0$, and by Lemma 5.1 we obtain $\Delta\Phi = H_k - 1$. In the following we assume we are not in a cone upon the L_0 request. Let i be the largest index with $x_i \neq 0$. By the update rule, we get that $x'_{k-1} = x_{k-1} + 1$ and $x'_j = x_j$ for all $j \neq k-1$. For the chopped signature $\overline{\chi'}$ this implies $\overline{x'_j} = \overline{x_j}$ for all $j \neq i$ and $\overline{x'_i} = \overline{x_i} + 1$, because $i \neq k$ as L_k is always singleton. It follows $\Delta\Psi = i/(k+1)$. On the other hand we have by Theorem 5.1 that $\Delta\Phi \leq H_k - H_1 - i/(k+1)$. □

Theorem 5.2 EQUITABLE2 *is H_k-competitive and requires $2 + \frac{\sqrt{5}-1}{2} \cdot k$ bookmarks.*

Proof. If the support size reaches the threshold $k+x$, i.e. x bookmarks, we apply upon a request from L_0 the forgiveness mechanism from [11]. Recall that we move the requested page artificially into L_1. This step does not increase OPT's overall cost. Then we process it as if it was requested from L_1. We have $\Delta cost = 1$ and $\Delta cost_{OPT} = 0$. Like in [11], we need to prove that $1 + \Delta\Phi + \Delta\Psi \leq 0$. Denote by χ the current signature, and let $x = \sum_{i=1}^{k} x_i$ be the number of bookmarks used by the algorithm. We have that $\Delta\Phi = -cost_1(\chi)$. We get that $1 + \Delta\Phi$ is the probability that a page in L_1 is in the algorithm's cache, which by the selection process of ONLINEMIN is at most $k/|S| = k/(x+k)$. Using the result in Fact 5.1 and the fact that $\sum_{j=1}^{k-1} \overline{x_j} = x - 1$, we need to ensure that:

$$\frac{k}{x+k} - \frac{x-1}{k+1} \leq 0\,.$$

Solving this inequation, we get $x \geq (1 - k + \sqrt{5k^2 + 6k + 1})/2$, which is at most $\frac{\sqrt{5}-1}{2}k + c$ for $c \geq 2$. Therefore, EQUITABLE2 needs only $\frac{\sqrt{5}-1}{2}k + c \approx 0.62k$ bookmarks. The cases where no forgiveness occurs are covered by Lemma 5.5. □

Lower Bound. We now show in Theorem 5.3 that EQUITABLE2 can not achieve $o(k)$ bookmarks and be H_k-competitive.

Theorem 5.3 *If* EQUITABLE2 *uses* $t \leq k/4$ *bookmarks, it is not* H_k*-competitive.*

Proof. For easiness of exposition we assume that k is divisible by 4. It suffices to build an input sequence which starts and ends in a cone where the cost of EQUITABLE2 using t bookmarks exceeds $H_k \cdot OPT$ for arbitrary large k. This sequence consists of three phases.

In the first phase we bring t additional pages into layer L_i (no forgiveness occurs), where the index $i > 0$ is determined later. To do so, we request a page in L_0 leading to $\chi = e_{k-1}$ followed by $k - i - 1$ requests from L_{i+1}. The resulting signature is $e_i = (0|\ldots|x_i = 1|\ldots|0)$. We repeat this step $t - 1$ more times and obtain the signature $\chi_i = t \cdot e_i$ which by Lemma 5.1 has the potential $\Phi_i = t(H_{t+i} - H_t)$. By Theorem 5.1, each request in L_0 increases Φ by at least $H_{k-i} - 1$, leading to a total amount of potential increases $\Phi_+ = t * H_{k-i} - t$. Since Φ decreases upon lazy requests the total cost of EQUITABLE during this phase is

$$t + \Phi_+ - \Phi_i = t \cdot (H_{k-i} - H_{t+i} + H_t).$$

The second phase starts with a request from L_0 which forces EQUITABLE2 to apply forgiveness. This leads to $\chi = t \cdot e_i + e_{k-1}$ whereas the signature used by EQUITABLE2 is $\chi_{Eq} = t \cdot e_{i-1}$. This means that page $q \in L_1$ in the (original) layer representation is for sure not in cache. We request q. We can repeat the last request type $i - 1$ additional times which leads to a total cost in the second phase of i whereas OPT pays 1. In the third phase we bring the (original) offset function to a cone, and repeat revealed requests (if needed) such that EQUITABLE also reaches a cone and we can repeat our attack. The third phase incurs no cost for OPT. Choosing $i = (k - t)/2$ we need to show:

$$\frac{tH_t + 0.5(k - t)}{t + 1} > H_k.$$

Setting $t = k/4$, we get:

$$1.5 + \cdot H_{k/4} - H_k - \frac{H_k}{k/4} > 0.$$

For the value $k = 200$ the left side is about 0.0036. The term $H_{k/4} - H_k$ is increasing in k. To see this let $k = k + 4$. We obtain a difference of $\frac{4}{k} - \frac{1}{k+1} - \frac{1}{k+2} - \frac{1}{k+3} - \frac{1}{k+4} > 0$. On the other hand $\frac{H_k}{k/4}$ is decreasing in k. We conclude that the inequation is true for $k \geq 200$. □

Trading Competitiveness for Space. We now show that EQUITABLE2 can achieve $o(k)$ bookmarks at the expense of competitiveness. This result is given in Theorem 5.4.

Theorem 5.4 *There exist implementations of* EQUITABLE2 *that are* $(H_k + c)$*-competitive and use* $k/(1 + c)$ *bookmarks, for* $k > 1$ *and* $c \geq 1$.

Proof. Again, we consider two functions Φ and Ψ, both initially set to zero, and for each request we prove that:

$$\Delta cost + \Delta\Phi + \Delta\Psi \leq (H_k + c)\Delta cost_{OPT} \ .$$

As before, Φ is the cost of a lazy sequence of requests in the support ending in a cone. However, Ψ is defined differently: $\Psi = \frac{c}{k+1} \sum_{j=1}^{k-1} j \cdot \overline{x_j}$.

For requests in L_0 when no forgiveness step is applied, we have $\Delta cost = 1$, $\Delta cost_{OPT} = 1$, and, by Theorem 5.1, we get $\Delta\Phi \leq H_k - H_1 - i/(k+1)$, where i is the largest index having $x_i > 0$. Also, similarly to Lemma 5.5, we get $\Delta\Psi \leq \frac{ci}{k+1}$, which, using $i < k$, leads to $1 + \Delta\Phi + \Delta\Psi \leq H_k + c$.

For pages in support, we analyze the request to a page $p \in L_i$. By definition of Φ, we have $\Delta cost + \Delta\Phi = 0$. The result in Fact 5.1 can be adapted straightforward to obtain $\Delta\Psi = -\frac{c}{k+1} \sum_{j=i}^{k-1} \overline{x_j}$. Altogether, we get $\Delta cost + \Delta\Phi + \Delta\Psi \leq 0$.

For requests in L_0 , when forgiveness must be applied, we use the same forgiveness mechanism from [11], where the requested page is artificially inserted in L_1 and processed as a page in L_1. Again, in this case, the algorithm is charged a cost of 1, and OPT is charged 0. We have that $1 + \Delta\Phi$ is the probability of a cache hit for a page in L_1, which is at most $\frac{k}{x+k}$, where $x = \sum_{j=1}^{k} x_j$ is the amount of bookmarks allowed. Using $\Delta\Psi = -\frac{c}{k+1}(x - 1)$, we need to ensure that $\frac{k}{x+k} \leq \frac{cx}{k+1}$. Solving the inequation, we get that it holds for $x \geq -\frac{k}{2} + \frac{\sqrt{c^2k^2+4kc}}{2c}$. Enforcing $x = k/(1 + c)$, the result follows. □

We note that the result in Theorem 5.4 gives a range of algorithms whose performance is between the classic EQUITABLE and MARKING algorithms, with respect to competitiveness and space usage; in particular, the interesting values for c are such that $c = \omega(1)$ and $c < H_k - 1$. That is because, classic EQUITABLE is H_k-competitive but uses $\Theta(k)$ bookmarks, while MARKING uses no bookmarks, but is $2H_k - 1$ competitive.

5.4 Partition

In this section we prove in the affirmative the conjecture in [11] that there exists a strongly competitive paging algorithm using $o(k)$ bookmarks. We propose a variation of the PARTITION algorithm [48], that we call PARTITION2, which uses $O(k/\log k)$ bookmarks. We furthermore give a simple lower bound showing that

for any H_k-competitive randomized paging algorithm, the number of pages having non-zero probability of being in cache must be at least $k + k/H_k$. This leads to a lower bound of k/H_k bookmarks for all algorithms which store all non-zero probability pages, i.e. representation of the approximated offset function, and have a deterministic forgiveness step. Note that this bound holds for all known H_k-competitive algorithms with bounded space usage, i.e. depending only on k.

5.4.1 Algorithm

In this section we give a brief description of the PARTITION algorithm in [48]. A crucial difference between PARTITION and EQUITABLE is that while the distribution of the cache configurations depends only on the current offset function for EQUITABLE, PARTITION is defined on a special, more detailed, representation of the offset function, which we denote in the following *set-partition*. We show in Observation 5.1 that the offset function alone does not suffice to determine the probability distribution for the cache of PARTITION[4]. It partitions the whole pageset into a sequence of disjoint sets $S_\alpha, S_{\alpha+1}, \ldots, S_{\beta-1}, S_\beta$ and each set S_i with $i < \beta$ has a *label* k_i. Initially $\beta = \alpha + 1$, S_β contains the first k pairwise distinct pages, the remaining pages are in S_α, and $k_\alpha = 0$. Throughout the computation S_β contains all revealed pages (pages which are in OPT's cache independent of the future requests) and S_α all the pages which are not in OPT's cache. Upon a request to page p the set-partition is updated as follows. If $p \in S_\beta$ nothing changes. If $p \in S_\alpha$ the following assignments are done:

$$S_\alpha = S_\alpha \setminus \{p\},\ S_{\beta+1} = \{p\},\ k_\beta = k - 1,\ \beta = \beta + 1.$$

The last case covers $p \in S_i$, where $\alpha < i < \beta$:

$$S_i = S_i \setminus \{p\},\ S_\beta = S_\beta \cup \{p\},\ k_j = k_j - 1\ (i \leq j < \beta).$$

Additionally, if there are labels which become zero, let j be the largest index such that $k_j = 0$; the following assignments are performed:

$$S_j = S_\alpha \cup \cdots \cup S_j,\ \alpha = j.$$

In [48] it was shown that the following invariants on the labels hold: $k_\alpha = 0$ and $k_i > 0$ for all $i > 0$; $k_\beta = k - |S_{\beta-1}|$. Furthermore, it holds at all times that:

$$k_i = (k_{i-1} + |S_i|) - 1.$$

[4]Previous work [1] gave a simplified and intuitive description of PARTITION, but which is not fully accurate.

Probability Distribution of Cache Configurations. The probability distribution of the cache content can be described as the outcome of the following selection process on the set-partition:

- $\mathcal{C}_\alpha = \emptyset$
- For $\alpha < i < \beta$ choose p uniformly at random from $\mathcal{C}_{i-1} \cup S_i$ and set $\mathcal{C}_i = (\mathcal{C}_{i-1} \cup S_i) \setminus \{p\}$
- $\mathcal{C}_\beta = \mathcal{C}_{\beta-1} \cup S_\beta$.

Note that, whereas for the selection process of OnlineMIN the size of C_i is given by i, for Partition we have that $|\mathcal{C}_i| = k_i$. The following result was given in [48, Lemma 3].

Lemma 5.6 *If p is requested from S_i, where $\alpha < i < \beta$, the probability that p is not in the cache of* Partition *is at most*

$$\sum_{i \le j < \beta} \frac{1}{k_j + 1}.$$

Cache Replacement. Apart from obeying the cache distribution previously introduced, Partition must satisfy two constraints, namely it must not evict pages upon a cache hit and it must not evict more than one page upon a cache miss. For any set $\mathcal{C}_i$, the membership of a page to $\mathcal{C}_i$ is encoded with a marking system on pages as follows. If a page is in set S_i, where $\alpha < i < \beta$, it has either no mark or a series of marks $i, i+1, \ldots, j-1, j$. If p has no mark then $p \notin \mathcal{C}_i$ and otherwise it is in the selection sets $\mathcal{C}_i, \mathcal{C}_{i+1}, \ldots, \mathcal{C}_{j-1}, \mathcal{C}_j$. The cache of Partition is at all times $\mathcal{C}_\beta$, with $|\mathcal{C}_\beta| = k$. For a page $p \in S_i$ it suffices to store the value m_p of the highest mark or $i-1$ if p has no mark.

Initially there are only the two sets S_α and S_β and thus no marks. If the requested page $p \in S_\beta$ nothing changes. If $p \in S_\alpha$ first the set-partition is updated, where β is increased by 1 and we have to determine $\mathcal{C}_{\beta-1}$. A page q is chosen uniformly at random from the k elements $\mathcal{C}_{\beta-2} \cup S_{\beta-1}$ (the cache content before the request), and this element is the only one not receiving a $\beta - 1$ mark. The page q is replaced in the cache by the requested page p. We now turn to the case $p \in S_i$, where $\alpha < i < \beta$. If p is in cache then $m_p = \beta - 1$ and we do nothing. Otherwise let $j \le \beta - 1$ be the lowest index such that $p \notin \mathcal{C}_j$. We choose uniformly at random a page $q \in \mathcal{C}_j$ and set $m_p = m_q$ and $m_q = j - 1$, i.e. p steals the marks of q. We repeat this until $m_p = \beta - 1$. The page which loses its $\beta - 1$ mark is replaced in cache by p. Afterwards the set-partition is updated.

