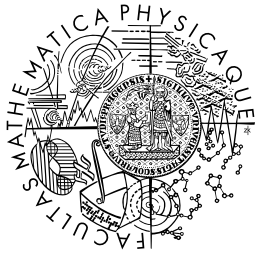

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Karel Král

Data structure behavior with variable cache size

Computer Science Institute of Charles University

Supervisor of the master thesis:	doc. Mgr. Michal Koucký, Ph.D
Study programme:	Computer Science
Study branch:	Discrete Models and Algorithms

Prague 2017

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Data structure behavior with variable cache size

Author: Karel Král

Institute: Computer Science Institute of Charles University

Supervisor: doc. Mgr. Michal Koucký, Ph.D, Computer Science Institute of Charles University

Abstract: Cache-oblivious algorithms are well understood when the cache size remains constant. Recently variable cache sizes have been considered. We are motivated by programs running in pseudo-parallel and competing for a single cache. This thesis studies the underlying cache model and gives a generalization of two models considered in the literature. We give a new cache model called the "depth model" where pages are accessed by page depths in an LRU cache instead of their addresses. This model allows us to construct cache-oblivious algorithms that cause a certain number of cache misses prescribed by an arbitrary function computable without causing a cache miss. Finally we prove that two algorithms satisfying the regularity property running in pseudo-parallel cause asymptotically the same number of cache misses as their serial computations provided that the cache is satisfying the tall-cache assumption.

Keywords: cache-oblivious data structures

I would like to thank my supervisor for his patience, valuable advice, all the time spent with me thinking about different mathematical problems, and questions that always turned out to be helpful and interesting. I am also grateful for all the support of my family and all of the people close to me. I thank Veronika Slívová for proofreading this thesis and her valuable comments.

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement n. 616787.

The research was also partially supported by a grant from Neuron Fund for Support of Science.

Contents

Introduction

Modern computer architectures use several layers of cache. Perhaps the most widely used model for modelling this is the two level memory consisting of a large memory and a smaller cache. If the requested page is in the cache the processor gets it almost immediately otherwise it has to wait for a constant amount of time. Usually we consider operations of the CPU and cache as short enough to omit and we only care about the number of accesses to the main memory.

Traditionally cache models deal just with a cache that does not change during the run of a program. With parallel computing and cloud computing there are more processes sharing resources at one time. This leads to a model where the cache size may fluctuate. An example may be a RAM memory viewed as a hard-disk cache where there are other programs competing for the RAM. Even cloud computing provides us with an example of such problem as the amount of memory available to an individual virtual machine may change over time due to the load of other virtual machines.

In the first two chapters we give a brief overview of relevant results. In Chapter 3 we give a detailed discussion of cache models for fluctuating sizes. We prove that the least recently used (LRU) caching algorithm is competitive with the optimal one in two different models. One of them is similar to the model introduced independently by Peserico [2013]. The second model is a common generalization of the previous one and the model studied by Bender et al. [2014]. This generalized model is important because it faithfully models cache size changes caused by other programs.

We also provide a novel view at the problem of caching where we assume that the program makes page accesses by depth of the page in an infinite cache instead of page numbers. We give a justification of this model and prove that this model and the classical model have the same power. Moreover we also prove that if the accessed page address can be computed using constantly many variables we can simulate one model in the other with no additional cache misses. Using this framework we are also able to create cache-oblivious programs with almost arbitrary cache behaviours. We are thus able to say something about cache constructible functions as a parallel to time or size constructible functions studied in complexity theory.

Chapter 4 contains a discussion of the most practical model – two algorithms competing for one cache. We note that this depends heavily on the properties of the competing algorithms. Namely we consider two basic properties – the tall cache assumption and regularity. We give a proof that when two cache-oblivious algorithms compete, their tall cache assumptions are satisfied, and they satisfy the regularity condition the number of cache misses is asymptotically the same as the number of cache misses caused when one runs after the other finishes.

1. Previous Models

We give a brief introduction to the variety of problems connected to caching that have been considered. It is not our goal to give a comprehensive list of all the directions of research done. But we at least mention some of the most basic directions and historical development. We point the interested reader to the survey on cache-oblivious algorithms by Demaine [2002].

1.1 History

There has not yet been a data storage that would be both fast enough and large enough at the same time, no matter if we are considering registers on a processor, CPU caches, RAM (random-access memory), hard drives or disc arrays. There is always demand for faster and larger ones. One way to mitigate this problem is to introduce an intermediate smaller but faster memory to cache chunks (pages) of the accessed data and hope that the program will request them repeatedly or will access their immediate neighbours located on the same page.

These considerations lead to the model where the memory consists of two parts one that is close to the processor, cheap to access but limited in space and second one that is much more far away from the processor but virtually with no limit on space.

One of the earliest works was done by Aggarwal et al. [1988] in the *external-memory* model. They started with algorithms for sorting, fast Fourier transform (FFT), permutation networks, permuting and matrix transposition. Several of these algorithms, namely sorting, FFT and matrix transposition, have been considered many times since then.

Not that long after them Frigo et al. [1999] consider their *cache-oblivious* model where algorithms do not depend on the cache parameters like the size and page length but one can still prove that they perform very well with respect to the cache and page size. Not knowing the cache parameters goes hand in hand with deferring page evictions to the cache itself. One of the most basic observations there is that there is an implementable eviction policy that behaves well enough. We discuss this problem in our model in Chapter 3.

One important characteristic of a cache is the *tall cache* assumption which is mostly satisfied in practice. It is used by many cache-oblivious algorithms.

Definition 1. *We say that a cache of size M with pages of size B satisfies the* tall cache *assumption if $M \in \Omega(B^2)$.*

The motivation for the tall cache assumption is that we can utilize the cache better when there is enough room for enough pages. The most basic example is transposition of a matrix in row-major order where we heavily use that a square submatrix can fit in the cache at one time. The tall cache assumption is usually true in practice. We assume the tall cache assumption unless stated otherwise. For many algorithms the tall cache assumption can be weakened to $M \in \Omega(B^{1+\varepsilon})$ for some $\varepsilon > 0$ (for more see Demaine [2002]). Nevertheless we use the term tall cache assumption as defined above.

1.2 Architecture of Computer Caches

We give a brief overview of caching architectures and the idealized cache model. We are not as much interested in specific values as in the ratios between them. The main trend is that the closer to the processor the cache is the faster but smaller it gets. We present only the bare minimum of cache types present in almost all common computers. The numbers and memory hierarchy is taken mainly from the book of Bryant et al. [2003] and specifications of main CPU and hardware producers.

The cache closest to the central processing unit is called CPU cache and most of the modern CPUs have multiple levels. Mostly we are talking about L1 and L2 caches. The size of L1 cache tends to be between 8kB to 128kB. The size of the L2 cache is larger than the size of the L1 cache. Sometimes even L3 cache is present and its size in modern computers is roughly 8MB but the sizes vary up to 128MB. The L1 and L2 caches tend to not be shared on multi-core CPUs but L3 cache tends to be shared between the cores.

The RAM can also be viewed as a form of disk cache. Modern hard disks come with cache included, so called disk buffers. This is another cache present in the computer. It is a common scenario that several computers are connected to a network and they use a distributed file system. One could then view a single hard drive as a cache of this file system.

The access times vary and generally are between 1 to 30 processor cycles when the page is present in the CPU cache, 50 to 200 cycles when it is present in the RAM and tens of millions cycles if the page is swapped on the hard drive.

We are using an idealized model where all caches are fully associative – that means that each page can be cached to any place in the cache. This is usually not true for real computer caches but there has been work to make the theoretical results applicable even in this situation.

2. Previous Results

In this work we are interested in models where the cache size fluctuates during the computation. This has been considered before. Here we name just few papers that have been most influential and related to our work. Another interesting area of research are parallel algorithms and studying their cache performance.

All of these papers consider caches that are not maintained by the algorithm itself but use some caching algorithm to decide which pages to keep in the cache and which pages to evict. Perhaps the most well known caching algorithm is the least recently used (LRU) algorithm. LRU also shows to be competitive when the cache size fluctuates.

There are two approaches that differ by how they measure time. Peserico [2013] uses page requests to measure time and cache size changes happen between two consecutive page requests. The time used by the Cache-Adaptive model of Bender et al. [2014] is measured solely by cache misses. Both of these models have their downsides. On the modelled computer cache size changes are caused by another algorithm or algorithms that also use the same cache. Thus the first model is considering size changes that happen each page request and thus it does not entirely faithfully model the influence of other running programs. The model considering that everything that happens between two cache misses takes no time at all which is also a bit idealized model. We establish LRU competitiveness on a common generalization of these models and thus provide a bit more realistic model.

2.1 Synchronized Caching Algorithms

Peserico [2013] studies a lot of different caching algorithms in the model where time is determined by page requests (regardless whether they cause a cache miss or not). Peserico [2013] shows there is a caching algorithm that has a good competitive ratio in the classical setting with constant size caches and arbitrarily un-optimal competitive ratio when the cache sizes change. On the other hand the paper also contains a result similar to ours in Section 3.1 that LRU is competitive.

Menache and Singh [2015] consider an interesting case of the previous model. They investigate algorithms running in a cloud setting with convex cost function associated with each program. The goal is to decide in real-time how to allocate the memory resources.