Observation 5.1 *The probability distribution of* Partition *does not depend on the offset function alone.*

Proof. To illustrate the claim, we give two scenarios leading to the same offset function where there exist a page having different probabilities of being in cache. In the first scenario, we start with the cone $L^1 = (p_1|\dots|p_{k-1}|q_1)$ and request two pages from L_0, namely q_2 and q_3. Since upon a request in L_0 PARTITION evicts a page uniformly at random from cache, the probability that q_1 is in cache after processing q_3 is $(k-1)^2/k^2$. In the second scenario, we start with offset function $L^2 = (p_1|\dots|p_{k-1}|q_2)$ and we request q_1 and q_3, both of which are in L_0. This leads to the same layer representation of the offset function as in the first scenario, but the probability that q_1 is in cache is now only $(k-1)/k$, which concludes the proof. □

5.4.2 Partition2

In this section we describe the PARTITION2 algorithm. As implied by its name, it is a variant of PARTITION which uses (deterministic) forgiveness to reduce the space usage from arbitrarily high bookmarks to $O(k/\log k)$ bookmarks. A lower bound is provided which shows that this bound is asymptotically optimal for algorithms using deterministic forgiveness. Unlike previous works, when a forgiveness step must be applied, we distinguish between two cases and apply two distinct forgiveness rules accordingly. The first of them is the same one used by EQUITABLE2 and covers only a single request, and the second one is a *forgiveness phase* which spans consecutive requests. To apply the forgiveness step of EQUITABLE2, we first provide an embedding of the set-partition into the layer representation of the offset function. Based on this embedding, we give a simple potential function which depends only on the signature of the offset function.

Layer Embedding. In the following we provide an embedding of the set-partition into the layer representation of the offset functions, as used by EQUITABLE. The layers become ordered sets and contain pages and set identifiers, the latter of which we visualize by $\star$. The initialization does not change and no set identifiers are present. The update rule changes mainly for the case $p \in L_0$:

$$L_{k-1} = (L_{k-1}, L_k, \star),\ L_k = \{p\}.$$

Upon the merge operation $L_{i-1} \cup L_i \setminus \{p\}$ in the case $p \in L_i$ we remove p from L_i and concatenate L_{i-1} with L_i without removing any set identifier. Upon merging L_1 into L_0 we delete all set identifiers from the resulting layer L_0. An example is given in Figure 5.1. The following fact follows inductively.

Fact 5.2 *For L_i, with $i > 0$ and $|L_i| = 1 + x_i$, it holds*

- *L_i contains exactly x_i set identifiers,*
- *if $x_i > 0$ then the last element in L_i is a set identifier.*

We describe how to obtain the sets of the set-representation. Let j be maximal such that $x_j > 1$. We have $S_\beta = L_{j+1} \cup \cdots \cup L_k$ and $S_\alpha = L_0$. A set $S_{\alpha+j}$, where $1 < j < \beta - \alpha$ consists of all pages between the $(j-1)$-th and the j-th set identifier; for $j = 1$, $S_{\alpha+1}$ consists of all support pages until the first set identifier. We say that each set $S_{\alpha+j}$, $0 < j < \beta - \alpha$, is *represented* by the j'th set identifier. As long as no pages are moved into S_α, the correspondence between the layer representation and the set-partition follows immediately from the update rules. Otherwise, by Lemma 5.7 and noticing that each L_i with $x_i > 0$ ends in a set delimiter, we obtain that p is in L_1 and moreover the pages moved to S_α correspond to $L_1 \setminus \{p\}$.

Lemma 5.7 *Let $S_a, S_{a+1}, \ldots, S_b$ be the sets whose identifiers are in layer L_i, $i \geq 0$. We have:*

$$k_b = i,\ k_{a+j} \geq i \text{ for } 0 \leq j < b - a\ .$$

Proof. We show that the invariant remains true after each update of the set-partition. Let p be the currently requested page; also let L and L' be the layer representation and S and S' the corresponding set-partition before and after processing p respectively.

If page $p \in S_\beta$ nothing (except a shift of the revealed layers in L) changes. If $p \in S_\alpha$ we also have $p \in L_0$. Page $q \in L_k$ followed by a new set identifier (representing the set $S_{\beta'-1}$) is appended to L_{k-1} and $L'_k = \{p\}$. All sets except for $S_{\beta'-1}$ are not affected. The set-partition update rule assigns $k_{\beta'-1} = k - 1$. Since the identifier of $S_{\beta'-1}$ is the rightmost element in L'_{k-1}, the result holds.

Now we turn to the case $p \in S_{i^*}$, where $\alpha < i^* < \beta$. Let L_i be the layer containing p. if L_i is singleton, then for all sets S_{j^*}, $j^* \geq i^*$ we have that both k_{j^*} and its corresponding layer index decrease by 1. Since the relevant parameters for the remaining sets don't change, the result holds. If L_i is not singleton, by construction L_i ends in a set identifier; this set identifier represents a set S_{j^*}, $j^* \geq i^*$. By inductive hypothesis, we get $k_{j^*} = i$. By the update rules, $k'_{j^*} = i - 1$ and it is the last set identifier in L'_{i-1}. All other set identifiers in L_i represent sets having labels at least i, which might decrease by at most 1. All these identifiers are moved to L'_{i-1} and the result follows. □

Lemma 5.8 *If p is requested from L_i, where $i > 0$, the probability that p is not in the cache of* Partition *is at most*

$$\sum_{j \geq i} \frac{x_j}{j+1}$$

Proof. If $p \in S_\beta$, then it is in a revealed layer L_i and thus $x_j = 0$ for all $j \geq i$ and the result holds. Let S_{i^*} be the set with $p \in S_{i^*}$, $\alpha < i^* < \beta$. Then by Lemma 5.6 we have the probability bounded by $\sum_{i^* \leq j^* < \beta} \frac{1}{k_{j^*}+1}$. All sets S^*_j,

Req	Offset function	
-	$L = (7, 8, 9 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6)$	
	$S = \{7, 8, 9\}_0\{1, 2, 3, 4, 5, 6\}$	$(\alpha = 1, \beta = 2)$
9	$L = (7, 8 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5, 6, \star \mid 9)$	
	$S = \{7, 8\}_0\ \{1, 2, 3, 4, 5, 6\}_5\ \{9\}$	$(\alpha = 1, \beta = 3)$
6	$L = (7, 8 \mid 1 \mid 2 \mid 3 \mid 4, 5, \star \mid 9 \mid 6)$	
	$S = \{7, 8\}_0\ \{1, 2, 3, 4, 5\}_4\ \ \{9,6\}$	$(\alpha = 1, \beta = 3)$
8	$L = (7 \mid 1 \mid 2 \mid 3 \mid 4, 5, \star \mid 9, 6, \star \mid 8)$	
	$S = \{7\}_0\ \{1, 2, 3, 4, 5\}_4\ \{9, 6\}_5\ \{8\}$	$(\alpha = 1, \beta = 4)$
1	$L = (7 \mid 2 \mid 3 \mid 4, 5, \star \mid 9, 6, \star \mid 8 \mid 1)$	
	S=$\{7\}_0\ \{2, 3, 4, 5\}_3\ \{9, 6\}_4\ \{8, 1\}$	$(\alpha = 1, \beta = 4)$
9	$L = (7 \mid 2 \mid 3 \mid 4, 5, \star, 6, \star \mid 8 \mid 1 \mid 9)$	
	$S = \{7\}_0\ \{2, 3, 4, 5\}_3\ \{6\}_3\ \{8, 1, 9\}$	$(\alpha = 1, \beta = 4)$
6	$L = (7 \mid 2 \mid 3, 4, 5, \star, \star \mid 8 \mid 1 \mid 9 \mid 6)$	
	$\{7\}_0\ \{2, 3, 4, 5\}_3\ \{\}_2\ \{8, 1, 9, 6\}$	$(\alpha = 1, \beta = 4)$
3	$L = (7 \mid 2, 4, 5, \star, \star \mid 8 \mid 1 \mid 9 \mid 6 \mid 3)$	
	$S = \{7\}_0\ \{2, 4, 5\}_2\ \{\}_1\ \{8, 1, 9, 6, 3\}$	$(\alpha = 1, \beta = 4)$
5	$L = (7, 2, 4 \mid 8 \mid 1 \mid 9 \mid 6 \mid 3 \mid 5)$	
	$\{7, 2, 4\}_0\ \{8, 1, 9, 6, 3, 5\}$	$(\alpha = 3, \beta = 4)$

Figure 5.1: Example for the layer embedding of the set-representation.

where $i^* \leq j^* < \beta$ have their identifier in some layer L_j with $j \geq i$ and using Lemma 5.7 we obtain $\frac{1}{k_{j^*}+1} \leq \frac{1}{j+1}$. Since each layer L_j contains exactly x_j identifiers the statement follows. □

Forgiveness. Forgiveness is applied when the support size reaches a threshold of $k + 3t$ (we define t later) and a page in L_0 is requested. Depending on the support we have two kinds of forgiveness: *regular forgiveness* and an *extreme forgiveness mode*. The regular forgiveness is applied if $|L_1| + \cdots + |L_t| > 2t$ and is an adaptation of the forgiveness step of EQUITABLE2. If a page p is requested from L_0 (equivalent to S_α), we first identify a page q satisfying that $q \in S_{\alpha+1} \cap L_1$. Note that there always exists such a page, since $k_{\alpha+1} \geq 1$ and $|S_1| = k_1 + 1$ and at least one of them is in L_1. We move q to L_0 and replace it, together with its marks, by p. Then we perform the set-partition and mark update where p is requested from $S_{\alpha+1}$. We stress that in terms of the layer representation of the offset function (used by e.g. EQUITABLE), we replace the requested page with an existing page in L_1, and replacing $q \in L_1$ by p and requesting p leads to the same offset function when the forgiveness step in [11] is applied. This has a cost of 1 for PARTITION and a cost of 0 for OPT. The size of the support decreases by $|L_1| - 1 \geq 0$.

The extreme forgiveness mode is applied if $|L_1| + \cdots + |L_t| \leq 2t$. We simply apply regular forgiveness for any page request in L_0 starting with the current one. This extreme forgiveness mode ends when reaching a cone.

Competitive Ratio and Bookmarks We use PARTITION with the forgiveness rule for $t = \lceil \frac{k}{\ln k} \rceil$ from the previous paragraph if $k > 10$ and denote the resulting algorithm PARTITION2. For $k \leq 10$ we apply the the regular forgiveness if the support size reaches $2k$.

Theorem 5.5 PARTITION2 *uses* $\Theta(\frac{k}{\log_k})$ *bookmarks and is* H_k*-competitive.*

Proof. The space bound follows from the fact that the support size never exceeds $k + 3t$ for $k > 10$, where $t = \lceil \frac{k}{\ln k} \rceil$. It remains to show that PARTITION2 is still H_k-competitive. We use the following potential on the layer representation of the offset function:

$$\Phi = \sum_{j=1}^{k-1} x_j \cdot (H_{j+1} - 1)$$

We denote by *cost* the cost of PARTITION2 and by OPT the cost of the optimal offline algorithm. We have to show that $cost \leq H_k \cdot OPT$ holds after each request. In all cases except the extreme forgiveness we show that the following holds before and after each request

$$\Phi + cost \leq H_k \cdot OPT.$$

This leads to $cost \leq H_k \cdot OPT$ since $\Phi \geq 0$. When applying the extreme forgiveness we assume that the potential inequation holds before the phase and show that it holds at the end of the phase, but not necessary during the phase. For requests during the phase we argue directly that it always holds $cost \leq H_k \cdot OPT$.

Let p be the requested page. If $p \in L_0$ without forgiveness, $\Delta OPT = 1$ and x_{k-1} increases by 1, which implies that $\Delta\Phi + \Delta cost = H_k - 1 + 1 = 1 \cdot H_k$.

If p is from some layer L_i, where $0 < i \leq k$, we use the bound on the cache miss probability from Lemma 5.8

$$\Delta\Phi + \Delta cost \leq -\sum_{j \geq i} \frac{x_j}{j+1} + \sum_{j \geq i} \frac{x_j}{j+1} \leq 0 \leq H_k \cdot \Delta OPT.$$

Now we analyze the cases where forgiveness occurs for $k > 10$. Assume that $|L_1| + \cdots + |L_t| \geq 2t + 1$ which implies that $x_1 + \cdots + x_t \geq t + 1$. We perform just one forgiveness step, yielding $\Delta cost = 1$ and $\Delta OPT = 0$. We have to show that $\Delta\Phi \leq -1$.

$$\Delta\Phi = -\sum_{j=1}^{k-1} \frac{x_j}{j+1} \leq -\sum_{j=1}^{t} \frac{x_j}{t+1} = -\frac{t+1}{t+1} = -1.$$

Now assume that $x_{t+1} + \cdots + x_{k-1} \geq 2t$. Before we start the extreme forgiveness mode, we have that

$$\Phi \geq \sum_{j=t+1}^{k-1} x_j(H_{j+1} - 1) \geq 2t(H_{t+2} - 1)$$

By the choice of $t = \lceil \frac{k}{\ln k} \rceil$ and the approximation $H_x \geq \ln x$ we obtain

$$\Phi \geq \frac{2k}{\ln k}(\ln k - \ln\ln k - 1) \geq k, \text{ if } k > 10.$$

Right before the phase starts we have $cost + \Phi \leq H_k \cdot OPT$, where $\Phi \geq k$ which is equivalent to $cost \leq H_k \cdot OPT - k$. Reaching the next cone implies at most $k-1$ unrevealed requests and thus the cost during this phase is bounded by $k-1$. This implies that $cost \leq H_k \cdot OPT$ holds. Since in a cone $\Phi = 0$ we also have at the end of the phase the invariant $cost + \Phi \leq H_k \cdot OPT$.

For the case $k \leq 10$ the analysis of the extreme forgiveness does not hold. In this case we use only the regular forgiveness step if we have k bookmarks. Using $x_1 + \cdots + x_{k-1} = k$ the same argument as before leads to $\Delta\Phi \leq -1$. □

Lemma 5.9 *For any H_k-competitive algorithm A there exists an input such that the maximal number of pages with non-zero probability of being in A's cache is at least $k + k/H_k$.*

Proof. We assume that A is H_k-competitive and the number of pages with non-zero probability is always less than $k+k/H_k$. We start in a cone $(p_1|p_2|\ldots|p_k)$ and request $q_1, q_2, \ldots, q_\alpha$, where $\alpha = k/H_k$ and all q_i have never been requested before. Thus OPT and A perform each α page faults. The resulting work function has the signature $(0|\ldots|0|\alpha|0)$ and the support has size $k + \alpha$. By our assumption there exists at least one page from the support on which A faults with probability 1. Since for the next $k-1$ requests the support does not change we can force $k-1$ page faults on A each with cost 0 for OPT. Afterwards we continue the request sequence to reach a cone and repeat our attack. We conclude that A is not H_k competitive

$$\frac{\text{cost}(A)}{\text{cost}(OPT)} = \frac{k-1+\alpha}{\alpha} = 1 + \frac{k-1}{k/H_k} > H_k \ ,$$

and the proof follows. □

5.5 Conclusions

We have shown that Partition2 improves the bookmark complexity from $O(k)$ to $O(k/\log k)$ and thus proved the conjecture that there exist H_k-competitive

randomized paging algorithms using $o(k)$ bookmarks. This is the best possible for algorithms using deterministic forgiveness techniques and store the whole representation of the (approximated) offset function. One possible direction to improve this bound is to use randomization at the forgiveness step. The more LRU-like distribution of PARTITION and its simple potential in the layer embedding seems to be the more promising candidate.