2.2 Cache-Adaptive Model

This section reviews results of Bender et al. [2014], Bender et al. [2016], Lincoln [2014], and Soorchaei [2015]. The time in their model changes only after a cache miss (this is equivalent to unit time cache misses and zero time cache hits). They provide proofs of several results about optimality in their model. These papers use the term *memory profile* to describe cache sizes over time. Under a memory profile m the cache has size $m(t)$ at time $t \in \mathbb{N}$. We are going to define and review some of their results. Bender et al. [2016] also give a recipe

similar to master theorem for analysing divide and conquer algorithms in their cache-adaptive model, but we omit their statements here.

We use the following definition later in this text.

Definition 2 (Bender et al. [2014])**.** *We say that a memory profile is a square profile if there are boundaries* $0 = t_0 < t_1 < \dots$ *such that for all* $t \in [t_i, t_{i+1}]$ *the cache size is* $m(t) = t_{i+1} - t_i$*. In other words the memory profile is a step function with steps that are as tall as they are long.*

Bender et al. [2014, 2016], Lincoln [2014], Soorchaei [2015] define optimality in their Cache-Adaptive model and prove that a non-trivial class of algorithms is optimal. They also prove the optimality of several other algorithms that do not belong to this class. Moreover they provide a framework similar to the Master Theorem method to analyse a wide variety of algorithms in the Cache-Adaptive model.

2.3 Parallel Algorithms

There has been a lot of work on parallel algorithms and their cache performance. Although this is a bit different as this area considers the model where there is one program running several threads and computing one result. The parallel threads however share caches and thus compete with each other. On the other hand we already know what is going on in the computer and there are no surprising cache size fluctuations.

Communication between processors seems to be the critical bottleneck in parallel versions of Strassen algorithm on modern systems. Ballard et al. [2012] study this problem and provide an algorithm that is communication-optimal. Their result also outperforms other algorithms in practice.

There is an extensive amount of research done on parallel algorithms and their cache performance. Even several cache architecture layers and layouts have been considered. A proper overview of this theory would be too long for this chapter and moreover this area is not that connected to our case.

3. Cache Models

In this chapter we address the problem of the underlying model. As well as Frigo et al. [1999] (and Prokop [1999] in his PhD thesis) we would like to start by stating that the *Least Recent Used* (LRU) algorithm behaves almost as good as the optimal one. But in the situation when the cache size may change its size we need to specify exactly when do the cache size changes occur. There are basically three different possibilities:

1. Both cache miss and cache hit take one unit of time but we would like to bound the number of cache misses made by the LRU algorithm. We deal with this model in Section 3.1 where we show the optimality of LRU caching. We obtained our results independently of Peserico [2013] who has similar results. Peserico also studies more caching algorithms under the same model. We have decided to keep this section as our proof is slightly different and we build on it in the next section.

2. Each cache miss lasts one unit of time and each cache hit lasts zero units of time. The optimality of LRU caching is studied by Bender et al. [2014] and in even more depth by Soorchaei [2015].

3. There is a common generalization of the previous two models where a cache hit lasts one unit of time and a cache miss lasts M units of time. The cache size changes happen in integer times in these time units. We consider this model in Section 3.2 and we also establish that LRU is competitive.

In Section 3.3 we continue by an observation about a general class of caching algorithms which behave nicely provided that the cache size is always at least some constant.

We conclude this chapter by providing the *Depth Cache* model which gives another view at the problem of page accesses and provides an easy way to construct programs with given cache behaviour. Several simulation algorithms are given in this section. We also investigate what happens with the depth cache under different permutations of cache sizes.

Definition 3. *The* least recently used *(LRU) caching algorithm keeps as many pages in the cache as possible and if a page has to be evicted it evicts the page that has been accessed least recently.*

It is common in the previous papers to first prove that we can use LRU instead of the optimal caching interchangeably and the number of cache misses will be asymptotically the same (when we give LRU more space and/or time). There is one slight assumption that is often omitted and that is the regularity of the algorithm.

Definition 4 (Prokop [1999])**.** *Let A be an algorithm and we denote $A(n; M, B)$ the number of misses caused on a problem of size n with an optimal cache of size M with pages of size B. Then A satisfies the* regularity condition *if*

$$A(n; M, B) = \mathcal{O}(A(n; 2M, B)) \text{ for each } n, M, B \in \mathbb{N}.$$

The inequality in the definition is natural as when we increase the cache size the algorithm will cause less cache misses. The regularity condition simply states that the decrease will not be enormous. Prokop [1999] originally used this assumption to state the fact that an algorithm causes asymptotically the same number of cache misses on an optimal cache and on a (twice as large) LRU cache. We use this assumption again in Chapter 4. All natural and studied algorithms satisfy this regularity property. On the other hand in Section 4.3 we use the depth cache model of Section 3.4 to create algorithms with an almost arbitrary decrease of the number of cache misses.

The basic trick is to prove competitiveness of LRU and then combine it with the definition of regularity to get that the number of cache misses caused by a program satisfying the regularity condition will be asymptotically the same on an optimal cache as on an LRU cache with some space and/or time advantage.

3.1 Sync Model

We assume a two level memory model where the cache size may vary over time. Let us number pages of the main memory (e.g. hard-disk pages) by numbers $[N] = \{1, 2, \ldots, N\}$, so the sequence of requests is a sequence $a_1, a_2, \ldots, a_n$ of numbers in $[N]$.

The caching is done not by our program, but automatically. We show that we can think of the page evictions as done optimally or almost optimally, similarly to the constant cache model.

We allow the cache size to change, even many times, between two consecutive page requests. Therefore we need to know the minimal cache size between the two requests. We denote m_i the minimal cache size after the ith request and before the $(i+1)$-st request.

Under these assumptions both the optimal cache algorithm and LRU have synchronized cache sizes – it cannot happen that one sequence lasts longer on an LRU cache than on the optimal one and thus the LRU cache would have much smaller cache size. Therefore we call this model the *Sync Model*.

3.1.1 Belady is Still Optimal in the Sync Model

The optimal off-line algorithm proposed by Belady [1966] remains optimal even in our general model. Peserico [2013] and Bender et al. [2014] also prove this result in their models.

It is a useful convention that the optimal algorithm uses all available space even if it could delete some page that will not be seen in the near future. Leaving a page in a cache longer instead of keeping an unused empty space obviously does not increase the number of misses.

3.1.2 LRU Stays Competitive in the Sync Model

The least recently used (LRU) algorithm is a well known paging algorithm, that is on-line, i.e., it gets pages one by one without knowing anything about the future. LRU keeps as much pages in its cache as possible. When there is no more free memory left it deletes the least recently used page.

It is convenient for us to number cache sizes by the same indices as the requests. The cache size could change slower, or even faster than every request. Our only assumption is that the cache size does not change during a page request. If the cache size increases or decreases many times between two page requests a_i, a_{i+1} we are interested only in the minimal cache size m_i.

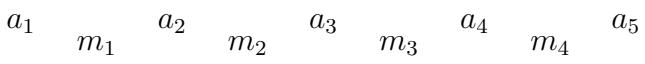

Figure 3.1: Sequence of requests with cache sizes.

For our purposes we would like to be able to use the least recently used algorithm and the optimal algorithm interchangeably. To be able to do this we need to know that LRU is about the same competitive even when the cache sizes change.

Let us denote by a superscript L the size of the LRU cache, and by superscript O the size of the optimal cache. The upcoming theorem states the competitive ratio is around $\frac{m^L}{m^L - m^O}$, thus we want to keep

$$\begin{aligned}
\frac{m_1^L}{m_1^L - m_1^O} &= \frac{m_2^L}{m_2^L - m_2^O} \\
m_1^L(m_2^L - m_2^O) &= m_2^L(m_1^L - m_1^O) \\
m_1^L m_2^L - m_1^L m_2^O &= m_2^L m_1^L - m_2^L m_1^O \\
-m_1^L m_2^O &= -m_2^L m_1^O \\
\frac{m_1^L}{m_2^L} &= \frac{m_1^0}{m_2^0}
\end{aligned}$$

We assume we are changing the cache size of LRU and of the optimal algorithm simultaneously by the same ratio. Another way to state this is that the optimal cache has size $m_i^O = \lfloor m_i^L / c \rfloor$ for some constant c at least two.

Theorem 1. *Let $m_1^L, m_2^L, \ldots, m_{n-1}^L$ be minimal sizes of the LRU cache. Let $m_1^O, m_2^O, \ldots, m_{n-1}^O$ be minimal cache sizes of the optimal algorithm. We assume that m_i^L / m_i^O remains constant for all i. The LRU algorithm makes no more than*

$$v \frac{m_1^L}{m_1^L - m_1^O}$$

misses where v is the number of misses of the optimal algorithm. We assume that both caches are empty at the beginning.

The idea of the proof is the same as in the classical proof used when the cache size is constant. We divide the requests to phases in such a way that during a phase LRU never evicts a page that has appeared in this phase. We then show that the number of cache misses caused by LRU can be upper-bounded by the number of different pages appearing in this phase. On the other hand the optimal cache is smaller and thus even if it would keep some pages cached from the last phase it would still make at least some cache misses (if the LRU cache is twice as large then the optimal cache causes at least half the number of cache misses of LRU).

Proof. We follow the idea of the proof described by Trevisan [2011]. To simplify reading we divide the proof into several lemmas.