We stress that the forgiveness used for PARTITION2 does not lead to $o(k)$ bookmarks for the distribution of EQUITABLE. Nonetheless, EQUITABLE is interesting due to its $O(\log k)$ runtime and the elegant potential definition. Moreover, the priority-based selection process in [21] gives an alternate approach to analyzing the EQUITABLE distribution by employing elementary combinatorics.

Chapter 6

Outperforming LRU via Competitive Analysis

The work **Outperforming LRU via Competitive Analysis on Parametrized Inputs for Paging** was published as a conference paper [51].

[51] G. Moruz and A. Negoescu. Outperforming LRU via competitive analysis on paramterized inputs for paging. In *Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1669–1680, 2012

The contents of this chapter correspond to the published conference version [51] except for minor layout changes.

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 [44] 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. [3] and Dorrigiv et al. [30]. 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.

6.1 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

*Partially supported by the DFG grants ME 3250/1-3 and MO 2057/1-1, and by MADALGO.

†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.

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 of the algorithm 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* [41, 60], states that some online algorithm A is c-competitive if for any input sequence it holds that $cost(A) \leq c \cdot cost(OPT) + b$, where $cost(A)$ and $cost(OPT)$ denote the cost of A and the optimal cost respectively, and b is a constant. For deterministic paging algorithms, a lower bound of k on the competitive ratio was shown in [60]. 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 [14]. For randomized algorithms, Fiat et al. [32] 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 algorithms achieving the optimal bound of H_k on the competitive ratio were proposed in [1, 11, 20, 48], each of them improving over its predecessors with respect to space complexity and running time for processing a page, up to $O(k)$ space and $O(\log k)$ time [20].

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 [68], which makes it (together with its variants) very popular in practice [63]. 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 characterization to obtain algorithms performing few cache misses.

Related Work. 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* [44] or *loose compet-*

itiveness [68]. Other approaches consider comparing algorithms directly, without relating them to an optimal offline algorithm. Relevant examples include the *Max/Max ratio* [13], the *random order ratio* [42], the *relative worst order ratio* [17], and *bijective analysis* and *average analysis* [6].

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 [26, 27] the paging strategy takes into account the most recently used pages, denoted *working set*. The *access graph model* [15, 31, 38] restricts input sequences by confining the next request to a restricted set of pages depending on the current request. In [3, 30] 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 [7]) 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) [17] and FARL (Farthest-To-Last-Request) [14, 33] 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) [61]. It simulates many algorithms and chooses the most promising one when deciding which page to evict. This allows it to outperform 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 [44] 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, 11], this shows that insights from classical competitive analysis can help to design algorithms with low fault rate on real world inputs.

6.2 Input Parametrization

A classical optimal offline algorithm, denoted LFD (Longest Forward Distance), has been proposed in [12], 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 [44]. Based on the request sequence seen thus far, the page currently requested can fall in one of the three categories below.

1. Requests to *revealed pages*, this are requests to pages we know for sure that they are in LFDs cache.
2. Requests to *unrevealed pages*, which might be in the cache of LFD, depending on the future requests.
3. 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.

6.2.1 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 $\omega(C) = 0$. In [44] 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 ω^p the partition after processing p, we have the following:[1]

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

The *support* of the offset function ω is defined as $S(\omega) = \cup_{i=1}^{k} L_i$. 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 $R(\omega) = \cup_{i=r}^{k} L_i$. Intuitively, the layer representation keeps track of the possible cache configurations of the optimal offline algorithm. To this end, in [44] it has been shown that a cache configuration is valid iff it holds that for each $i \in \{1, \ldots, k\}$ we have $|C \cap (\cup_{j=1}^{i} L_j)| \leq i$. 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 6.1 *Between two requests in L_0, at most $k-1$ pairwise distinct pages are requested. Moreover, LFD faults on some page p iff $p \in L_0$.*

6.2.2 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

[1] We use a different, yet equivalent notation to the one in [44].

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 σ let $\lambda_r(\sigma)$, $\lambda_u(\sigma)$, and $\lambda_0(\sigma)$ denote the number of requests in σ to revealed pages, unrevealed pages in the support, and pages that are not in support respectively.

Definition 6.1 *For some input σ, the attack rate $r(\sigma)$ is defined as $r(\sigma) = \frac{\lambda_0(\sigma)+\lambda_u(\sigma)}{\lambda_0(\sigma)}$. Also, we denote by $\mathcal{I}(r)$ the set of inputs having an attack rate at most r, i.e. $\mathcal{I}(r) = \{\sigma | r(\sigma) \leq r\}$.*

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 $\mathcal{I}(k)$ 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.

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. 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 6.1 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 $\frac{\lambda_u+\lambda_0}{\lambda_u+\lambda_0+\lambda_r}$; 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 λ_r

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

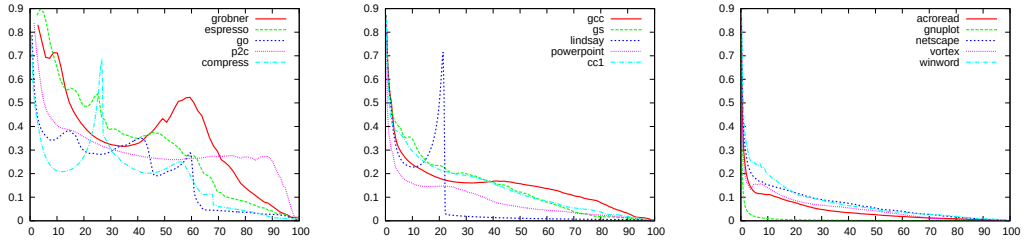


Figure 6.1: The attack rate r in real-world inputs. The x-axis shows the percentage of pages that fit in cache, i.e. k, and the y-axis shows the ratio between r and the cache size k. The constant line $f(x) = 1$ corresponds to $r = k$ and is the upper bound on the competitive ratio by standard competitive analysis.

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 [26]. To quantify the locality of reference in the input, Albers et al. [3] 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. [30] 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.

We conduct experiments which show the fault rate as predicted by the four approaches, together with the actual fault rate of LRU. In Figure 6.2 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[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.

6.3 Input-Parametrized Competitive Ratio

6.3.1 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

[3]For all datasets we considered the full input to compute the parameters for locality of reference in the Max model, as opposed to [3] 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 [3] for the same application.

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 [44] 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 [20]. 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 [20] 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 [11]. 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.

6.3.2 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 6.1.

Class	Competitive ratio
CacheMin	∞
Marking	$[2r-1, 2r]$
OnOPT	r

Algorithm	Competitive ratio
LFU, MRU, LIFO	∞
FWF	$[2r-1, 2r]$
LRU, FIFO	r

Table 6.1: The guaranteed competitive ratio for the generic classes (left) and for classic algorithms (right).

Lemma 6.1 *The competitive ratio for any deterministic paging algorithm on an input in $\mathcal{I}(r)$, for any arbitrary rational $r \in [1 \dots k]$, 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 $r = \frac{kn_1+n_2}{n_1+n_2}$. 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 6.2 *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 $\sigma = p_1, \dots, p_k, (xy)^*$. 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 6.2 *The competitive ratio for any marking algorithm on an input in $\mathcal{I}(r)$ 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 6.1 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 bound. For each of them we count the number of cache misses done by Mark (MK) and OPT (λ_0), and the number of requests λ_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 ω. Initially, we have $\omega = (p_1|\dots|p_k)$ and after the first request to p_{k+2} we have $\omega = (p_1|\dots|p_{k_2}|p_{k-1}, p_k, p_{k+1}|p_{k+2})$. After the request to p_2 we have $\omega = (p_{k+2}|p_k|\dots|p_3|p_2)$, 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. □

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 6.3 *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 6.1 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 λ_0 done by OPT concludes the proof.

□

6.4 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.

6.4.1 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 OPTs 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.

6.4.2 RDM on Real-World Traces

We conduct experiments 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 [17] and EELRU [61].

RLRU (Retrospective LRU) was proposed in [17], 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 [61] 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 6.3 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.

6.5 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.

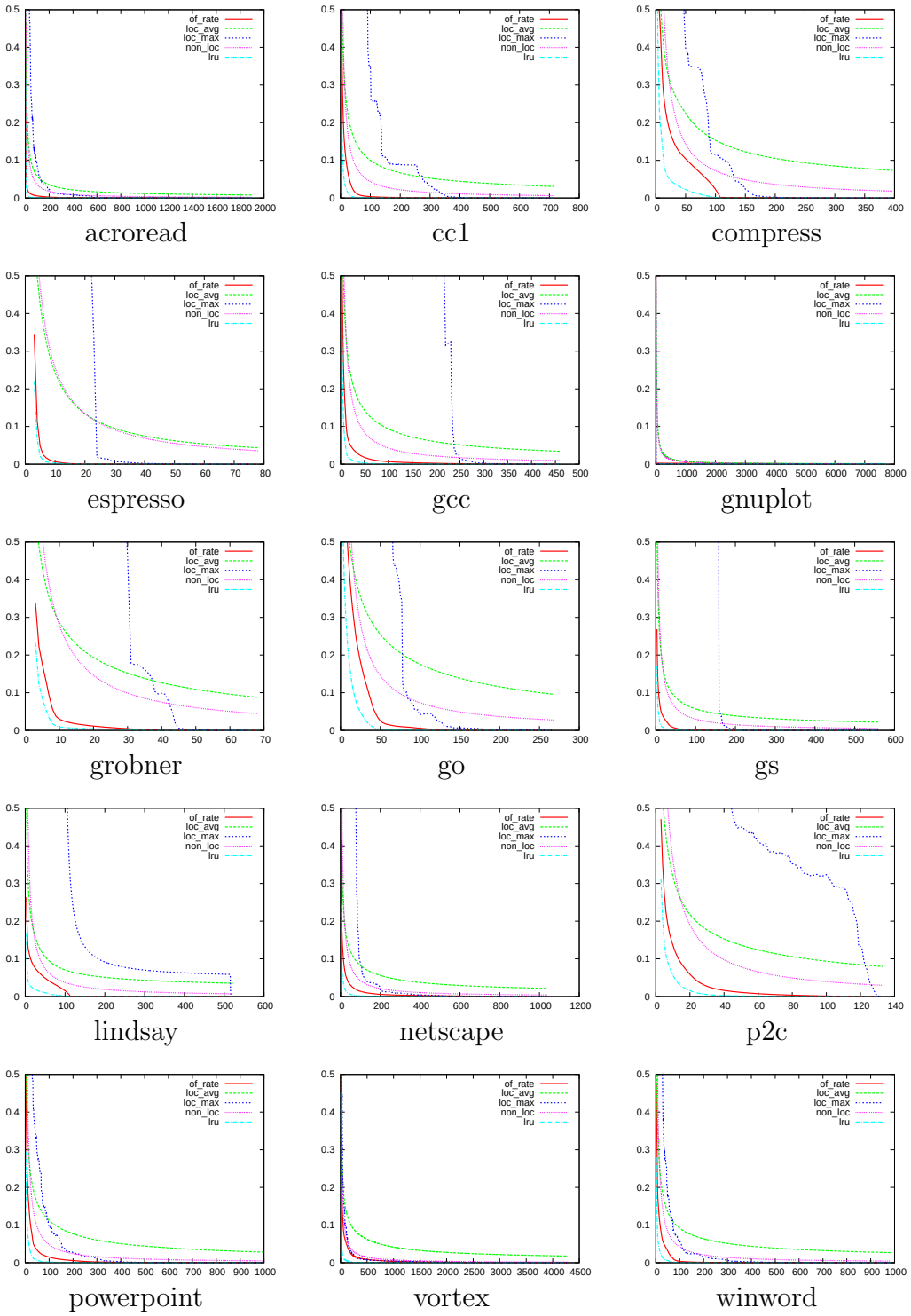


Figure 6.2: The predicted fault rate by offset function $of = \frac{\lambda_0+\lambda_u}{\lambda_0+\lambda_u+\lambda_r}$, the locality of reference in the Max- and Average-model, and the non-locality of reference, together with the actual performance of LRU for the first twelve datasets. The x-axis shows the cache size and the y-axis shows the fault rate.