We divide the sequence of page requests $a_1, a_2, \cdots, a_n$ into *phases*. Intuitively we may imagine that the LRU cache is empty at the start of each phase and the phase ends one request before the first deletion of a page by the LRU algorithm. The phases are a pure artificial construct which helps us to analyse the number of misses and the LRU does not know about phases and it does not delete all pages from cache at any moment.

More formally let $a_1, a_2, \ldots, a_n$ be the sequence of requests. First phase starts at 1 and ends at t for the smallest t such that $m_t^L < |\{a_i \mid i = 1, \ldots, t+1\}|$, or at n if no such t exists. The second phase is the first phase of the sequence $a_{t+1}, a_{t+2}, \ldots, a_n$ and we define following phases recursively.

Lemma 2. *Let $1 \leq t = T_1 < \ldots < T_p = n$ be the ends of phases and let k_i be the number of different pages in the i-th phase. LRU causes no more than k_i misses during the phase i. Moreover $k_i - 1 \leq m^L_{T_i - 1}$ and $k_i > m^L_{T_i}$.*

The optimal algorithm causes at least $\max(0, k_i - m^O_{T_{i-1}} - 1)$ misses.

Proof. By the definition of a phase no page causes more than one miss to the LRU algorithm thus LRU makes at most k_i cache misses during the phase i. The inequality $k_i - 1 \leq m^L_{T_i - 1}$ holds because there can be only one additional page requested and the phase did not ended earlier. The inequality $k_i > m^L_{T_i}$ again follows from the definition of a phase.

On the other hand the optimal algorithm sees at least k_i pages during the phase i and it had at most $m^O_{T_{i-1}} + 1$ pages cached from the previous phase (it has at most $m^O_{T_{i-1}}$ pages in the cache before the last page request of the $i-1$-st phase). Thus the optimal algorithm causes at least $\max(0, k_i - m^O_{T_{i-1}} - 1)$ misses. □

It is convenient for us to bound the number of misses of the optimal algorithm by $k_i - m^O_{T_i} - 1$ instead of $\max(0, k_i - m^O_{T_{i-1}} - 1)$.

Lemma 3. *Let T_i and k_i be as in Lemma 2. The optimal caching algorithm incurs at least*

$$\sum_i \max(0, k_i - m^O_{T_{i-1}} - 1) \geq \sum_i k_i - m^O_{T_i} - 1$$

cache misses.

Proof. We can see that at the beginning even the optimal algorithm makes k_1 misses instead of $k_1 - m^O_{T_1} - 1$ we are hoping for and thus we get $m^O_{T_1}$ extra misses.

During the second phase the optimal algorithm causes at least $\max(0, k_2 - m^O_{T_1} - 1)$ misses by the previous lemma. We take our $m^O_{T_1}$ extra misses from the first phase, and add those to the misses from the second phase. If $0 \geq k_2 - m^O_{T_1}$ we have $m^O_{T_1} \geq k_2$, and thus we can count that the optimal algorithm did at least k_2 misses. If $k_2 > m_{T_1^O}$ we have at least $k_2 - m^O_{T_1} + m^O_{T_1} - 1 = k_2 - 1$ misses. Again we analyse that the optimal algorithm did at least $k_2 - m^O_{T_2} - 1$ and $m^O_{T_2}$ extra ones are kept for the next phase.

We continue in this fashion and at each phase the optimal algorithm does at least $k_i - m^O_{T_i} - 1$ misses with $m^O_{T_i}$ extra ones for the next phase. The claim thus follows. □

On the other hand LRU causes at most k_i misses during each phase. Thus for each k_i misses of LRU the optimal algorithm causes at least $k_i - m_{T_i^O} - 1$ cache misses. But we already know that $k_i > m^L_{T_i} \geq m^O_{T_i}$. Thus for each k_i misses of LRU the optimal caching algorithm causes at least $m^L_{T_i} - m^O_{T_i}$ cache misses. □

3.2 Exact Time Model

In this section we give a common generalization of models of Bender et al. [2014] and Peserico [2013]. We consider a model where the cache size changes at exact times but cache misses and cache hits take different positive times.

A *memory profile* m is a function denoting the size of a cache in a given time. This notation is more common in papers of Bender et al. [2014] and Soorchaei [2015]. We are also using similar techniques and proving analogues to their theorems concerning LRU competitiveness.

Definition 5 (Exact Time Model)**.** *In the* Exact Time Model *we assume that the memory size changes occur at given times denoted by natural numbers. Each cache hit lasts for exactly one unit of time and each cache miss lasts for M units of time.*

This is a generalization as by setting $M = 1$ we get the model of Peserico [2013] and in the limit $M \to \infty$ we get the model of Bender et al. [2014]. Our proof gives slightly worse constants then the previous ones. We use time augmentation this means that we consider the LRU cache misses to last a shorter amount of time than the OPT cache misses. We also assume that the cache is always large enough to contain at least M different pages. So our result is not directly comparable to the result of Peserico [2013] or Bender et al. [2014]. Soorchaei [2015] shows that this time augmentation is necessary up to a constant multiplicative factor in their model (this result is already observed in the paper by Bender et al. [2014]).

Definition 6 (Bender et al. [2014])**.** *A memory profile m' is c-memory augmented for a constant $c \in \mathbb{R}$ from a memory profile m if $m'(t) = cm(t)$.*

Definition 7 (Bender et al. [2014])**.** *A memory profile m' is c-time augmented for a constant $c \in \mathbb{R}$ from a memory profile m if $m'(t) = m(\lfloor t/c \rfloor)$.*

There is no way for a paging algorithm to utilize a large and fast cache size increase as it can load at most one page each M time units. On the other hand it is common for a program to wait for user input, network connection or to be rescheduled by the operating system. All these could cause the program to be inactive for a very long time thus giving another running programs enough time to utilize the cache and thus decrease our program's cache size. We summarize this paragraph in the observation below.

Observation 4. *We may assume without loss of generality that the cache increases its size by at most one each M time units. But arbitrarily large size decreases are still possible.*

3.2.1 LRU is competitive in the Exact Time Model

The main result of this section is Theorem 7. Its proof will combine approaches both from Theorem 1 and its counterpart given by Bender et al. [2014]. Moreover we begin by the definition of inner square profiles used by the latter. Recall that a square profile (Definition 2) is a memory profile that is piecewise constant and the time can be cut to form squares – the cache size is constant and its size is the length of the time interval. This definition is not crucial for the proof but provides a handy tool to make the presentation more clear.

Definition 8 (Bender et al. [2014])**.** *Let m be a memory profile. We define an* inner square profile *of m denoted by m' to be the square profile (see Definition 2) with inner square boundaries $t_0 = 0$ and t_{i+1} is defined inductively as the largest natural number such that $t_{i+1} - t_i \leq m(t)$ for each time $t \in [t_i, t_{i+1})$. Recall that then $m'(t) = t_{i+1} - t_i$ for each $t \in [t_i, t_{i+1})$. See Figure 3.2.*

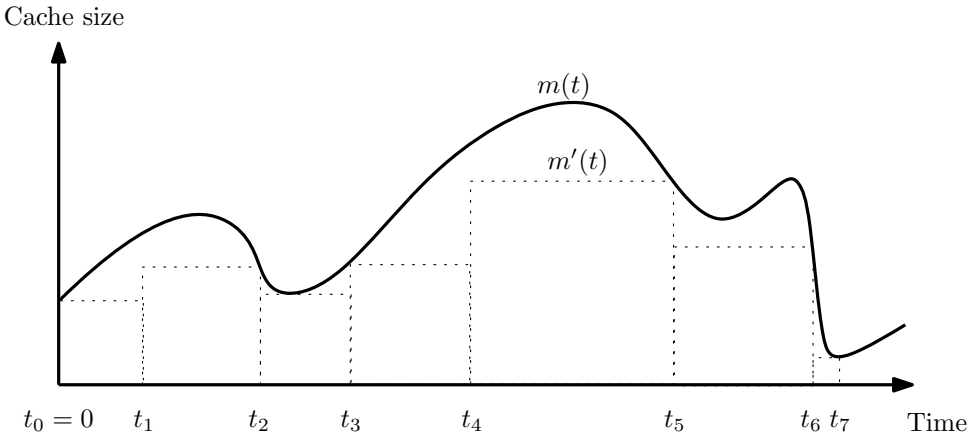

Figure 3.2: A memory profile m with inner squares and the inner square memory profile m' which is the upper envelope of the dotted squares.

We restate a variant of Lemma 3.1 by Bender et al. [2014]. This seems similar but the setting of the Exact Time Model is different. The proof of both the following lemma and theorem is also inspired by the aforementioned paper.

Lemma 5. *Let m denote a memory profile that does not increase its size too quickly, that is $\forall i < j \in \mathbb{N}, j - i \leq M$ we know that $m(j) \leq m(i) + 1$ holds where M is the time duration of a cache miss. Let $t_0 < t_1 < \ldots$ be the inner square boundaries of the square memory profile m' which is the inner square profile of m. Under these assumptions the following inequalities hold:*

1. $\forall t \in \mathbb{N}\colon m'(t) \leq m(t)$,
2. $\forall i \in \mathbb{N}\colon t_{i+2} - t_{i+1} \leq \lceil (1 + \frac{1}{M})(t_{i+1} - t_i) \rceil \leq 2(t_{i+1} - t_i)$,
3. $\forall i, \forall t \in [t_{i+1}, t_{i+2})\colon m(t) \leq (t_{i+1} - t_i) + \lceil (3(t_{i+1} - t_i))/M \rceil \leq 4(t_{i+1} - t_i)$.