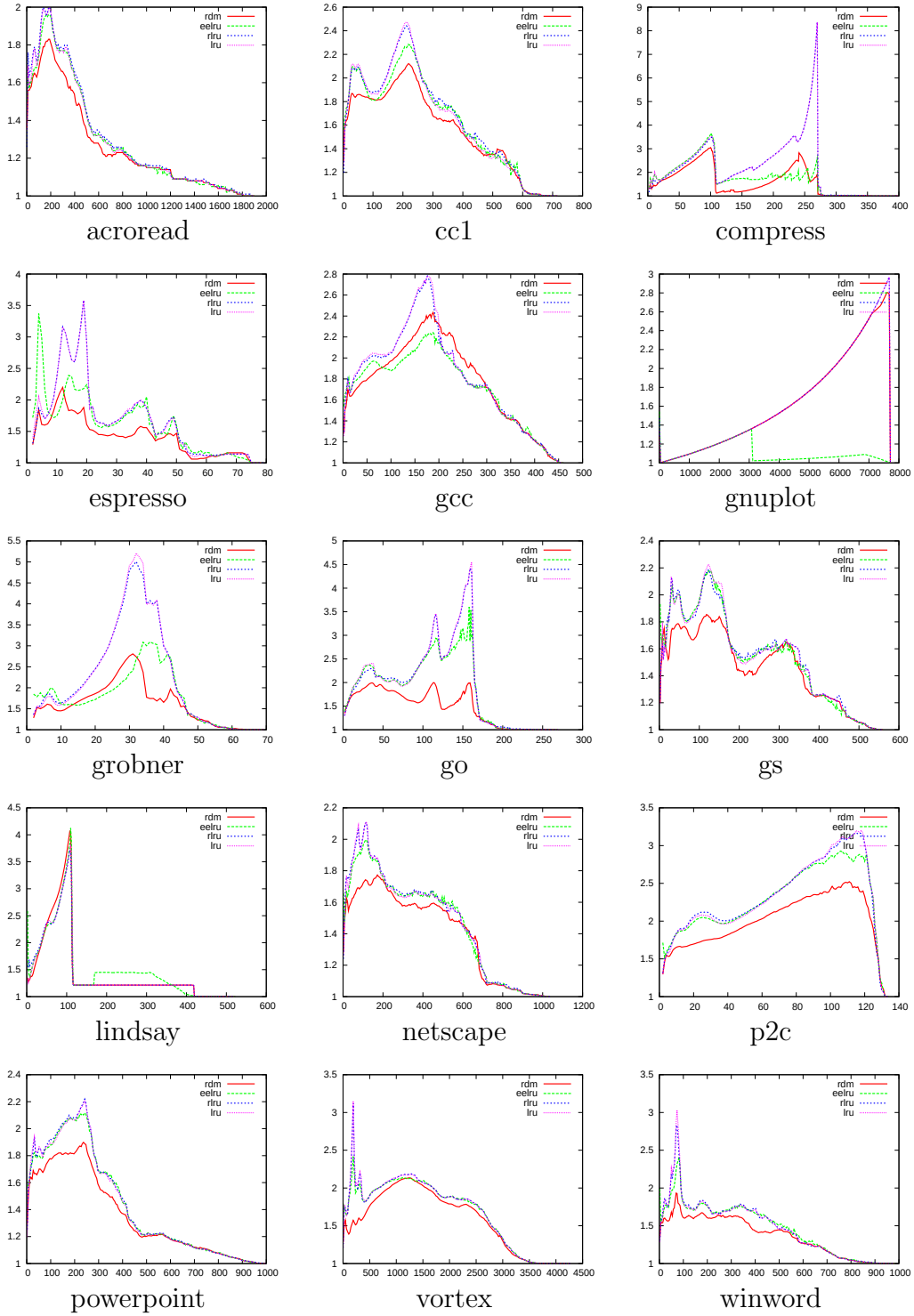


Figure 6.3: The empirical competitive ratio on various inputs for RDM, LRU, RLRU, and EELRU. The x-axis shows the cache size and the y-axis shows the competitive ratio.

Chapter 7

Engineering Efficient Paging Algorithms

The work **Engineering Efficient Paging Algorithms** was published as a conference paper [54] and as a journal paper [53] (invited for the special issue of the *Journal of Experimental Algorithmics* dedicated to SEA 2012).

[54] G. Moruz, A. Negoescu, C. Neumann, and V. Weichert. Engineering efficient paging algorithms. In *Proc. 11th International Symposium on Experimental Algorithms*, pages 320–331, 2012

[53] G. Moruz, A. Negoescu, C. Neumann, and V. Weichert. Engineering efficient paging algorithms. *Journal of Experimental Algorithmics, Special issue of SEA 2012 (to appear)*

The contents of this chapter correspond to the journal version [53] which includes all results of the conference paper [20]. It extends the conference paper by more detailed experimental results and proofs omitted due to lack of space.

Engineering Efficient Paging Algorithms*

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

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 [51], 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.

7.1 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 to be processed is in cache, the algorithm simply proceeds to the next page. However, 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,

*This work is 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).

†Department of Computer Science, Goethe University Frankfurt am Main, Robert-Mayer-Str. 11-15, 60325 Frankfurt am Main, Germany. Email: {gabi,negoescu,neumann,weichert}@cs.uni-frankfurt.de.

apart from cache misses, factors such as runtime and space usage have a major impact in deciding which algorithms to use [63, 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 [68]), but also because they have efficient implementations with low overhead in terms of space and runtime.

Typically, online algorithms in general and paging algorithms in particular are analyzed using *competitive analysis* [41, 60], 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 [60], 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 [32] 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 [48], Equitable and Equitable2 [1, 11], and OnMIN [20].

Based on the layer partition in [44], we proposed in [51] 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 (LRU is also in OnOPT). 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 partition in [20, 44]. 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 a fast implementation of the OnOPT class. If the fraction of revealed requests is $1 - O(1/k)$ our implementation yields an amortized runtime of $O(1)$ per request with very small constant factors. We show on real-world input traces[2] that, for the particular case of RDM, the new implementation outperforms the tree based approach in [20]. Moreover, we compare the runtime of RDM with that of LRU and FIFO and show that the runtimes of RDM and LRU are comparable, albeit

[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`.

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 [33], heuristics motivated by the access graph model from [15] 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 [17], 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) [39] 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.

7.1.1 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 [12]. 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 LFD's cache content equals C. A precise mathematical characterization of all possible valid configurations was given by Koutsoupias and Papadimitriou [44] and an equivalent variant of this characterization is used by the OnOPT algorithm class [51]. 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

σ	L_0	L_1	L_2	L_3
7,1,3	2,4,9	7	1	3
7,1,3,7	2,4,9	1	3	7
7,1,3,7,3	2,4,9	1	7	3
7,1,3,7,3,9	2,4	1	3,7	9
7,1,3,7,3,9,4	2	1	3,7,9	4
7,1,3,7,3,9,4,3	2	1,7,9	4	3
7,1,3,7,3,9,4,3,1	2,7,9	4	3	1

Figure 7.1: Application of layer update rules for $k = 3$ and request sequence σ.

resulting by the request of page p. The layers are updated as follows (an example is given in Figure 7.1):

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

In [44] it has been shown that a cache configuration C is valid iff it holds that for each $i \in \{1, \ldots, k\}$ we have $|C \cap (\cup_{j=1}^{i} L_j)| \leq i$.

The *support* of L is defined as $L_1 \cup \cdots \cup L_k$. Denoting *singleton* a layer with one element, let r be the smallest index such that $L_r, \ldots, L_k$ are singletons; the pages in $L_r \cup \cdots \cup L_k$ 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 reflects the rank of its next request among the other pages. For a priority based future prediction the cache update rule ensures that their cache content is always identical to LFD's, given that the prediction is correct. Algorithms in this class differ only in the priority strategy. The pseudo-code is given in Algorithm 3. We note that LRU is also in OnOPT, and is obtained by setting the current timestamp as priority for the currently requested page. The fact that no cache misses are performed on revealed requests guarantees a reasonable performance for all OnOPT algorithms, 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 [51].

Algorithm 3 OnOPT algorithms

procedure OnOPT(Page p, Cache M) ▷ Processes page p
  Assign p its priority
  if $p \notin M$ and $p \in L_0$ **then** ▷ Update cache
    Evict page in M with smallest priority
  else if $p \notin M$ and $p \in L_i$, $i > 0$ **then**
    Identify j such that $j \geq i$ and $|(L_1 \cup \cdots \cup L_j) \cap M| = j$
    Evict page in $L_1 \cup \cdots \cup L_j$ having smallest priority
  end if
  Update the layers ▷ Layers update
end procedure

7.1.2 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. Detailed information about these traces is given in Table 7.2.

Application	#pages	#requests	OS / Collected by	Description
grobner	68	7787835	Linux / VMTrace	Grobner basis functions
espresso	78	326938361	Linux / VMTrace	circuit simulator
p2c	133	30722431	Linux / VMTrace	Pascal to C transformer
go	268	106790719	Windows NT / Etch	AI program playing "Go"
compress	397	129116176	Windows NT / Etch	Compression utility
gcc	459	37524334	Linux / VMTrace	GNU C/C++ compiler
lindsay	522	123690749	Linux / VMTrace	hypercube simulator
gs	559	134371942	Linux / VMTrace	Postscript interpreter
cc1	717	263765501	Windows NT / Etch	Compiler core for gcc
winword	984	114359299	Windows NT / Etch	MS Word
powerpoint	1000	37384786	Windows NT / Etch	MS Powerpoint
netscape	1038	22077106	Windows NT / Etch	Netscape web browser
acroread	1904	94794501	Windows NT / Etch	Acrobat Reader
vortex	4276	543247591	Windows NT / Etch	Database program
gnuplot	7719	68458509	Linux / VMTrace	Plotting utility

Figure 7.2: Details on the cache traces we ran experiments on – the number of pairwise distinct pages requested, the total number of requests, the operating system under which the application was run, the tool used to collected the trace, and a description of the given application. The page size was 4KB [39].

The charts in Figure 7.3 show that if enough pages fit in memory (usually about 10%), almost all 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.

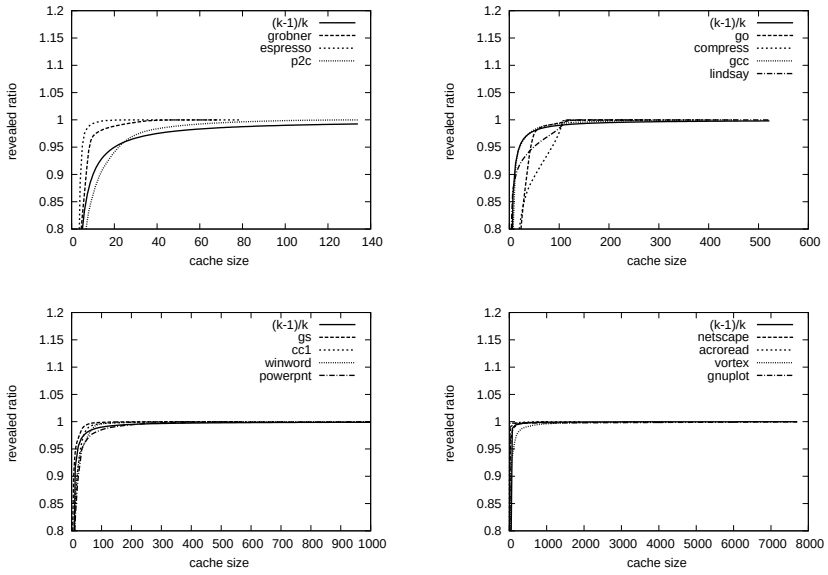


Figure 7.3: The ratio of revealed requests for all cache traces. For decently large cache sizes the ratio of revealed requests is about $(k-1)/k$ for all traces.

7.2 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 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 4, an example for $k = 7$ is provided in Figure 7.4.

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

Algorithm 4 Layer compression

```
procedure LAYER COMPRESSION(Partition L = (L_0, ..., L_i))  ▷ Compress L
    T = ∅;
    for i = 1 to k − 1 do
        if |L_i| = 1 then                                   ▷ L_i is singleton
            L_i = ∅; T = T ∪ L_i;
        else                                                ▷ L_i is not singleton
            L_i = L_i ∪ T; T = ∅;
        end if
    end for
    L_k = L_k ∪ T;
end procedure
```

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 7.1 *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 7.1.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8
Uncompressed	10,3	2	7,5	4	1,11	8	9	6
Compressed	10,3		2,7,5		4,1,11			8,9,6

Figure 7.4: Example of a layer representation and its compressed version.

Theorem 7.1 *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. Initially the first requested k pairwise distinct pages are distributed as singletons in $L_1, \ldots, L_k$ and thus $\mathcal{L}_k$ contains all these k pages and $\mathcal{L}_1, \ldots, \mathcal{L}_{k-1}$ are empty sets by construction. For all three cases we first identify the layer index in L which contains the requested page p, apply the update rule in L to obtain L^p and determine the changes in $\mathcal{L}$ such that $\mathcal{L}^p$ is the compressed partition of L^p.

If $p \in \mathcal{L}_0$ then p is in L_0. We distinguish two cases: $|\mathcal{L}_k| = 1$ and $|\mathcal{L}_k| > 1$. If $|\mathcal{L}_k| = 1$ then $|L_{k-1}| > 1$ and $L_k = \mathcal{L}_k$. The update of L modifies only two layers: $L^p_{k-1} = L_{k-1} \cup L_k$ and $L^p_k = \{p\}$. This affects $\mathcal{L}^p$ in the following way: $\mathcal{L}^p_{k-1} = \mathcal{L}_{k-1} \cup L_k = \mathcal{L}_{k-1} \cup \mathcal{L}_k$ and $\mathcal{L}^p_k = L^p_k = \{p\}$. Now assume $|\mathcal{L}_k| > 1$ which implies that L_{k-1} and L_k are singletons and $\mathcal{L}_{k-1} = \emptyset$. After the update of L we have $L^p_{k-1} = L_{k-1} \cup L_k$, $L^p_k = \{p\}$ and $|L^p_{k-1}| = 2$ which leads to $\mathcal{L}^p_{k-1} = \mathcal{L}_k$, which is equivalent to $\mathcal{L}^p_{k-1} = \mathcal{L}_{k-1} \cup \mathcal{L}_k$ since $\mathcal{L}_{k-1} = \emptyset$. Finally, $\mathcal{L}^p_k = \{p\}$.

The next case is $p \in \mathcal{L}_i$, $0 < i < k$. Since $\mathcal{L}_i$ is non-empty it implies that $|L_i| > 1$. First we assume $|L_{i-1}| > 1$, which implies $\mathcal{L}_i = L_i$ and $p \in L_i$. The update in L results in $L^p_{i-1} = L_{i-1} \cup L_i \setminus \{p\}$, $L^p_k = \{p\}$ and the index of the layers $L_{i+1}, \ldots, L_k$ is decreased by 1. Thus it still holds $|L^p_{i-1}| > 1$ leading to $\mathcal{L}^p_{i-1} = \mathcal{L}_{i-1} \cup L_i \setminus \{p\} = \mathcal{L}_{i-1} \cup \mathcal{L}_i \setminus \{p\}$. The decrease of the layer index is applied straightforward in $\mathcal{L}$. Since the layer $L^p_k = \{p\}$ extends the rightmost sequence of singletons in L we obtain $\mathcal{L}^p_k = \mathcal{L}_k \cup \{p\}$. The case $|L_{i-1}| = 1$ differs by the fact that p is not necessarily from L_i. In this case there exists an index $x < i$ where $|L_x| > 1$ and $L_{x+1} = \cdots = L_{i-1}$ are singletons. Page p is in one of the layers $L_{x+1}, \ldots, L_i$. Independent of the the exact layer index in L, after the update rule we have that $L^p_{x+1}, \ldots, L^p_{i-2}$ are singletons and $|L^p_{i-1}| > 2$ leading to $\mathcal{L}^p_{i-1} = \mathcal{L}_i \setminus \{p\} = \mathcal{L}_{i-1} \cup \mathcal{L}_i \setminus \{p\}$, since $\mathcal{L}_{i-1} = \emptyset$.