Proof. 1. Follows directly from the definition of an inner square profile which is always at most the size of the original profile m.

2. For each square of the inner square profile m' we have a witness of its size, in other words $\forall i \in \mathbb{N}\ \exists t_i^* \in [t_i, t_{i+1})\colon m(t_i^*) = m'(t_i) = t_{i+1} - t_i$.

   Our memory profiles satisfy the property that they are not increasing too much, thus we conclude that

$$\begin{aligned} m(t_{i+1}) &\leq m(t_i^*) + \lceil (t_{i+1} - t_i^*)/M \rceil \\ &= (t_{i+1} - t_i) + \lceil (t_{i+1} - t_i^*)/M \rceil \\ &\leq \lceil (1 + 1/M)(t_{i+1} - t_i) \rceil . \end{aligned}$$

   By the very definition of an inner square profile we have $t_{i+2} - t_{i+1} \leq m(t_{i+1})$ as this is one of the sizes that have to be higher than the cache size in the given square.

   Combining results from the previous two paragraphs we get

$$t_{i+2} - t_{i+1} \leq m(t_{i+1}) \leq \lceil (1 + 1/M)(t_{i+1} - t_i) \rceil .$$

3. Again by the same argument we have a bounding time t_i^* that is equal to the cache size during the time interval i (that is during $[t_i, t_{i+1})$). We thus have $\forall t \in [t_{i+1}, t_{i+2})\colon$

$$\begin{aligned} m(t) &\leq m(t_i^*) + \left\lceil \frac{t - t_i^*}{M} \right\rceil \\ &\leq (t_{i+1} - t_i) + \left\lceil \frac{t_{i+2} - t_i}{M} \right\rceil \\ &= (t_{i+1} - t_i) + \left\lceil \frac{(t_{i+2} - t_{i+1}) + (t_{i+1} - t_i)}{M} \right\rceil \\ &\leq (t_{i+1} - t_i) + \left\lceil \frac{\lceil (1 + 1/M)(t_{i+1} - t_i) \rceil + (t_{i+1} - t_i)}{M} \right\rceil \\ &\leq (t_{i+1} - t_i) + \left\lceil \frac{2(t_{i+1} - t_i) + (t_{i+1} - t_i)}{M} \right\rceil \\ &= (t_{i+1} - t_i) + \left\lceil \frac{3(t_{i+1} - t_i)}{M} \right\rceil \end{aligned}$$

□

Lemma 6. *Let us denote the time when the LRU cache finishes processing of a page access sequence $\sigma = p_1, p_2, p_4, \ldots, p_{|\sigma|}$ on a memory profile m by $C_{LRU}(m, \sigma)$, similarly we define $C_{OPT}(m, \sigma)$ to be the time when the optimal caching algorithm finishes processing of σ. All time is counted in the Exact Time Model where each cache miss of the OPT algorithm lasts M units of time and a cache hit lasts 1 unit of time. Consider any fixed $X \geq 32$. A cache miss of the LRU algorithm lasts M units of time but each cache hit lasts $8X$ units of time. Where we assume that $M \geq X$.*

Let m' be the inner square memory profile of m and let $m'_{X,8X}$ stand for the X-size and $8X$-time augmented inner square profile. We assume that $m(t) \geq M$ for each time t. Without loss of generality we may assume that $m(0) = M$. Then

$$C_{LRU}(m'_{X,8X}, \sigma) \leq 8X C_{OPT}(m, \sigma) + 3M.$$

The fact that the profile of the LRU cache is $8X$-time augmented is compensated by the fact that each cache hit lasts $8X$ units of time. This way the progress in the non-augmented time is the same as the one of the OPT algorithm. On the other hand each cache miss of LRU lasts $M/8X$ time instead of M units of time. This implies that not only LRU finishes earlier than OPT but moreover it also makes at most X times more cache misses.

The basic idea of the proof is to inductively prove that LRU finishes a square earlier than the optimal caching algorithm OPT. We amortize each page access that causes a cache miss to the LRU cache and not to the OPT cache. We consider two cases – when the page was evicted by a cache size decrease and when it was not.

It is clearly not vital to have different time and space augmentation. The LRU algorithm with a larger cache size causes at most as many cache misses as with a smaller cache size as discussed in Section 3.3. On the other hand this discrepancy of augmentations greatly simplifies our arguments.

The additive factor $3M$ is used in the base case. We may as well assume that LRU may load first three pages for free. Consider the case when $m(0) = M$ and $m(M) = 2M$, thus the first two squares have sizes M and $2M$. If the optimal algorithm would case a single cache miss and then accessed the same page for $2M - 1$ times it would finish during the second square. On the other hand LRU would not finish during the first square.

Observe that the memory profile depicted by Figure 3.2 does not satisfy assumptions of Lemma 6 as either the size $m(0) > M$ or there is a time t such that $m(t) < M$. It illustrates a general memory profile.

We do not write floor functions in the proof when we are dividing M/X for the sake of clarity. We could always make X twice as big and we assume $M \geq X$. This is the reason we have chosen X with a large reserve.

Proof. Let us slightly abuse the notation by using OPT_j as the set of pages present in the OPT cache in time j. In the same way we use LRU_j.

We show by induction on the number of squares that for all page access sequences σ if the optimal completion time $C_{OPT}(m, \sigma) \in [t_{i+1}, t_{i+2})$ then

$$C_{LRU}(m'_{X,8X}, \sigma) \leq 8Xt_{i+1} + 3M.$$

This induction assumption is made possible by giving the LRU algorithm two advantages over the optimal one. One is the memory augmentation and the second one is the time augmentation. Memory augmentation helps us when there are no cache size decreases because having a larger cache allows for an argument similar to the one used in constant size LRU analysis. The other advantage is the time augmentation which allows us to assume that the LRU algorithm always finishes with a great enough advantage ahead of the optimal one.

Base case $i = 0$:

Let us suppose that the optimal cache finishes before time t_2. We know that $t_2 \leq 3M$ by Lemma 5. Then LRU can load all pages that OPT does and finish earlier.

Inductive case:

All we need to do is to amortize the number of page accesses that cause a cache miss to LRU but not to the optimal cache. Let us consider a page request to a page address p that has caused a cache miss only to the LRU cache. We distinguish two cases why the page address p was evicted: because of a page size decrease and when it was evicted because of another page request.

- Let us consider the case when the page address p was evicted because the square size dropped. We have already observed that instead of keeping unused space in the cache the optimal caching algorithm may keep pages that are not going to be requested in its cache until another cache miss occurs. Thus if LRU evicts k pages because of a size decrease we may assume that the optimal algorithm evicts at least $\ell := \lceil k/X \rceil$ pages.

  All of these ℓ pages caused a cache miss to the optimal cache at some point as we assume both the LRU and the optimal cache are empty at the beginning. When these ℓ pages caused a cache miss to OPT it took

  $$\ell M = \lceil k/X \rceil M$$

  units of time.

  On the other hand the LRU cache was able to load all of them in time

  $$\frac{\ell M}{8X} = \frac{\lceil k/X \rceil M}{8X}$$

  and it takes

  $$\frac{kM}{8X}$$

  time to load the k evicted pages to LRU again.

  Thus we need that

  $$\frac{kM}{8X} + \frac{\lceil k/X \rceil M}{8X} \leq \lceil k/X \rceil M$$

  which trivially holds. The argument would go through with time augmentation $4X$ but we use the similar argument in the next section thus we double the time augmentation to be able to use it twice.

- The situation when the page address p was evicted because another page was loaded into the LRU cache also needs to be analyzed in two cases. One of these cases – when the size of LRU did not decreased too much since the last access to p will use an argument similar to the one used in the classical proof of LRU competitiveness. Other case will be similar to the previous one and will use the observation that if enough pages were evicted because of a cache size decrease then we may load them again.

  First of all we need to lower bound the number of pages present in the LRU cache. We know that the page address p was already accessed due to our assumption that it is already present in the optimal cache. Let j denote the number of the page request when the page address p was accessed for the last time (that means $\sigma_j = p$). If it would be the case that

  $$C_{OPT}(m, \sigma_1, \sigma_2, \ldots, \sigma_j) \in [t_{i+1}, t_{i+2})$$

and

$$C_{LRU}(m, \sigma_1, \sigma_2, \ldots, \sigma_j) \in [t_i, t_{i+1})$$

we have that the size of the LRU cache is at least $X(t_{i+2} - t_{i+1})/2$ by Lemma 5. If this would not be the case we know that LRU processed the prefix of length j of the access sequence earlier and it has a time advantage. We could use this time advantage to load the page address p and finish on time. Thus let us in the following suppose that the size of the LRU cache during the request j was at least $X(t_{i+2} - t_{i+1})/2$.

We distinguish two cases – if the LRU cache decreases its size by at least $(t_{i+2} - t_{i+1})/4$ in total between the time it processes the request σ_j and now or not.

- The LRU cache does not evict more than $X(t_{i+2} - t_{i+1})/4$ pages because of cache size decreases. This means that it had size at least

  $$X(t_{i+2} - t_{i+1})/4$$

  all this time. The page address p was on top of the LRU stack when it caused a cache miss to it the last time. Since the time the sequence had to access at least $X(t_{i+2} - t_{i+1})/4$ different pages to move the page address p down enough in the stack.

  Let us consider the last $X(t_{i+2} - t_{i+1})/4$ different page addresses. LRU could keep them all in its cache so it caused at most $X(t_{i+2} - t_{i+1})/4$ cache misses and it took it at most

  $$MX(t_{i+2} - t_{i+1})/(4 \cdot 8X) = (t_{i+2} - t_{i+1})M/32$$

  time units. On the other hand the OPT cache had at most $(t_{i+2} - t_{i+1})$ pages cached at the beginning of this address sequence so it took it at least

  $$(X/4 - 1)(t_{i+2} - t_{i+1})M$$

  time units.

  LRU has therefore enough time to even load all the $X(t_{i+2} - t_{i+1})/4$ page addresses again.
- If at least $k = X(t_{i+2} - t_{i+1})/4$ page addresses were evicted from the LRU cache because of size decreases then we can use a similar computation as in the case when p when p was evicted because of a size decrease. We thus get that LRU may even load $X(t_{i+2} - t_{i+1})/2$ page addresses back.

□

Theorem 7. *Under the assumptions of Lemma 6 the LRU algorithm provided with X-memory and $8X$-time augmented cache always completes sooner than the optimal cache algorithm.*

This theorem also gives us that not only the LRU finishes earlier but due to the prolonged time of cache hit it is also competitive in the number of cache misses.

Proof. By Lemma 5 the augmented inner square profile has size at least as big as the original one so it also makes no more misses then the original augmented profile of LRU cache (for clarification see Section 3.3). Together with Lemma 6 the theorem follows. □

3.3 Algorithms with Linearly Ordered Pages

The most basic property that we would like our caching algorithm to satisfy is that the cache behaves at least as good as if it had its minimal size all the time. In other words this is a class of cache algorithms without a generalization of Belady anomaly described by Belady et al. [1969].

We present a class of cache algorithms that satisfy this condition. The LRU algorithm is a special case of such an algorithm and thus it behaves at least as good as if it had minimal size all the time. On the other hand Peserico [2013] gives an example of a cache algorithm with an optimal competitive ratio, but that is not competitive when the cache capacity changes over time.

Observation 8. *Let the cache size be never smaller than some constant m and let us assume there is a total ordering of pages, such that at most one page changes its place in this total order during a page request. Moreover we assume that the caching algorithm does not depend on the cache size. If the cache keeps just the first m_i pages of the linear order, then the cache behaves at least as good as if it had the minimal size m all the time.*

The ordering is a very natural notion of saying "this page will be swapped before another."

Proof. We can see that one request causes at most one access to main memory. Let us assume we do the same sequence of requests on a cache C with constant size m and on a cache D with variable size, at each moment at least m large (its size $s(t) \geq m$ for each time t), and that both have the same algorithm and thus the same linear ordering of pages all the time. The content of the cache with variable size D is always a superset of the content of cache C. Thus all misses of the cache D are also misses of the cache C. □

3.4 Depth Model

Here we introduce a new cache model, where pages are not indexed and accessed using their addresses but instead the program indexes using the depth of the page in an infinite LRU cache. We show that these models are interchangeable when the indexing function is computable on processor. Let us note that our simulation uses more processor time than the original programs.

This model also allows us to construct programs with an arbitrary (given by a function computable on processor) cache behaviour. Moreover this is a useful tool how to think about the changing cache size setting. We also show that the models have the same computational power, although the number of cache misses will differ. Similarly as the complexity theory studies time constructible functions we thus show that every function computable on processor is LRU cache constructible.

3.4.1 Definition of the Depth Model

In the standard RAM programs we address memory by page numbers (thus when we need to we call this model the *addressing cache model*). In the *depth model* we have a *depth-cache* which is a stack of potentially infinite size which is empty in the beginning. Pages are identified by their depth in this cache (with LRU strategy). When a page is requested it is taken from its place in the stack and pushed on the top of the stack. If there are not enough pages in the cache a new empty page is pushed on the top. The depth of the page that is on the top of the depth-cache is defined to be 1.

The depth-cache makes a cache miss if the accessed page has depth greater than the cache size or when we access a page depth greater than the number of pages present in the cache we also make a cache miss.

Observation 9. *Let us consider two caches of the same constant size, one of them is addressed by depths and the other is addressed by page addresses and uses the LRU algorithm. When we access the same pages the Depth Model cache makes a cache miss if and only if the usual addressing model makes a cache miss.*

Proof. The observation follows from the definition of depth-cache and the definition of a cache miss in the depth cache model. □

It is convenient for us to distinguish accessing using depth and using address. We denote $f_a(i)$ the i-th address accessed by a program. Similarly we denote $f_d(i)$ the depth in the depth-cache of the i-th address accessed by the same program. That means although values $f_a(i)$ and $f_d(i)$ might differ, the page is the same.

In the next three sections we are dealing just with functions that are *computable on the processor*. That means we are able to compute them using just constants in the processor cache and registers. Similarly we could deal with functions that are computable using only accesses to a lower level cache and cause no cache misses on the higher level.

Definition 9. *We say that a function $f(n_1, \ldots, n_{arity(f)})$ is* computable on processor *if it can be enumerated using only constantly many variables.*

Remarks

The depth model might also be used to simulate multiple levels of LRU cache as follows. The cache could have several sizes corresponding to the sizes of the multiple levels of LRU caches and when the algorithm accesses a page in a depth d all caches of size at most $d-1$ cause a cache miss and especially if the depth d is greater than the number of present pages all caches cause a cache miss.

The depth model might also be generalized even more to allow arbitrary penalization for accessing deep pages. We could, for instance, model linked lists by a very similar model where we cause d misses by accessing a page of depth d which corresponds to the d steps needed when accessing the d-th element of a linked list and moving it to the front.

Given that this model is so natural it is surprising that we have not found any similar one in the literature.

3.4.2 Simulation of the Depth Model with Constant Space Addressing Using the Page Addresses

Theorem 10. *Given a function $f_d\colon \mathbb{N} \to \mathbb{N}$ that is computable on the processor we can compute each $f_a(i)$ without making additional cache misses.*

The idea is to track the position of a single page in the depth-cache and see which page is in the right depth.

Proof. For every time i we can compute how many pages are there in the depth-cache. This is easily done using just processor registers (in constant space) by going over all times $j = 0$ to i and counting how many times we access a page that is deeper than the current number of pages present in the cache (note that this could be greater than its size, as its size only tells us when the cache will make a cache miss). Note that this procedure uses only two variables and several computations of $f_d(j)$, thus can be done without causing a cache miss.

Let n_i denote the number of pages present in the stack of the depth-cache in time i. If $f_d(i) > n_i$ we can set $f_a(i) = n_i + 1$. By induction this page was not used before by the Addressing Cache.

Algorithm 1: *ni*: Returns the number of pages present in the cache in time *time*.

```
if time = 0 then
    return 0
end
presentpages = 0
for 0 ≤ j < time do
    if f_d(j) > presentpages then
        presentpages ← presentpages + 1
    end
end
return presentpages
```

Note that we could even compute n_i continuously during the algorithm (instead of computing it over and over again).

If $f_d(i) \leq n_i$ the program would like to access a previously accessed page and we need to find its address. All we need to do is to decide if a page j is in depth $f_d(i)$ in time i for every accessed page. We show that this can also be done just using constantly many variables and using several computations of $f_d(i)$.

For a page of address j (such that $1 \leq j \leq n_i$) we know that the first access is in time t such that $n_t = j - 1$ and $f_d(t) > n_t$. In time $t + 1$ the page will be at the top of the depth-cache. Now the only thing we have to maintain is its depth $d_j(k)$. We know that its depth will increase in time k if $f_d(k) > d_j(k)$ (we access a page that is deeper than our page of address j), its depth will stay the same if $f_d(k) < d_j(k)$ (we access a page that is on a lower position in the cache), and if $f_d(k) = d_j(k)$ then $d_j(k + 1) = 1$ (as we accessed the page of address j). This can be done in constant space. After we find a page such that $d_j(i) = f_d(i)$ we simply set $f_a(i) = j$.

$$d_j(k+1) = \begin{cases} d_j(k) + 1 & \text{if } f_d(k) > d_j(k), \\ d_j(k) & \text{if } f_d(k) < d_j(k), \\ 1 & \text{if } f_d(k) = d_j(k). \end{cases}$$

Algorithm 2: *firstAccTime*: Computes the first access time of the page address *page* before time *time*.

```
presentpages ← 0
for 0 ≤ j ≤ time do
    if f_d(j) > presentpages then
        presentpages ← presentpages + 1
    end
    if presentpages ≥ page then
        return j + 1
    end
end
return -1
```

Algorithm 3: *getDepth*: Computing the depth of the page address *page* at time *time*.

```
depth ← 1
fat ← firstAccTime(page, time)
if fat < 0 then
    return -1
end
for fat ≤ j < time do
    fdj ← f_d(j)
    if fdj = depth then
        depth ← 1
    end
    else if fdj > depth then
        depth ← depth + 1
    end
end
return depth
```

Algorithm 4: f_a: Computing $f_a(i)$ for given time i.

```
presentpages ← ni(i)
fdi ← f_d(i)
if fdi > presentpages then
    return presentpages + 1
end
for 1 ≤ j ≤ presentpages do
    if getDepth(j, i) = fdi then
        return j
    end
end
```

□

3.4.3 Simulation of the Address Model with Constant Space Addressing Using the Depth Model

Theorem 11. *Given a function $f_a\colon \mathbb{N} \to \mathbb{N}$ that is computable on the processor we can compute each $f_d(i)$ without making additional cache misses.*

The idea is to count the number of pages that are accessed after the last access to the given page. This enables us to compute its depth in the depth-cache.