In the last case $p \in \mathcal{L}_k$, which implies that p is requested from one layer which is part of the rightmost consecutive singleton sequence in L. The update rule for L just permutes this singleton sequence, which does not affect the compressed partition $\mathcal{L}$. □

7.3 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 provide experimental results which support that, for the particular case of RDM, our improved implementation not only significantly outperforms the original approaches from [21], but is also competitive with LRU and FIFO in terms of runtime.

7.3.1 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 nl 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 a of page-set size where for a page p the associated information is accessed by a look-up at the corresponding element $a[p]$. 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 $\mathcal{L}_0$. We store in a *layer structure* information only about the non-empty layers in the support. We do so using an array of *layer identifiers* $(l_1, \ldots, l_{nl})$ where l_i corresponds to the ith non-empty layer and nl is the number of non-empty layers. Note that 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 layer identifier l_i, corresponding to layer $\mathcal{L}_j$, we keep the following: the timestamp $l_i.t$, a value $l_i.v$ which is at all times equal to $1 + e_i$ where e_i is the number of consecutive empty layers preceding $\mathcal{L}_j$, and $l_i.mem$ which stores the number of pages in $\mathcal{L}_j$ that are in cache, see e.g. Figure 7.5.

Lemma 7.2 *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 time for any page in $\mathcal{L}_i$ is smaller than the last request time of any page in $\mathcal{L}_{i+1}$. □

By Lemma 7.2, we have that $l_1.t < l_2.t < \cdots < l_{nl}.t$. Therefore, identifying the layer that a certain page belongs to can be done using binary search with its last request time as key. Also, layers can be inserted and deleted in $O(nl)$ time. Finally, it supports a *find-layer-j* operation, which, given a layer index i, returns the leftmost layer identifier l_j, with $j \geq i$ such that $|M \cap (\mathcal{L}_1 \cup \cdots \cup \mathcal{L}_j)| = j$. 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, there are typically 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.

t	7	14	21
v	3	2	3
mem	2	3	3
	$\mathcal{L}_3$	$\mathcal{L}_5$	$\mathcal{L}_8$

Figure 7.5: Example for $\mathcal{L} = (\emptyset|\emptyset|2,3,5,10|\emptyset|1,4,7,11|\emptyset|\emptyset|6,8,9)$ for the cache $M = (2,10,1,4,11,6,8,9)$. Pages are not stored in the layer structure.

Implementing OnOPT. We implement OnOPT algorithms using the structures described above. The pseudo-code is given in Algorithm 5.

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 time is greater than or equal to $l_{nl}.t$. Therefore, processing a revealed page takes $O(1)$ time.

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 $l_{nl}.v > 1$ then set $l_{nl}.v = l_{nl}.v - 1$ as $\mathcal{L}_{k-1}$ was empty; otherwise, i.e. $l_{nl}.v = 1$, delete this layer. Afterwards, we simply append a new layer l_{nl} with $l_{nl}.t$ set to the current timestamp, $l_{nl}.v = 1$, and $l_{nl}.mem = 1$: 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 up the page's layer l_i in the layer

Algorithm 5 The pseudo-code for OnOPT algorithms on compressed layers.

```
procedure OnOPT(Page p, Cache M)                          ▷ Processes page p
    Assign p its priority and update last request time
    if p is revealed then return                          ▷ Nothing to be done
    end if
    if p ∈ L_0 then                                       ▷ p is Opt-miss page
        Evict page in M with smallest priority
        Update layers
    else                                                  ▷ p is unrevealed
        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 in (L_1 ∪ ··· ∪ L_j) ∩ M having smallest priority
        end if
        Update layers
    end if
end procedure
```

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 time 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 $l_{nl}.v = l_{nl}.v + 1$ to reflect a new empty layer before $\mathcal{L}_k$. After updating the last request time for the requested page, it becomes revealed since this value is greater than $l_{nl}.t$. Thus, processing an unrevealed page takes $O(nl)$ time for a cache hit and $O(k)$ time for a cache miss.

Theorem 7.2 *Assuming m pairwise distinct pages are requested, a cache of size k, and nl 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 $O(nl)$ time for cache hits and $O(k)$ time for cache misses.*

Corollary 7.1 *Assuming that a ratio of $1-O(1/k)$ requests are to revealed pages, our implementation processes a request in $O(1)$ amortized time.*

Note that to decide the page to be evicted upon a cache miss we need a fully dynamic (i.e., supporting inserts and deletions) data structure which, given a timestamp t, finds the page having the smallest priority among the pages whose last request time is at most t. This can be achieved in $O(\log k)$ worst case time using a priority search tree, as was used in [20]. Unfortunately, this requires that, for each request the priority of the requested page to be updated, which takes $\Theta(\log k)$ worst case time. Thus, this would undermine the very idea of processing the overwhelming amount of revealed requests as fast as possible.

7.3.2 Experimental results

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

Experimental Setup. For OnOPT algorithms, apart from the engineered version previously introduced we implemented the two versions described in [20]. 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 for recency information and takes $O(1)$ time per request. The second LRU 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. The pseudo-code for the two LRU implementations is given in Figure 7.6. For FIFO, a circular array stores the FIFO queue. We note that we first measure the runtime of the paging algorithms alone, i.e. ignoring the time for a page replacement, and then we will address a scenario where each cache miss incurs a time penalty.

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. The source code, the charts, and the input traces are available online at `www.ae.cs.uni-frankfurt.de/sea12`.

We note that RDM is not the only algorithm outperforming LRU with respect to the number of cache misses incurred. Examples in this direction include RLRU [17] and EELRU [39]. RLRU (Retrospective LRU) was shown to outperform LRU on traces extracted from database applications. It is a mark-like algorithm which tries to keep in its cache the pages that OPT would have. EELRU (Early Eviction LRU) takes a different approach in that it simulates 256 paging algorithms which evict pages based on their recency (i.e. amount of pairwise-distinct pages requested since their last request), and selects the algorithm in this collection which works best. Unfortunately, we were unable to find efficient implementations for these algorithms which would render them competitive with the engineered version of RDM. Nonetheless, experimental data for the number

of cache misses incurred by both algorithms on the traces considered is given in [51]. Our implementations for these algorithms were significantly slower than RDM, i.e. by factors of at least 50. For RLRU, the number of misses incurred is virtually the same as LRU, however an implementation competitive with LRU seems highly unlikely. On the other hand, on most traces EELRU consistently outperforms LRU, however RDM typically incurs fewer misses. Even though there could be room for improvement for EELRU, by simulating a large collection of algorithms it must spend time to update the state of all of them upon each page request. Since this requires extensive time, we are skeptic that it could be implemented faster than RDM. Since we are unable to conclusively implement these two algorithms as best as possible, we refrain from giving experimental data for them; nonetheless, given their complexity and the number of misses incurred for the traces considered, we think it is unlikely they would outperform RDM.

```
LRUList(Page p, List M)
if p ∈ M then
    Move p to head of M
else
    Delete tail of M
    Add p as head of M
end if
```

```
LRULinear(Page p, Array M)
if p ∈ M then
    return
else
    Scan M to identify LRU page q
    Replace q by p
end if
```

Figure 7.6: Pseudo-code for LRUList (left) and LRULinear (right).

Non-empty Layers. We first compare the number of non-empty layers that we use in our implementation against the k layers used in the non-compressed one. In Figure 7.7 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 runtime for the five OnOPT variants, namely the one that we engineered as previously described together with the two implementations in [20], each of them using the compressed and uncompressed layer partition; as priority assignment, we used RDM. 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. To improve the binary search tree version which uses the compressed layer partitioning, we use an approximation of RDM where for revealed requests priorities do not change and this update becomes unnecessary and is not performed; the results shown are for this approximation.

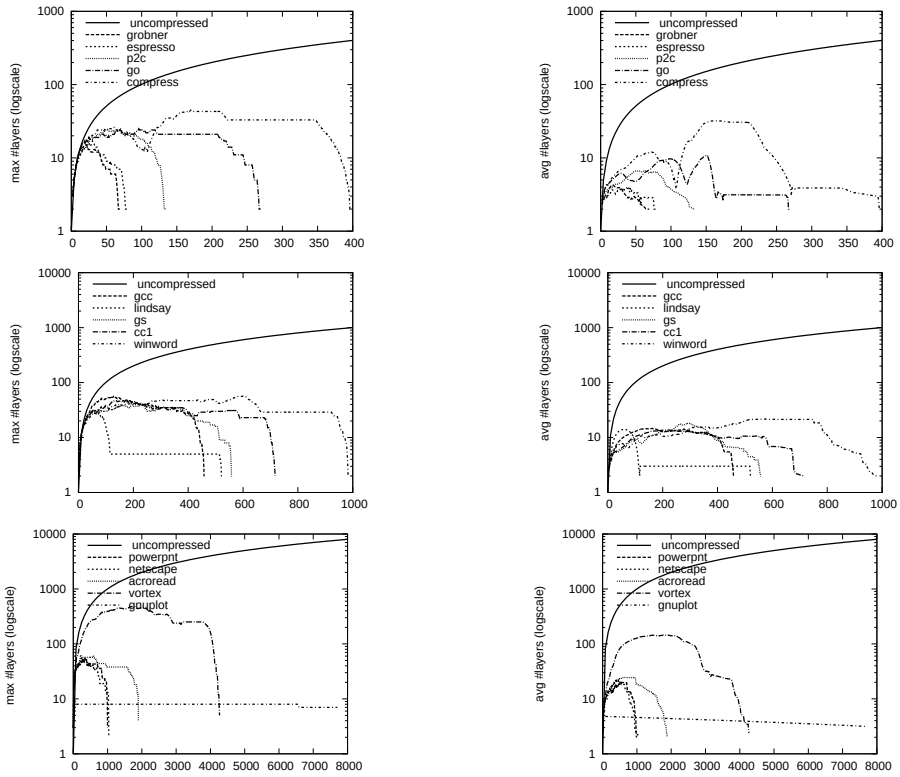


Figure 7.7: Maximum (left) and average (right) number of non-empty layers in the compressed layer partition, and the number of layers in the uncompressed partition. The x-axis is the cache size k.

The runtime results for all the traces are given in Figures 7.9, 7.10, and 7.11. 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.

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 Figures 7.12, 7.13, and 7.14 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 time for the requested page, whereas LRUlist moves elements in the recency list which triggers higher constants in the runtime, and, at times,

data cache misses. Finally, we note that RDM typically is slower than algorithm 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 time for the current page, while RDM does not increase the time counter upon revealed requests meaning that no data associated with pages changes if many consecutive revealed requests occur. This typically happens for large cache sizes.

Misses with Time Penalty. We now simulate a scenario where for each cache miss we inflict a time penalty. We choose a typical cost for a cache miss of 9ms [63, Chapter 1.3.3]. Again, we compare RDM to LRULINEAR and FIFO, where the runtime of the algorithm is 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 often dominated by the penalty for cache misses. The results in Figures 7.15, 7.16, and 7.17 show that both LRU and RDM outperform FIFO consistently. Despite the zero runtime for LRU, RDM still outperforms it for about 48% of the experiments (for each trace we considered about 100 equally distanced cache sizes; for traces having less than 100 pages all were considered). A more detailed trace-by-trace comparison between LRU and RDM is given in Figure 7.8. In particular, LRU wins for traces where its number of misses is almost equal to or better than RDM, e.g. lindsay or gnuplot, or for large cache sizes (typically larger than 75% of the pageset size), when very few misses occur and the runtime component becomes significantly more important. In most of these cases, the fact that the runtime of LRU is ignored becomes decisive.

7.4 Conclusions

In this paper we considered an integrated view of paging algorithms, where the focus is not only on the number of misses incurred, but the actual runtime of the page replacement strategy is also included. In general, algorithms with good theoretical quality guarantees outperforming LRU and FIFO in terms of page faults on real-world traces are rather complex. In the case of RDM we show that there exist implementations with realistic chances to compete in practice. These implementations can be used for arbitrary algorithms in the OnOPT class. For an OnOPT algorithm, as long as the priority assignment is not too expensive and the cache miss behavior is good, our implementation yields an algorithm with a good

Trace	LRU better (in %)	RDM better (in %)
acroread	63.7	36.3
cc1	60.5	39.5
compress	31.5	68.5
espresso	52	48
gcc	57.7	42.3
gnuplot	90.1	9.9
go	39.5	60.5
grobner	34.3	65.7
gs	69.5	30.5
lindsay	90.4	9.6
netscape	41.9	58.1
p2c	5.3	94.7
powerpoint	55.3	44.7
vortex	48.5	51.5
winword	54	46

Figure 7.8: The relative performance of LRU vs RDM for each trace, i.e. the number of cache sizes (in %) for which each of these algorithms is better, when cache misses incur a penalty of 9ms and the runtime for LRU is set to zero.

overall performance. For the given traces our engineered implementation of RDM was clearly the best compared to the other RDM implementations. Yet we believe that for traces with larger page set size e.g. 10^6 tree-based implementations (existent or new) may become competitive.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments which helped to improve the presentation of the results in this paper.

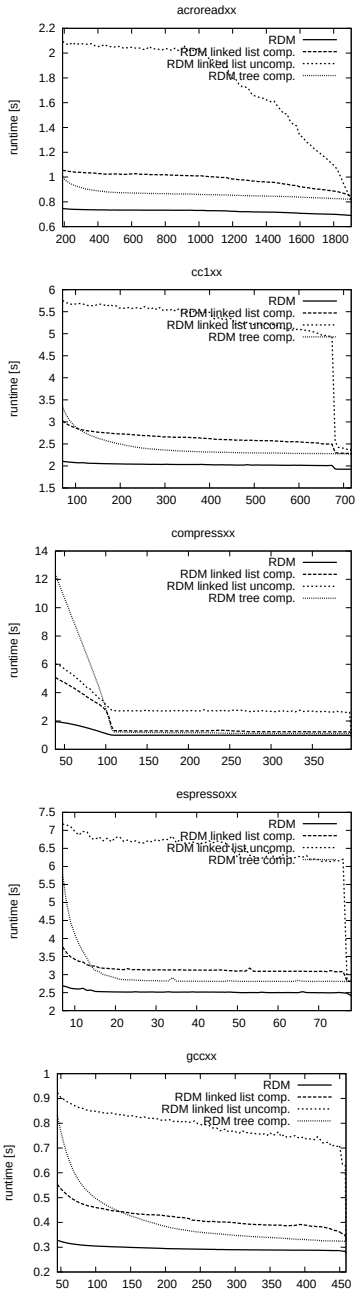


Figure 7.9: The runtime for various implementations of the OnOPT algorithms for RDM priorities.