Proof. We can determine if a page address was accessed at some time before a given time. Also we can compute the maximal accessed address. All this can be done on processor by going through all accesses. Using these primitives we can compute how many pages are there present in the depth-cache in a given time.

Algorithm 5: *maxaddr*: Returns the maximum of used page addresses until *time*.

```
maxa ← 0
for 0 ≤ i < time do
    if f_a(i) > maxa then
        maxa ← f_a(i)
    end
end
return maxa
```

Algorithm 6: *addrhasbeenaccessed*: Determines if a given address *addr* was accessed before time *time*.

```
for 1 ≤ k < time do
    if f_a(k) = addr then
        return True
    end
end
return False
```

Algorithm 7: *ni*: Determines how many pages are there present in time *time*.

```
maxa ← maxaddr(time)
pagespresent ← 0
for 1 ≤ j < maxa do
    if addrhasbeenaccessed(j, time) then
        pagespresent ← pagespresent + 1
    end
end
```

For a given page address j we can compute the time of first access in constant space. We just have to find the minimum time t such that $f_a(t) = j$. Thus if the time i is the first time the page $f_a(i)$ is accessed, we set $f_d(i) = n_i + 1$ where n_i is the number of pages present in the depth-cache in time i.

If the page $f_a(i)$ has already been accessed, we have to compute its depth. For every two pages we can keep track which one has been accessed the last. Thus we can compute how many pages has been accessed after the last access of page $f_a(i)$ which gives us the depth of page $f_a(i)$.

Algorithm 8: *accessedlater*: Returns True if *page*2 was accessed after *page*1 (before time time) and False if not. If *page*1 was not visited returns True if *page*2 was visited and False otherwise.

```
p1vis ← False
p2vis ← False
p1 ← True
for 0 ≤ i < time do
    if f_a(i) = page1 then
        p1vis ← True
        p1 ← True
    end
    else if f_a(i) = page2 then
        p2vis ← True
        p1 ← False
    end
end
if ¬ p1vis then
    return p2vis
end
else
    return ¬ p1
end
```

Algorithm 9: *getDepth*: Returns the depth of page address *page* at time *time*.

```
depth ← 1
maxa ← maxaddr(time)
for 1 ≤ i ≤ maxa do
    if accessedlater(page, i, time) then
        depth ← depth + 1
    end
end
return depth
```

Algorithm 10: f_d: Computes $f_d(i)$ using f_a.

```
fai ← f_a(i)
if addrhasbeenaccessed(fai, i) then
    return getdepth(fai, i)
end
else
    return ni(i) + 1
end
```

□

3.4.4 Programs with Cache Behaviour Given by a Constant Space Computable Function

Loosely speaking in this section we prove that every cache behaviour given by a function computable on processor can be realized by an algorithm that does not need to consider the cache size. We require just small and clearly necessary conditions on both functions m and a in the following theorem. The function a gives the number of page accesses that should be done and the function m determines the number of cache misses on this many page accesses and a depth cache of a given size. The only assumption that might be theoretically weakened is the fact that both m and a are computable on processor. On the other hand this is a very rich class of functions describing asymptotic behaviours.

Theorem 12. *Let us assume a function* $a\colon \mathbb{N} \to \mathbb{N}$ *and a function*

$$m\colon \mathbb{N} \times \mathbb{N}^+ \to \mathbb{N}^+$$

such that

$$\forall n, s \in \mathbb{N}\colon m(n,s) \leq a(n)$$

and

$$\forall n, s \in \mathbb{N}\colon m(n,s) \geq m(n,s+1),$$

both of them computable on processor. Moreover we assume that

$$m(n,s) = m(n,n) \text{ for each } s \geq m(n,n).$$

We make a cache-oblivious algorithm A that takes input n and makes exactly $a(n)$ memory accesses and exactly $m(n,s)$ cache misses on an LRU cache of size s (without knowing the size s).

Proof. By Theorem 10 it suffices to construct a function $f_d(i)$ that is addressing a depth-cache and is also computable on processor.

We may suppose that $m(n,n)$ is the number of pages ever accessed and $m(n,1) = a(n)$ as we want to make $a(n)$ memory accesses and accessing the page at depth 1 more times gives us nothing.

First of all we access all pages that will ever be accessed in the first for loop. Then we make sure that at each cache size s we make $m(n,s) - m(n,s+1)$ additional cache misses than with cache size $s+1$. This is done easily by accessing the page at the depth $s+1$.

Algorithm 11: $A(n)$: Makes $a(n)$ memory accesses and $m(n,s)$ misses on a depth-cache of size s.

```
for 1 ≤ d ≤ m(n,n) do
    Access the page at the depth d
end
for 1 ≤ s < m(n,n) do
    smisses ← m(n,s) − m(n,s+1)
    for 0 ≤ i < smisses do
        Access the page at the depth s+1
    end
end
```

□

This approach accesses depths in a mostly increasing order. Theorem 10 gives us a slightly stronger tool, namely we can make misses according to an arbitrary function f_d that is computable on processor.

3.4.5 Any Computation on the Address Model is Computable on the Depth Model

Here we give only a proof that the models are similarly powerful but the number of cache misses will be dramatically different.

Observation 13. *Anything that can be computed by an algorithm that uses depth addressing can be also computed by an algorithm that uses the usual addressing by page addresses.*

Proof. We can simulate the depth-cache using a single array and move pages as they are accessed. □

Observation 14. *Anything that can be computed by an algorithm that uses addressing by page addresses can also be computed by an algorithm that uses depth addressing.*

Proof. When the original algorithm accesses the page address $f_a(i)$ we just access the depth $f_d(i) = f_a(i)$ and then all the depths $f_a(i) - 1, f_a(i) - 2, \ldots, 3, 2, 1$ in decreasing order. This leaves the LRU depth-cache in such a state that depths correspond to page addresses. □

It would be interesting to decide if these kinds of simulations can be made in such a way that they cause the same number of cache misses as their counterparts.

3.4.6 Changing Cache Size

In this section we investigate what happens to the depth model when we permute the cache sizes in time. This is different than in the classical addressing model because not all permutations there are valid. In the classical model we may assume that the cache size grows by at most one page each time, as we cannot cache more pages anyway. This does not hold for the depth-cache – we may assume arbitrary size changes because the cache is only interested in the accessed depth. In other words the depth-cache is not empty when we increase its size. This is the main advantage of the depth-cache model.

Another interesting feature of depth-caches is that provided that the cache has constant size we can permute the accessed depths and still get the same number of cache misses, although we cannot expect to access the same set of pages. This does not hold at all for an address cache – for instance take a program that reads each item in an array thousand times and then continues with others versus a program that thousand times scans the array.

We give a simple observation dealing with the case where no misses are caused by accessing an empty cache. This models the situation when the program is already running and all pages have already been accessed. Equivalently we could access all pages before the start of our program and cause at most twice many cache misses.

We can simulate all permutations of cache sizes either by permuting the actual sizes or by permuting the depth accesses. In the statement of the next theorem we choose the latter.

Definition 10 (Oxley [2006])**.** *A pair $M = (E, \mathcal{I})$ is called a* matroid *iff E is a finite set (called the* ground set*) and $\mathcal{I}$ (called the set of* independent sets*) is a family of subsets of E with the following properties (also known as independent sets properties):*

(I1) The empty set is independent $\emptyset \in \mathcal{I}$.

(I2) Every subset of an independent set is independent $S \in \mathcal{I}, T \subseteq S \Rightarrow T \in \mathcal{I}$.

(I3) For any two independent sets $A, B \in \mathcal{I}$ of different size – without loss of generality $|A| < |B|$ we have an element $e \in B \setminus A$ such that $\{e\} \cup A \in \mathcal{I}$.

Definition 11. *Let S_n stand for the set of all permutations over n elements*

$$S_n = \{\pi \colon [n] \to [n] \mid \pi \text{ is bijective}\}.$$

Definition 12. *For a fixed depth $d \in \mathbb{N}$, cache size function $s \colon \mathbb{N} \to \mathbb{N}$ and a finite set of times $T \subset \mathbb{N}$ we denote $Smaller_T(s, d) = \{i \in T \mid s(i) < d\}$ the set of times where the cache size is smaller than d.*

Theorem 15. *Let $s \colon \mathbb{N} \to \mathbb{N}$ be a function that determines the size of a depth-cache, that is the size of the depth-cache in time i is $s(i)$. Let $f_d \colon [n] \to \mathbb{N}$ be the sequence of page requests to the depth-cache. We denote $m(f_d, s)$ the number of cache misses caused by the sequence of n requests determined by f_d on a depth-cache of sizes $s(i)$.*

Let us suppose that all cache misses are caused by accessing a page with a depth greater than the cache size. In other words no cache miss is caused because of there are not enough pages present in the cache.

Let us denote M_w the maximal number of cache misses (worst case) over all permutations $M_w = \max_{\pi \in S_n} m(f_d, s \circ \pi)$. Let us denote M_b the minimal number of cache misses (best case) over all permutations $M_b = \min_{\pi \in S_n} m(f_d, s \circ \pi)$. There is an algorithm that computes both M_w and M_b and runs in polynomial time and polynomially many times calls f_d.

Let us denote M_a the average number of cache misses over all permutations $M_a = \mathbb{E}_{\pi \in S_n}[m(f_d, s \circ \pi)]$ where $(s \circ \pi)(i) = s(\pi(i))$. Then

$$M_a = \sum_{i=1}^{n} \frac{Smaller_{[n]}(s, f_d(i))}{n}$$

holds.

Proof. Let us prove the cases one by one.

(M_w) Let $M = (R, \mathcal{I})$ be a pair where R is the set of times of requests $R = [n]$ and $\mathcal{I}$ denotes the set of subsets of R such that for each $I \in \mathcal{I}$ all requests in I cause a cache miss in some permutation. In other words we have a

different time for each request such that the request would cause a cache miss at that time.

Formally when we have $I \in \mathcal{I}$ such that $I = \left\{i_1, i_2, \ldots, i_{|I|}\right\}$ then there is a set $C_I = \left\{c_1, c_2, \ldots, c_{|I|}\right\}$ such that $c_k \neq c_\ell$ for $k \neq \ell$ and accessing a page in depth $f_d(i_\ell)$ at the time c_ℓ causes a cache miss, that is $s(c_\ell) < f_d(i_\ell)$. We show that M forms a matroid.

Side note: Another formalization would be to say that there exists a permutation $\pi \in S_n$ such that $\pi(i_\ell) = c_\ell$ and we have $s(\pi(i_\ell)) < f_d(i_\ell)$. But this permutation would not be uniquely determined usually as the independent sets I have mostly cardinality smaller than n. We thus believe that the I and C_I formalization given in the previous paragraph and used in the rest of the proof is easier to read and comprehend.

(I1) As there are no requests in the empty set thus all of those cause a cache miss.

(I2) If we have a set of requests $I \in \mathcal{I}$ such that all of those can cause a cache miss (each in different time) the same holds for each subset of I.

(I3) Let us suppose that the sets $A, B \in \mathcal{I}$ are sorted by the accessed depth, in other words $A = \left\{a_1, a_2, \ldots, a_{|A|}\right\}, B = \left\{b_1, b_2, \ldots, b_{|B|}\right\}$ then for each $i < j$ we have $f_d(a_i) \geq f_d(a_j)$ and the same holds for the set B, thus $\forall i < j \colon f_d(b_i) \geq f_d(b_j)$. Additionally we may suppose that if $f_d(i) = f_d(j)$ and both $i, j \in A \cap B$ the ordering is the same on the intersection of A and B.

Without loss of generality we may assume that the times for $A \cap B$ are the same. Suppose we would have a request $r \in A \cap B$ such that $c_r \neq c'_r, c_r \in C_A, c'_r \in C_B$ (let us take the smallest such in B, that is with the greater $f_d(r)$). Now we may take the time c_r if $s(c_r) > s(c'_r)$ or c'_r if $s(c'_r) \geq s(c_r)$ and use it for the same request r in both C_A and C_B. Without loss of generality let $s(c_r) > s(c'_r)$, if c_r is used in C_B elsewhere as some c'_t we can replace it with c'_r and the inequality $f_d(t) > s(c'_t) = s(c_r) > s(c'_r)$ would also hold.

We explicitly reason that we cannot cycle in the previous argument. If it holds that $t > r$ everything is ok, we have reduced the number of differences on the beginning of the set $A \cap B$ by one and we may continue until all times in $A \cap B$ are the same. The case when $f_d(t) > f_d(r)$ gives us a contradiction either with the choice of r or with the fact that c_r is used twice in C_A.

Now we can take the time $e \in B \setminus A$ that is the smallest in the ordering given by the ordering of B (that is with the greatest $f_d(e)$). We accompany it with the time $c_e \in [n] \setminus C_A$ with the smallest cache size, that is $\min s(c_e)$. Observe that as the size of B is strictly greater than the size of A thus the size of C_B is also strictly larger than the size of C_A. Moreover the intersection of $A \cap B$ uses the same times so there is a time in $C_{B \setminus A}$ that is not used in C_A.

We observe that $s(c_e) < f_d(e)$ by definition of e and c_e.

For a given set $I \subseteq [n]$ we can decide in polynomial time if $I \in \mathcal{I}$ or not. This can be done greedily by sorting depths of page accesses in I and then assigning times with the greatest $s(i)$ such that the given page request still causes a cache miss.

It is a well known fact that we can use a greedy algorithm on matroids in polynomial time Oxley [2006]. We could get an even easier proof by using the matroid intersection theorem Oxley [2006] – given any two matroids $M_1 = (E, \mathcal{I}_1), M_2 = (E, \mathcal{I}_2)$ one can find a maximal set that is independent in both matroids

$$\max_{S \in \mathcal{I}_1 \cap \mathcal{I}_2} |S|.$$

We define two matroids on the ground set $[n] \times [n]$. One has independent sets consisting of pairs with different second parts ($((a,b),(c,d) \in I$ thus $c \neq d$) and the other matroid has the condition on independent sets that for each pair (a,b) we have a cache miss $f_d(a) > s(b)$.

(M_b) The proof is analogous to (M_w).

(M_a) By linearity of expectation we have

$$\begin{aligned} M_a &= \mathbb{E}_{\pi \in S_n}[m(f_d, s \circ \pi)] \\ &= \sum_{i=1}^{n} \Pr_{\pi \in S_n}[f_d(i) > s(\pi(i))] \\ &= \sum_{i=1}^{n} \frac{\text{Smaller}_{[n]}(s, f_d(i))}{n}. \end{aligned}$$

□

4. Competing Algorithms

In this chapter we investigate perhaps the most interesting case when there are two algorithms competing for one cache. We give a bound on the number of cache misses of two pseudo-parallel algorithms. Moreover we discuss two basic assumptions – the tall cache assumption and the regularity assumption used in our proof. These assumptions tell us whether the results of the first section are useful and generalizable to algorithms running in pseudo-parallel.

4.1 Two Competing Algorithms

We are not aware of any application of changing cache size other than when there are two or more programs competing for the same cache. This is even consistent with the metaphor of cache where a bookshelf represents the cache and there are two people sharing it. The bookshelf itself also does not change its size, but the people compete for it. Of course there is one huge difference between this metaphor and our cache model and that is that the cache is self maintained unlike the bookshelf.

The following simple observation provides us with a way to bound the number of cache misses of two algorithms running in pseudo parallel (the CPU switches back and forth between these two programs) and using the same cache. It is an easy observation that these results can be immediately generalized to arbitrarily many pseudo parallel programs.

Observation 16. *Let us consider two programs A and B running in pseudo parallel and sharing the same cache. We can bound the total number of misses of both these algorithms by the sum of cache misses caused by A on a cache of half the size and B on a cache of half the size.*

Proof. Let us consider the following caching strategy: we divide the cache into two halves and each page request of the program A is cached using the first half and each request of B is cached using the second half. Moreover caching in both halves is done optimally.

This is a valid caching algorithm. We are not able to implement it in practice as the caches neither know which program is requesting the page (this would not be such an issue) nor are able to predict which page will be used farthest in the future. Luckily for us we can use the LRU caching policy instead and get similar performance results as we have discussed in depth in Chapter 3.

This caching algorithm makes the same number of cache misses as if both A and B would have been run separately on two different caches of half the size. □

There are programs that behave differently regarding different cache sizes. Only few examples might be:

- A program computing the sum of all elements of an array. This memory access sequence is sometimes called a *scan*. This takes $\Theta(N/B)$ misses regardless of the cache size (where N is the number of elements of the array and B stands for the page size). This bound would be true even for a cache of unit size ($M = 1$).

The same independence on the cache size appears even for searching in a van Emde Boas tree (Prokop [1999] proves it takes $\mathcal{O}(\log_B N)$ page requests to search in such a tree).

- On the other hand sorting an array of elements using funnelsort takes

$$\mathcal{O}(1 + (n/B)(1 + \log_M n))$$

cache misses as shown by Frigo et al. [1999] under the tall cache assumption introduced in Definition 1.

The same case occurs for matrix transposition – under the tall cache assumption Frigo et al. [1999] can transpose an $m \times n$ matrix and cause at most $\mathcal{O}(1 + mn/B)$ misses (the matrix is in row-major layout).

The list of algorithms in the above categories is not comprehensive at all but on the previous examples we have seen algorithms with different demand for cache sizes. It is thus natural to generalize Observation 16 to other divisions than halves.

Observation 17. *Let us consider two programs A and B running in pseudo parallel and sharing the same cache. Let M stand for the cache size and let us consider two natural numbers $M_1, M_2 \in \mathbb{N}$ such that $M_1 + M_2 = M$. We can bound the total number of misses of both these algorithms by the sum of cache misses caused by A on a cache of size M_1 and B on a cache of size M_2.*

Proof. The same argument as in the proof of Observation 16. □

We can even optimize over all these cache subdivisions.