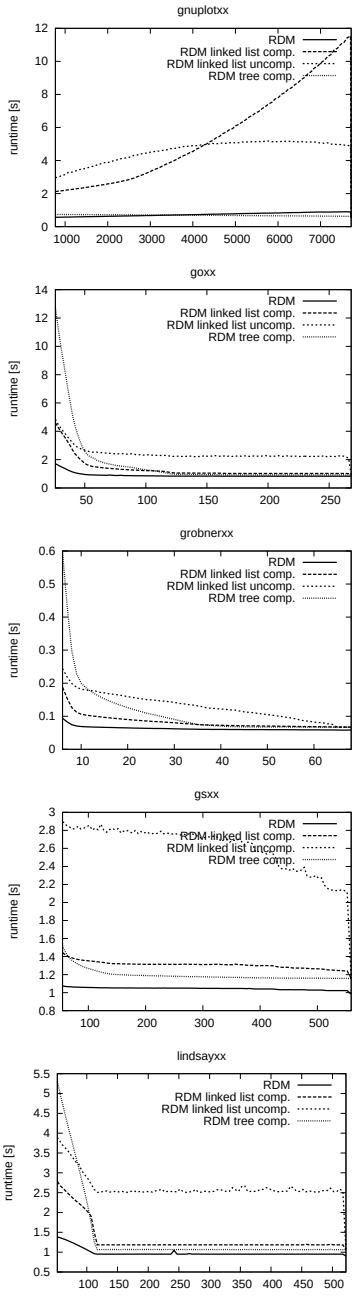


Figure 7.10: The runtime for various implementations of the OnOPT algorithms for RDM priorities.

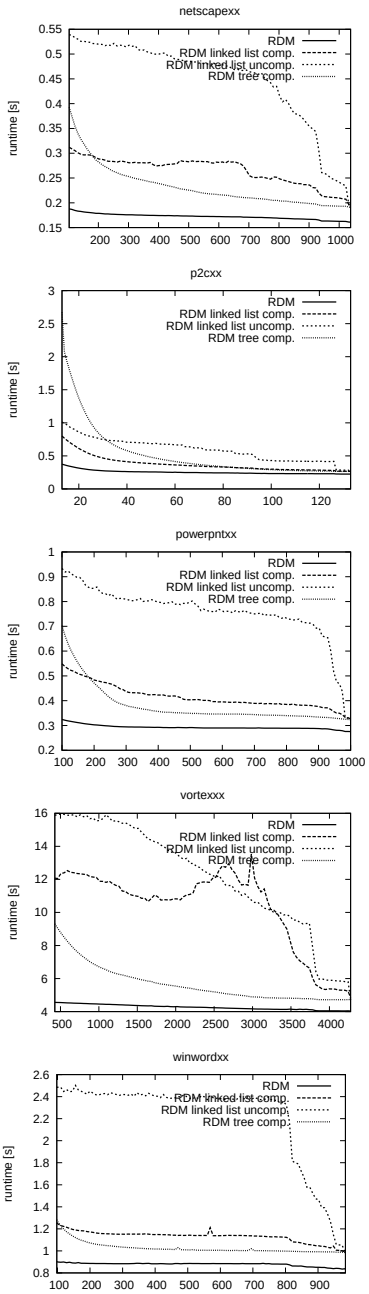


Figure 7.11: The runtime for various implementations of the OnOPT algorithms for RDM priorities.

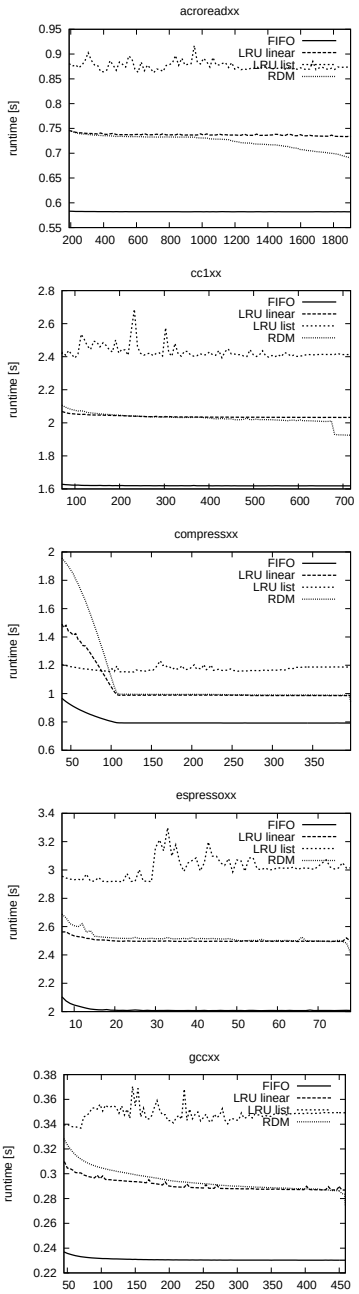


Figure 7.12: The runtime for RDM, the two implementations of LRU, and FIFO.

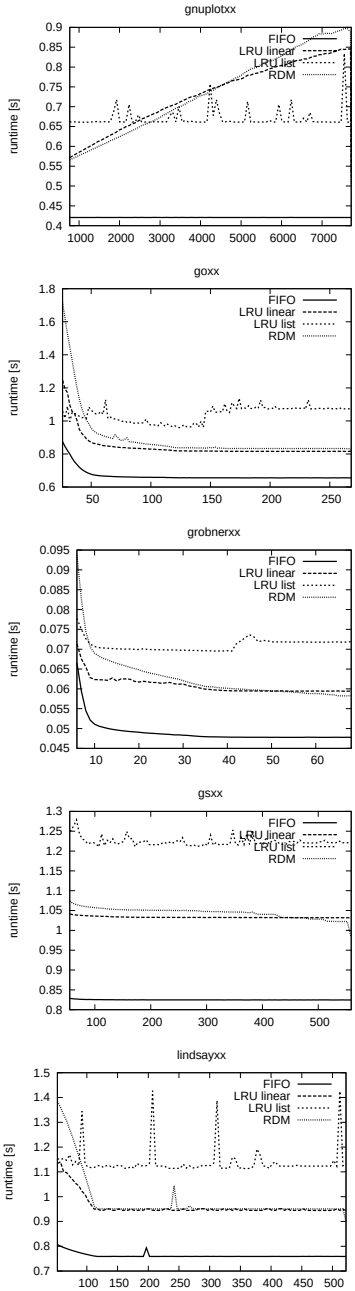


Figure 7.13: The runtime for RDM, the two implementations of LRU, and FIFO.

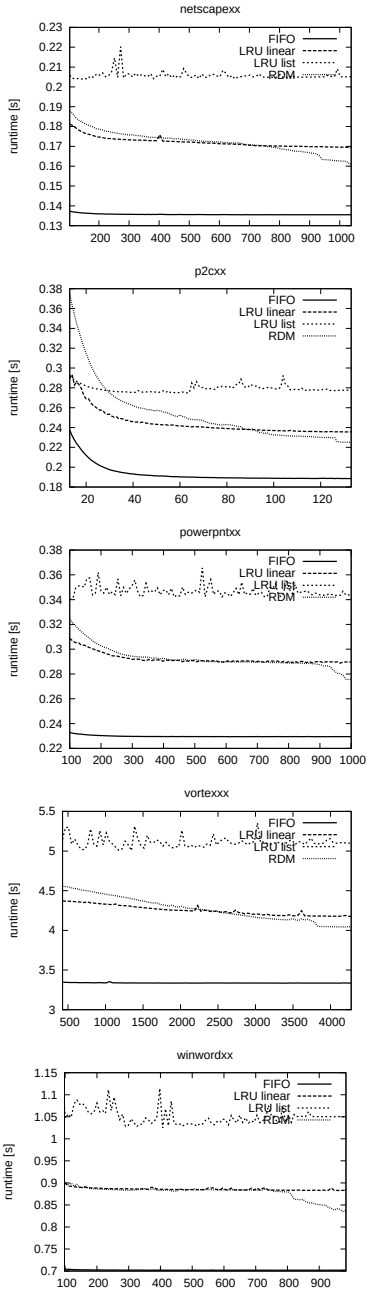


Figure 7.14: The runtime for RDM, the two implementations of LRU, and FIFO.

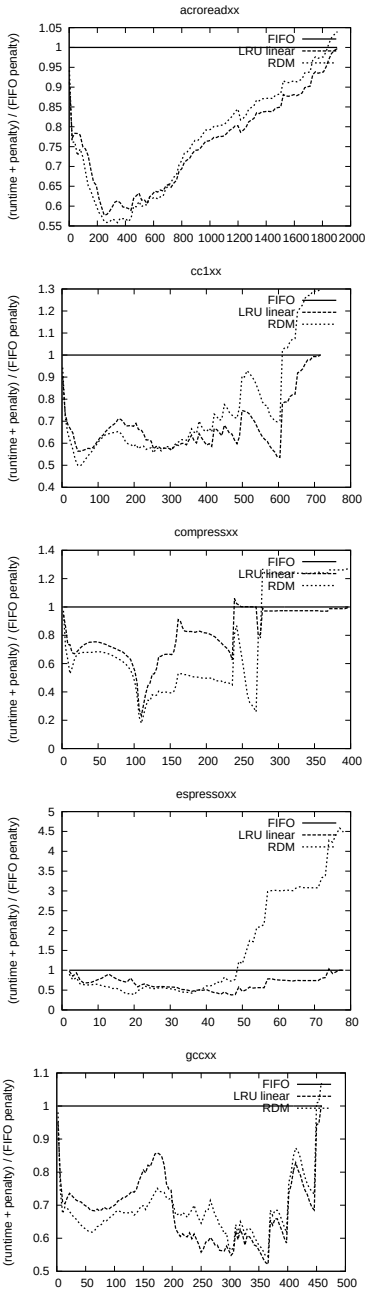


Figure 7.15: The total runtime, calculated as actual time plus cache miss penalty, of RDM and LRU compared to FIFO when a cache miss costs 9ms. The actual runtime for LRU and FIFO is set to zero.

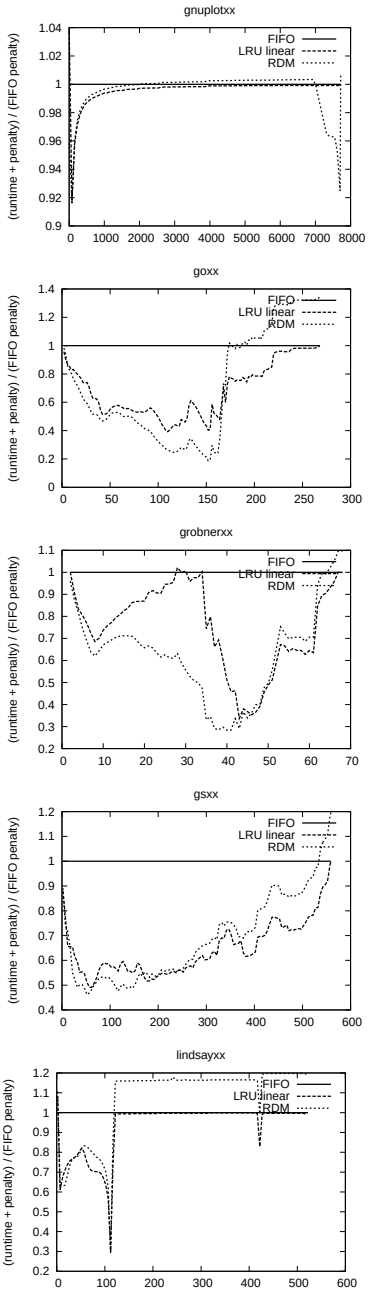


Figure 7.16: The total runtime, calculated as actual time plus cache miss penalty, of RDM and LRU compared to FIFO when a cache miss costs 9ms. The actual runtime for LRU and FIFO is set to zero.

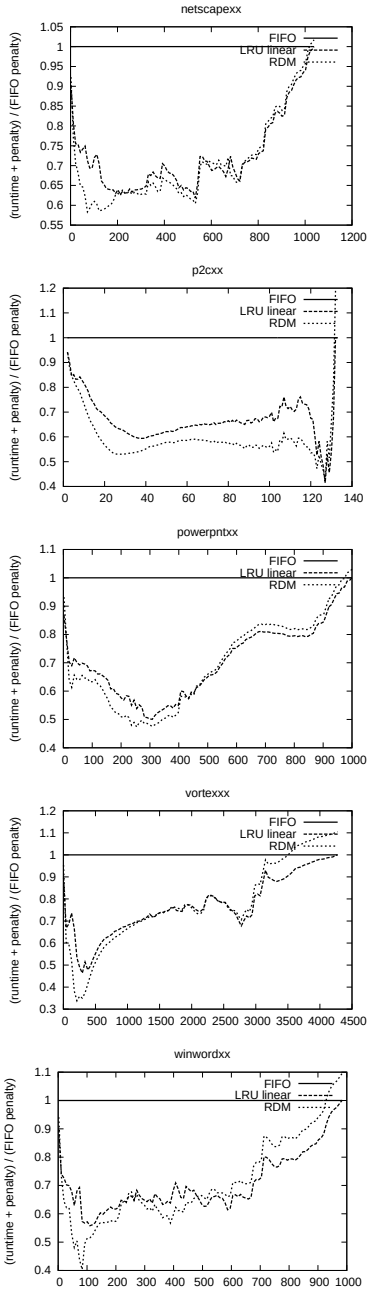


Figure 7.17: The total runtime, calculated as actual time plus cache miss penalty, of RDM and LRU compared to FIFO when a cache miss costs 9ms. The actual runtime for LRU and FIFO is set to zero.

Bibliography

[1] D. Achlioptas, M. Chrobak, and J. Noga. Competitive analysis of randomized paging algorithms. *Theoretical Computer Science*, 234(1-2):203–218, 2000.

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

[3] S. Albers, L. M. Favrholdt, and O. Giel. On paging with locality of reference. *Journal of Computer and System Sciences*, 70(2):145–175, 2005.

[4] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proc. 39th Annual Symposium on Foundations of Computer Science*, pages 534–544. IEEE Computer Society, 1998.

[5] A. Andersson and M. Thorup. Dynamic ordered sets with exponential search trees. *Journal of the ACM*, 54(3), 2007.

[6] S. Angelopoulos, R. Dorrigiv, and A. López-Ortiz. On the separation and equivalence of paging strategies. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 229–237, 2007.

[7] S. Angelopoulos and P. Schweitzer. Paging and list update under bijective analysis. In *Proc. 20th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1136–1145, 2009.

[8] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[9] W. W. Bein, R. Fleischer, and L. L. Larmore. Limited bookmark randomized online algorithms for the paging problem. *Information Processing Letters*, 76(4-6):155–162, 2000.

[10] W. W. Bein, L. L. Larmore, and J. Noga. Equitable revisited. In *Proc. 15th Annual European Symposium on Algorithms*, pages 419–426, 2007.

[11] W. W. Bein, L. L. Larmore, J. Noga, and R. Reischuk. Knowledge state algorithms. *Algorithmica*, 60(3):653–678, 2011.

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

[13] S. Ben-David and A. Borodin. A new measure for the study of on-line algorithms. *Algorithmica*, 11(1):73–91, 1994.

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

[15] A. Borodin, S. Irani, Prabhakar, and B. Schieber. Competitive paging with locality of reference. *Journal of Computer and System Sciences*, 50(2):244–258, 1995.

[16] A. Borodin, N. Linial, and M. E. Saks. An optimal on-line algorithm for metrical task system. *J. ACM*, 39(4):745–763, 1992.

[17] J. Boyar, L. M. Favrholdt, and K. S. Larsen. The relative worst-order ratio applied to paging. In *Journal of Computer and System Sciences*, volume 73, pages 818–843, 2007.

[18] R. Brause. *Betriebssysteme: Grundlagen und Konzepte*. Springer-Verlag New York Incorporated, 2003.

[19] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134. IEEE, 1999.

[20] G. S. Brodal, G. Moruz, and A. Negoescu. Onlinemin: A fast strongly competitive randomized paging algorithm. In *Proc. 9th International Workshop on Approximation and Online Algorithms:, WAOA 2011, Revised Selected Papers*, pages 164–175. Springer, 2012.

[21] G. S. Brodal, G. Moruz, and A. Negoescu. Onlinemin: A fast strongly competitive randomized paging algorithm. *Journal Theory of Computing Systems, Special issue of the 9th Workshop on Approximation and Online Algorithms*, 2013.

[22] H.-T. Chou, D. J. DeWitt, et al. *An evaluation of buffer management strategies for relational database systems*. University of Wisconsin-Madison, Computer Sciences Department, 1985.

[23] M. Chrobak, E. Koutsoupias, and J. Noga. More on randomized on-line algorithms for caching. *Theoretical Computer Science*, 290(3):1997–2008, 2003.

[24] H. M. Deitel, P. J. Deitel, and D. R. Choffnes. *Operating systems*. Pearson/Prentice Hall, 2004.

[25] R. Dementiev and L. Kettner. Stxxl: Standard template library for xxl data sets. In *In: Proc. of ESA 2005. Volume 3669 of LNCS*, pages 640–651. Springer, 2005.

[26] P. J. Denning. The working set model for program behaviour. *Communications of the ACM*, 11(5):323–333, 1968.

[27] P. J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, 6(1):64–84, 1980.

[28] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[29] R. Dorrigiv. *Alternative measures for the analysis of online algorithms*. PhD thesis, University of Waterloo, 2010.

[30] R. Dorrigiv, M. R. Ehmsen, and A. López-Ortiz. Parameterized analysis of paging and list update algorithms. In *Proc. 7th International Workshop on Approximation and Online Algorithms*, pages 104–115, 2009.

[31] A. Fiat and A. R. Karlin. Randomized and multipointer paging with locality of reference. In *Proc. 27th Annual ACM Symposium on Theory of Computing*, pages 626–634, 1995.

[32] 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.

[33] 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.

[34] A. Fiat and G. J. Woeginger, editors. *Online Algorithms, The State of the Art (the book grow out of a Dagstuhl Seminar, June 1996)*, volume 1442 of *Lecture Notes in Computer Science*. Springer, 1998.

[35] R. A. Fisher and F. Yates. *Statistical tables for biological, agricultural, and medical research (3rd edition)*. Oliver and Boyd, 1948.

[36] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[37] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, 1994.

[38] S. Irani, A. R. Karlin, and S. Phillips. Strongly competitive algorithms for paging with locality of reference. *SIAM Journal on Computing*, 25(3):477–497, 1996.

[39] 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.

[40] A. R. Karlin, M. S. Manasse, L. A. McGeoch, and S. Owicki. Competitive randomized algorithms for non-uniform problems. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 301–309, 1990.

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

[42] C. Kenyon. Best-fit bin-packing with random order. In *In 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 359–364, 1996.

[43] E. Koutsoupias and C. H. Papadimitriou. Beyond competitive analysis. In *Proc. 35th Symposium on Foundations of Computer Science*, pages 394–400, 1994.

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

[45] A. Kovács, U. Meyer, G. Moruz, and A. Negoescu. Online paging for flash memory devices. In *Proc. 20th International Symposium on Algorithms and Computation,, ISAAC 2009*, pages 352–361, 2009.

[46] M. S. Manasse, L. A. McGeoch, and D. D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11(2):208–230, 1990.

[47] E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.

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

[49] U. Meyer, A. Negoescu, and V. Weichert. New bounds for old algorithms: On the average-case behavior of classic single-source shortest-paths approaches. In *Proc. First International ICST Conference on Theory and Practice of Algorithms in (Computer) Systems, TAPAS 2011*, pages 217–228, 2011.

[50] G. Moruz and A. Negoescu. Improved space bounds for strongly competitive randomized paging algorithms. In *Proc. 40th International Colloquium on Automata, Languages, and Programming, ICALP 2013 (to appear).*

[51] G. Moruz and A. Negoescu. Outperforming LRU via competitive analysis on paramterized inputs for paging. In *Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1669–1680, 2012.

[52] G. Moruz and A. Negoescu. Improved space bounds for strongly competitive randomized paging algorithms. Technical report, Goethe-Universität Frankfurt am Main, 2013.

[53] G. Moruz, A. Negoescu, C. Neumann, and V. Weichert. Engineering efficient paging algorithms. *Journal of Experimental Algorithmics, Special issue of SEA 2012 (to appear).*

[54] G. Moruz, A. Negoescu, C. Neumann, and V. Weichert. Engineering efficient paging algorithms. In *Proc. 11th International Symposium on Experimental Algorithms*, pages 320–331, 2012.

[55] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The lru-k page replacement algorithm for database disk buffering. *SIGMOD Rec.*, 22(2):297–306, June 1993.

[56] S. Podlipnig and L. Böszörmenyi. A survey of web cache replacement strategies. *ACM Computing Surveys (CSUR)*, 35(4):374–398, 2003.

[57] P. Raghavan and M. Snir. Memory versus randomization in on-line algorithms. *Automata, Languages and Programming*, pages 687–703, 1989.

[58] G. M. Sacco and M. Schkolnick. Buffer management in relational database systems. *ACM Transactions on Database Systems (TODS)*, 11(4):473–498, 1986.

[59] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating system concepts.* J. Wiley & Sons, 2009.

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

[61] Y. Smaragdakis, S. F. Kaplan, and P. R. Wilson. The eelru adaptive replacement algorithm. *Performance Evaluation*, 53(2):93–123, 2003.

[62] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, 1981.

[63] A. S. Tanenbaum. *Modern operating systems (3. ed.)*. Pearson Education, 2008.

[64] J. Wang. A survey of web caching schemes for the internet. *ACM SIGCOMM Computer Communication Review*, 29(5):36–46, 1999.

[65] D. E. Willard. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM Journal of Computing*, 29(3):1030–1049, 2000.

[66] A. C.-C. Yao. Probabilistic computations: Toward a unified measure of complexity (extended abstract). In *FOCS*, pages 222–227, 1977.

[67] N. Young. *Competitive paging and dual-guided on-line weighted caching and matching algorithms*. PhD thesis, Citeseer, 1991.

[68] N. E. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, 1994.

Design of Competitive Paging Algorithms with Good Behaviour in Practice

von Andrei Laurian Negoescu

Diese Dissertation befasst sich mit theoretischen und praktischen Aspekten des Paging-Problems. Das Problem ist Teil der Speicherverwaltung in Systemen mit einer Speicherhierarchie, in der schneller Speicher teuer und dadurch knapp ist. Seitenersetzungsstrategien verwalten die Anordnung von Daten zwischen zwei Speicherebenen. Wir unterscheiden zwischen einem schnellen, kleinen Speicher (*Cache*) und einem langsamen, aber sehr großen Speicher (*Disk*). Wenn auf Daten zugegriffen wird, ist es wünschenswert diese bereits im Cache zu haben, um lange Zugriffszeiten auf den langsamen Speicher zu vermeiden. Wir gehen davon aus, dass die Daten in gleichgroße Stücke aufgeteilt sind, die *Speicherseiten* genannt werden. Mit k bezeichnen wir die Anzahl der Seiten, die in den Cache passen.

Das wichtigste Anwendungsgebiet stellt die Verwaltung des virtuellen Speichers durch Betriebssysteme dar [18, 24, 59, 63]. In diesem Anwendungsszenario entspricht der Cache dem schnellen Arbeitsspeicher. Wenn die Gesamtmenge an Daten die Kapazität des Arbeitsspeichers überschreitet, müssen Speicherseiten auf der viel langsameren Festplatte ausgelagert werden. Wird auf eine Speicheradresse zugegriffen und die entsprechende Seite befindet sich im Cache, so haben wir einen *Treffer* und sind fertig. Andernfalls sagen wir, dass ein *Seitenfehler* auftritt und die entsprechende Seite muss von der Festplatte in den Arbeitsspeicher geladen werden. Ein Seitenfehler verursacht zeitaufwendige I/O Operationen. Ist der Cache beim Auftreten eines Seitenfehlers voll, so muss ein Paging-Algorithmus entscheiden, welche Seite aus dem Cache auf der Festplatte ausgelagert wird. Das Ziel ist die Minimierung der Anzahl der Seitenfehler. Andere Beispiele für die Anwendung von Paging Algorithmen zur Verwaltung von Daten zwischen Arbeitsspeicher und Festplatte sind z.B. Datenbanksysteme [22, 55, 58, 62], Web-Caching [19, 56, 64] und Werkzeuge für Externspeicher-Algorithmen wie die C++ Bibliothek STXXL [25].

Wettbewerbsfaktor. Eine optimale Seitenersetzungsstrategie lagert die Seite aus dem Cache aus, die am entferntesten in der Zukunft wieder angefragt wird. Diese Strategie ist jedoch nicht praktisch durchführbar aufgrund der Tatsache, dass normalerweise nur wenig oder überhaupt nichts über zukünftige Zugriffsmuster bekannt ist. Deshalb ist Paging eines der am meisten untersuchten Probleme auf dem Gebiet der Online-Algorithmen. Bei Online-Algorithmen handelt es sich um Algorithmen, die ohne vorherige Kenntnis der gesamten Eingabe unwiderrufliche Entscheidungen treffen müssen.

Der erste Teil dieser Dissertation behandelt die theoretische Analyse von randomisierten Paging-Algorithmen. Wir benutzen das klassische Evaluierungsmo-

dell der Kompetitiven Analyse [60]. Ein deterministischer Algorithmus A ist *c-kompetitiv* oder hat den *Wettbewerbsfaktor* c, wenn für jede Eingabesequenz gilt, dass

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

Dabei entspricht $cost(A)$ der Anzahl Seitenfehler von A, $cost(OPT)$ der Anzahl Seitenfehler der optimalen offline Lösung und b ist eine Konstante. Beachte, dass ein offline Algorithmus bei seinen Entscheidungen Zugriff auf die gesamte Eingabe hat. Ist A ein randomisierter Algorithmus so bezeichnet $cost(A)$ die erwartete Anzahl Seitenfehler und die Definition des Wettbewerbsfaktors folgt analog. Ein Online-Algorithmus ist *optimal kompetitiv*, wenn dieser den bestmöglichen Wettbewerbsfaktor hat.

Im Fall von deterministischen Algorithmen ist für den Wettbewerbsfaktor die untere Schranke von k bekannt [60]. Mehrere Algorithmen haben diesen optimalen Wettbewerbsfaktor, unter den bekanntesten sind LRU (Least Recently Used), FIFO (First In First Out) and FWF (Flush When Full) [60]. Randomisierte Algorithmen haben einen Wettbewerbsfaktor von mindestens H_k [32], wobei $H_k = \sum_{i=1}^{k} 1/i$ die k-te harmonische Zahl ist. In derselben Arbeit wurde der randomisierte Algorithmus MARK vorgestellt und bewiesen, dass dieser $2H_k$-kompetitiv ist. Obwohl MARK nicht den optimalen Wettbewerbsfaktor erreicht, hat es den Vorteil ein einfacher und schneller Algorithmus zu sein. Später haben Achlioptas et al. [1] den exakten Wettbewerbsfaktor von $2H_k - 1$ für MARK bestimmt.

Der erste H_k-kompetitive randomisierte Algorithmus PARTITION wurde von McGeoch and Slater [48] vorgestellt. Speicherbedarf und Laufzeitkomplexität von PARTITION sind nicht durch die Größe k des Caches begrenzt. Obwohl die Minimierung der Anzahl von Seitenfehlern bei der kompetitiven Analyse im Vordergrund steht, spielen für den Einsatz in der Praxis Speicherbedarf und Laufzeit eine wichtige Rolle. Der erste optimal kompetitive Algorithmus, der einen Fortschritt in Richtung Effizienz darstellte, war EQUITABLE [1]. EQUITABLE kann in $O(k^2)$ Zeit eine Seitenanfrage bearbeiten und hat einen Speicherbedarf von $O(k^2 \log k)$.

Borodin and El-Yaniv haben in ihrem Buch [14] das offene Problem aufgelistet, ob es H_k-kompetitive Algorithmen gibt, die mit einem Speicherbedarf von $O(k)$ auskommen. Bein et al. [10] haben dieses Problem gelöst, indem sie den Algorithmus EQUITABLE2 vorgestellt haben. Dieser ist eine Variante von EQUITABLE welcher nur $2k$ Bookmarks benutzt, sich also Informationen über maximal $2k$ Seiten merkt, die nicht im Cache sind. In derselben Arbeit wurde die Vermutung aufgestellt, dass $o(k)$ Bookmarks möglich sind. Obwohl EQUITABLE2 nur $O(k)$ Speicherbedarf hat, wird weiterhin im worst-case $O(k^2)$ Zeit für die Verarbeitung einer Seitenanfrage benötigt.

Ergebnisse I. In dieser Dissertation stellen wir den optimal kompetitiven Algorithmus OnlineMin vor, der eine Seitenanfrage in worst-case Zeit $O(\log k)$ [20] verarbeitet und einen Speicherbedarf von $O(k)$ hat. Wir verbessern die Laufzeit auf $O(\log k / \log\log k)$ in [21] durch Ausnutzen der Mächtigkeit des RAM Modells. Die beste vorher bekannte Laufzeit bei gleichem Speicherbedarf war $O(k^2)$ [10] .

Weiterhin verbessern wir die Analyse von Equitable2 und zeigen, dass nur ungefähr $0.62k$ Bookmarks notwendig sind, beweisen jedoch dass $o(k)$ Bookmarks im Fall von Equitable2 nicht möglich sind. Stattdessen führen wir eine Variante von Partition ein, welche wir Partition2 bezeichnen. Wir zeigen dass Partition2 optimal kompetitiv ist und $\Theta(k/\log k)$ Bookmarks benötigt. Dadurch beweisen wir die $o(k)$ Bookmark Vermutung [10].

Theorie vs. Praxis. Der zweite Teil dieser Arbeit befasst sich mit der Diskrepanz zwischen Theorie und Praxis im Fall des Paging-Problems. Die Kompetitive Analyse wird oft für die zu pessimistischen Qualitätsgarantien von deterministischen Paging-Algorithmen kritisiert. Young [67] hat den *empirischen Wettbewerbsfaktor* untersucht, d.h. das Verhältnis zwischen den Kosten eines Online-Algorithmus und den Kosten eines optimalen Offline-Algorithmus auf Eingaben aus der Praxis. Der empirische Wettbewerbsfaktor von LRU ist eine, von k unabhängige, kleine Konstante (≈ 4), wohingegen die Kompetitive Analyse einen Wettbewerbsfaktor von k ergibt. Unterschiede von Faktor 100 und mehr wurden zwischen den experimentellen Ergebnissen und der theoretischen Schranke beobachtet. Ein anderer Kritikpunkt ist die Tatsache, dass die Kompetitive Analyse den Algorithmus LRU nicht von FWF oder FIFO separiert, obwohl aus der Praxis bekannt ist, dass LRU deutlich besser abschneidet als die beiden anderen Algorithmen [14].

Die aktuelle Forschung beschäftigt sich intensiv mit alternativen Modellen zur Evaluierung von Online-Algorithmen im allgemeinen und Paging im Speziellen. Eine Forschungsrichtung macht Einschränkungen über die Eingabe bei der kompetitiven Analyse, wie *Diffuse Adversary* [44] oder *Loose Competitiveness* [68]. Andere Ansätze vergleichen Online-Algorithmen direkt miteinander, ohne einen direkten Vergleich mit der optimalen offline Lösung. Relevante Beispiele umfassen *Max/Max Ratio* [13], *Random Order Ratio* [42], *Relative Worst Order Ratio* [17], *Bijective Analysis* und *Average Analysis* [6]. Einen ausführlichen Überblick über alternative Evaluierungsmethoden liefert die Arbeit von Dorrigiv [29]. Die meisten der neuen Ansätze sind damit beschäftigt, bereits existierende Online-Algorithmen nach ihrer Performance zu ordnen, so dass diese Ordnung mit den Beobachtungen aus der Praxis übereinstimmt. Insbesondere gilt für viele Ansätze (z.B. Diffuse Adversary, Bijective Analysis kombiniert mit der Lokalitätseigenschaft [7]), dass sie LRU als den besten Algorithmus im jeweiligen Modell ausmachen. Nur selten haben die alternativen Modelle das Design neuer Algorithmen mit einer niedrigen Anzahl an Seitenfehlern in der Praxis motiviert.

Beispiele dafür sind RLRU (Retrospective LRU) [17] und FARL (Farthest-To-Last-Request) [14,33], welche durch die Relative Worst Order Ratio und entsprechend das Access Graph Modell [15,31,38] motiviert wurden.

Ergebnisse II. Wir führen die *Attack Rate* r ein [51]. Dabei handelt es sich um eine Parametrisierung der Eingabe für die kompetitive Analyse des Paging-Problems. Der Parameter r ist abhängig von der Eingabe, variiert zwischen 1 und k und entspricht in etwa dem Grad an Ungewissheit, welche Seiten der optimale Offline-Algorithmus im Cache hat. Im Gegensatz zu vielen anderen Ansätzen machen wir nicht die LRU-freundliche Annahme der Lokalitätseigenschaft. Wir beweisen, dass r eine untere und obere Schranke für den Wettbewerbsfaktor von deterministischen Online-Algorithmen ist. Der optimale Wettbewerbsfaktor r wird von LRU und FIFO erreicht, aber nicht von FWF. Experimente auf Anfragesequenzen aus der Praxis zeigen, dass der Wert von r meistens sehr viel kleiner als k ist. Somit ist für LRU und FIFO der Unterschied zwischen dem empirischen Wettbewerbsfaktor und r deutlich kleiner als in der klassischen kompetitiven Analyse.

Eine in der Praxis geläufigere Kennzahl für die Güte eines Paging-Algorithmus ist das Verhältnis zwischen Anzahl Seitenfehler und der Länge der Eingabe, die so genannte *Fehlerrate*. Unsere Parametrisierung führt zu oberen Schranken für die Fehlerrate von r-kompetitiven Algorithmen. Experimente zeigen, dass unsere parametrisierte Schranke die niedrige Fehlerrate von LRU deutlich besser erklärt als die parametrisierten Schranken von Albers et al. [3] und Dorrigiv et al. [30], welche auf dem Lokalitätsprinzip beruhen.

Bekannte Algorithmen nach ihren Resultaten in der Praxis zu ordnen und/oder ihre Performance abzuschätzen ist eine Aufgabe von Evaluierungsmodellen, welche von vielen (zumindest teilweise) erfolgreich gemeistert wird. Eine andere wünschenswerte Eigenschaft ist die Anregung neuer Algorithmen, welche eine gute Performance in der Praxis liefern. Wir entwerfen die OnOPT Klasse, welche r-kompetitive Algorithmen enthält, und LRU beinhaltet. Aus dieser Klasse extrahieren wir den neuen Algorithmus RDM und führen Experimente auf praxisrelevanten Eingabesequenzen durch. Die experimentellen Ergebnisse zeigen, dass RDM die Leistung von LRU und einige seiner Varianten auf vielen Eingaben übertrifft. Dies steht im Gegensatz zu neueren Modellen, die LRU als den besten Algorithmus herausstellen. Es ist zu beachten, dass die OnOPT Klasse stark durch die klassische kompetitive Analyse inspiriert ist. Dies zeigt, dass Einblicke aus der kompetitiven Analyse helfen kann, neue Algorithmen zu entwerfen, die gute Ergebnisse auf praxisrelevanten Eingaben liefern.

Das Hauptziel von Paging-Algorithmen ist die Minimierung der Anzahl von Seitenfehlern, da ein Seitenfehler zu einer Verzögerung von mehreren Millisekunden in der Programmausführung führen kann. Man darf jedoch nicht die Laufzeit des Paging-Algorithmus selbst vernachlässigen. Haben wir einen Algorithmus mit

wenigen Seitenfehlern, der jedoch eine immense Berechnungszeit verschlingt, so kann es von Vorteil sein stattdessen einen einfachen Algorithmus mit mehr Seitenfehlern zu benutzen, wie z.B. FIFO. Wir entwickeln eine effiziente Implementierung von RDM und vergleichen diesen mit anderen Algorithmen bezüglich ihrer *Gesamtleistung* [54]. Die Gesamtleistung besteht aus der Laufzeit des Paging-Algorithmus und einer üblichen Zeitstrafe von 9 ms für jeden Seitenfehler. Die Experimente zeigen, dass RDM die zwei Algorithmen LRU und FIFO auch in der Gesamtleistung übertrifft, sogar wenn wir den Konkurrenten von RDM eine idealistische Laufzeit von 0 ms anrechnen.

Publikationen. Die Ergebnisse dieser Dissertation basieren auf vier Konferenzpapiere und zwei Journal-Publikationen (alle 6 peer-reviewed). Ich war bei allen sechs Publikationen Hauptautor.

[20] G. S. Brodal, G. Moruz, and A. Negoescu. Onlinemin: A fast strongly competitive randomized paging algorithm. In *Proc. 9th International Workshop on Approximation and Online Algorithms:, WAOA 2011, Revised Selected Papers*, pages 164–175. Springer, 2012

[51] G. Moruz and A. Negoescu. Outperforming LRU via competitive analysis on paramterized inputs for paging. In *Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1669–1680, 2012

[54] G. Moruz, A. Negoescu, C. Neumann, and V. Weichert. Engineering efficient paging algorithms. In *Proc. 11th International Symposium on Experimental Algorithms*, pages 320–331, 2012

[21] G. S. Brodal, G. Moruz, and A. Negoescu. Onlinemin: A fast strongly competitive randomized paging algorithm. *Journal Theory of Computing Systems, Special issue of the 9th Workshop on Approximation and Online Algorithms*, 2013

[53] G. Moruz, A. Negoescu, C. Neumann, and V. Weichert. Engineering efficient paging algorithms. *Journal of Experimental Algorithmics, Special issue of SEA 2012 (to appear)*

[50] G. Moruz and A. Negoescu. Improved space bounds for strongly competitive randomized paging algorithms. In *Proc. 40th International Colloquium on Automata, Languages, and Programming, ICALP 2013 (to appear)*

Aufbau. Die Dissertation ist folgendermaßen aufgebaut: In Kapitel 2 führen wir die wichtigsten Konzepte von Online-Algorithmen ein. Wir beschreiben die wichtigsten Paging-Algorithmen und nehmen Bezug auf ihren Wettbewerbsfaktor. Als einen vorbereitenden Schritt für unsere experimentellen Ergebnisse, führen

wir drei existierende Modelle zur Analyse der Fehlerrate ein. Dieses Kapitel endet mit einem Überblick über die Zustände des optimalen Offline-Algorithmus, einem wichtigen Bestandteil unserer Ergebnisse. In Kapitel 3 werden die Resultate unserer Forschung präsentiert. Wir beginnen mit den theoretischen Ergebnissen zur Laufzeit [20,21] und dem Speicherverbrauch [50] von Algorithmen mit optimalem Wettbewerbsfaktor. Als nächstes führen wir die Attack Rate r ein [51] sowie den Algorithmus RDM. Wir schließen das dritte Kapitel mit der Beschreibung einer effizienten Implementierung von RDM ab [53, 54]. Kapitel 4-7 enthalten unsere wissenschaftlichen Arbeiten.

Curriculum Vitae

Persönliche Daten

Andrei Laurian Negoescu
Luisenstr. 28
63067 Offenbach am Main

Tel.: (+49) 163 9067498
E-Mail: negoescu@cs.uni-frankfurt.de

Geb. am 02. 10. 1981 in Bukarest, Rumänien

Schulbildung

1994-2001	Herderschule Frankfurt, Abitur (1.7) (Leistungskurs Mathematik und Physik)

Studium

10/2001–10/2007	Diplom in Informatik an der Goethe Universität Frankfurt, Note: *sehr gut*, ausgezeichnet als eines der besten zwei Diplome im Studiengang Informatik
	Diplomarbeit *Queue Management für DiffServ*: Note *sehr gut* Betreuer: Prof. Dr. Georg Schnitger
seit 08/2010	Promotionsstudent an der Goethe Universität Frankfurt Dissertation *Design of Competitive Paging Algorithms with Good Behaviour in Practice*, eingereicht am 7.Mai 2013 Betreuer: Prof. Dr. Ulrich Meyer

Berufserfahrung

04/2005–06/2007	Halm GmbH, studendische Hilfskraft: Inbetriebnahme Photovoltaic Meßsysteme, Internetauftritt
seit 08/2007	Goethe Universität Frankfurt am Main, wissenschaftlicher Mitarbeiter am Lehrstuhl Algorithm Engineering bei Prof. Dr. Ulrich Meyer

Die fünf wichtigsten Publikationen

G. Moruz and A. Negoescu. Outperforming LRU via competitive analysis on paramterized inputs for paging. In *Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1669–1680, 2012

G. S. Brodal, G. Moruz, and A. Negoescu. Onlinemin: A fast strongly competitive randomized paging algorithm. *Journal Theory of Computing Systems, Special issue of the 9th Workshop on Approximation and Online Algorithms*, 2013

G. Moruz, A. Negoescu, C. Neumann, and V. Weichert. Engineering efficient paging algorithms. In *Proc. 11th International Symposium on Experimental Algorithms*, pages 320–331, 2012

G. Moruz and A. Negoescu. Improved space bounds for strongly competitive randomized paging algorithms. In *Proc. 40th International Colloquium on Automata, Languages, and Programming, ICALP 2013 (to appear)*

A. Kovács, U. Meyer, G. Moruz, and A. Negoescu. Online paging for flash memory devices. In *Proc. 20th International Symposium on Algorithms and Computation,, ISAAC 2009*, pages 352–361, 2009

Andere Publikationen

U. Meyer, A. Negoescu, and V. Weichert. New bounds for old algorithms: On the average-case behavior of classic single-source shortest-paths approaches. In *Proc. First International ICST Conference on Theory and Practice of Algorithms in (Computer) Systems, TAPAS 2011*, pages 217–228, 2011

G. S. Brodal, G. Moruz, and A. Negoescu. Onlinemin: A fast strongly competitive randomized paging algorithm. In *Proc. 9th International Workshop on Approximation and Online Algorithms:, WAOA 2011, Revised Selected Papers*, pages 164–175. Springer, 2012

G. Moruz, A. Negoescu, C. Neumann, and V. Weichert. Engineering efficient paging algorithms. *Journal of Experimental Algorithmics, Special issue of SEA 2012 (to appear)*

Mitarbeit in der Lehre (Übungsbetrieb)

Algorithmentheorie: WS 07/08, WS 09/10, WS 11/12

Datenstrukturen: SS 08, SS 09, SS 10

Parallel and Distributed Algorithms: WS 08/09, WS 10/11

Approximationsalgorithmen: WS 12/13

Effiziente Algorithmen: SS 13

Programmiererfahrung

Java, C/C++

Fremdsprachen

Englisch, Rumänisch, Französisch