Theorem 18. *Let us consider two programs A and B running in pseudo parallel and sharing the same cache. Let us denote $A(m)$ the number of cache misses caused by the algorithm A on a cache of size m and $B(m)$ the number of cache misses caused by the algorithm B. We can bound the total number of misses of both these algorithms running in pseudo parallel on a cache of size M by*

$$\min_{M_1+M_2=M} A(M_1) + B(M_2).$$

Proof. Immediate consequence of Observation 17. □

4.2 Tall Cache Assumption

We have seen that the tall cache assumption is important for some algorithms. It is quite natural when dealing with matrices in row-major order or column-major order. We can evade this in some cases by using another matrix representation, for instance the Z-order (also called Lebesgue curve or Morton order). The Z-order is given recursively by the Z-pattern over sub matrices, that is we first do the Z-order of the top left sub matrix, than we continue with Z-orders of the top right, bottom left and bottom right sub matrices. These representations are depicted for a matrix A of size 4×4 in Figure 4.1, Figure 4.2, and Figure 4.3

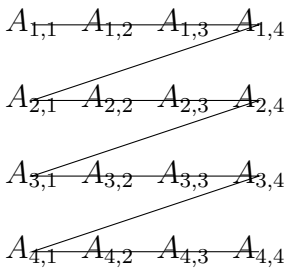

Figure 4.1: Row-major order: the matrix in a one dimensional array is stored as: $A_{1,1}, A_{1,2}, A_{1,3}, A_{1,4}, A_{2,1}, A_{2,2}, A_{2,3}, A_{2,4}, A_{3,1}, A_{3,2}, A_{3,3}, A_{3,4}, A_{4,1}, A_{4,2}, A_{4,3}, A_{4,4}$.

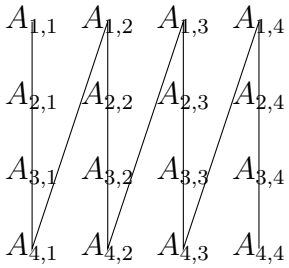

Figure 4.2: Column-major order: the matrix is one dimensionally stored as: $A_{1,1}, A_{2,1}, A_{3,1}, A_{4,1}, A_{1,2}, A_{2,2}, A_{3,2}, A_{4,2}, A_{1,3}, A_{2,3}, A_{3,3}, A_{4,3}, A_{1,4}, A_{2,4}, A_{3,4}, A_{4,4}$.

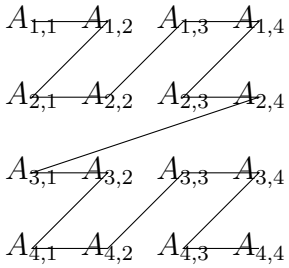

Figure 4.3: Z-order: the matrix in a one dimensional array is stored as: $A_{1,1}, A_{1,2}, A_{2,1}, A_{2,2}, A_{1,3}, A_{1,4}, A_{2,3}, A_{2,4}, A_{3,1}, A_{3,2}, A_{4,1}, A_{4,2}, A_{3,3}, A_{3,4}, A_{4,3}, A_{4,4}$.

respectively. The Z-order has been studied by Chatterjee et al. [1999], Chatterjee et al. [2002], and Frens and Wise [1997].

Many other algorithms use the tall cache assumption. For instance the famous funnelsort also heavily depends on it. Interestingly we know that by the result of Brodal and Fagerberg [2003] that the tall cache assumption is necessary for achieving an optimal comparison based cache-oblivious sorting.

This leads us to the conclusion that the cache size cannot be arbitrarily divided between arbitrary programs. Moreover when considering a situation of multiple programs competing with each other over one cache we would very much like to satisfy all possible tall cache assumptions.

4.3 Regularity

Regularity as in Definition 4 is a basic property of algorithms and data structures described by Prokop [1999] and Frigo et al. [1999]. It is also the basic property that allows us to use LRU instead of the optimal cache replacement policy. In this section we investigate the role of regularity to the competition for a single cache.

Observation 19. *For each $m \in \mathbb{N}$ there is a cache-oblivious algorithm that given n causes m cache misses if the cache is at least m large and causes n cache misses if the cache is smaller than m.*

Proof. We need to construct an algorithm that does not know the cache size but nevertheless causes the specified number of misses. We note that $f_d(i) = m$ is a function computable on processor and thus by Theorem 10 there exists the required algorithm. □

This simple observation might also be proven directly as the previous algorithm corresponds to periodic scanning of m different pages. But what is more important is that such an algorithm is not regular for any cache size M such that $M < m \leq 2M$. This single point of irregularity thus gave us many cache sizes that do not satisfy the regularity condition. One would like to bound the number of such "breaking points" or the number of such irregularity sizes but when we set $m := \lfloor n/\alpha(n) \rfloor$ we have $\lfloor n/2\alpha(n) \rfloor$ sizes M which do not satisfy the regularity condition for an arbitrarily small function α – think of it as the inverse Ackermann function.

One could construct even wilder cache behaviours easily using Theorem 10 or directly Theorem 12. On the other hand these are adversarial algorithms that we do not usually run. Informally the regularity corresponds to the fact that mostly we consider a sub-problem small enough to fit in the cache and if the cache doubles the sub-problem that fits to the doubled cache is at most twice as large. We do not know a natural cache-oblivious algorithm that would disrupt this pattern and would behave like the previous adversarial algorithm.

Conclusion

We have provided a common generalization of two known results that LRU is competitive. This generalization gives a result for perhaps the most practical setting where a cache miss is much longer than a cache hit but both take non-zero amount of time. We have also provided a new view of caches namely the depth model which allows us to easily construct algorithms with given cache behaviours. And we have provided an analysis of what happens when two or more programs share one cache in pseudo parallel. This result also provides practical advice how to design fast algorithms which can be summed as follows the cache-oblivious algorithms that satisfy the regularity condition and are not much dependent on the tall cache assumption behave well.

Open Problems

It would be interesting to investigate what happens in the case when more threads share a cache, but they also share some of their pages. This resembles the problem of parallel algorithms and cache behaviour which was already considered in the literature.

The question of balancing the number of misses and the level of parallelism of several programs seems to be of a great practical importance. However it seems to be hard to theoretically grasp both the formalization of the load-balancing and all details of practical cache implementations in today's computers.

The depth cache model would be even more interesting if we were able to decide whether the simulations provided in Section 3.4.5 can be made in such a way that they cause the same number of cache misses as their counterparts or at least at most some reasonable function of the cache misses of their counterparts. Moreover there is the question of whether there can be a precise simulation of one model in the other one when the access function is not computable on processor.

Bibliography

Alok Aggarwal, Jeffrey Vitter, et al. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. Communication-optimal parallel algorithm for Strassen's matrix multiplication. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 193–204. ACM, 2012.

Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

Laszlo A. Belady, Robert A. Nelson, and Gerald S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of the ACM*, 12(6):349–353, 1969.

Michael A. Bender, Roozbeh Ebrahimi, Jeremy T. Fineman, Golnaz Ghasemiesfeh, Rob Johnson, and Samuel McCauley. Cache-adaptive algorithms. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 958–971. Society for Industrial and Applied Mathematics, 2014.

Michael A. Bender, Erik D. Demaine, Roozbeh Ebrahimi, Jeremy T. Fineman, Rob Johnson, Andrea Lincoln, Jayson Lynch, and Samuel McCauley. Cache-adaptive analysis. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 135–144. ACM, 2016.

Gerth Stølting Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 307–315. ACM, 2003.

Randal Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. *Computer systems: a programmer's perspective*, volume 2. Prentice Hall Upper Saddle River, 2003.

Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra, and Mithuna Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 13th international conference on Supercomputing*, pages 444–453. ACM, 1999.

Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala, and Mithuna Thottethodi. Recursive array layouts and fast matrix multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 13(11):1105–1123, 2002.

Erik D. Demaine. Cache-oblivious algorithms and data structures. *Lecture Notes from the EEF Summer School on Massive Data Sets*, 8(4):1–249, 2002.

Jeremy D. Frens and David S. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In *ACM SIGPLAN Notices*, volume 32, pages 206–216. ACM, 1997.

Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297. IEEE, 1999.

Andrea Lincoln. *Analysis of Recursive Cache-Adaptive Algorithms*. PhD thesis, Citeseer, 2014.

Ishai Menache and Mohit Singh. Online caching with convex costs. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 46–54. ACM, 2015.

James G. Oxley. *Matroid theory*, volume 3. Oxford University Press, USA, 2006.

Enoch Peserico. Paging with dynamic memory capacity. *arXiv preprint arXiv:1304.6007*, 2013.

Harald Prokop. *Cache-oblivious algorithms*. PhD thesis, MIT, 1999.

Roozbeh Ebrahimi Soorchaei. *Cache-Adaptive Algorithms*. PhD thesis, Stony Brook University, 2015.

Luca Trevisan. Combinatorial optimization: Exact and approximate algorithms. *Standford University*, 2011.

List of Figures

List of Abbreviations

$\{x_1, x_2, x_3, \ldots\}$	The set containing elements x_1, x_2, x_3, ...
$\mathbb{N}$	The set of natural numbers $\{0, 1, 2, 3, \ldots\}$.
$\mathbb{N}^+$	The set of positive natural numbers $\{1, 2, 3, \ldots\}$.
$[n]$	The set of the first n positive natural numbers $\{1, 2, \ldots, n\}$.
$\mathbb{Z}$	The set of integers $\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$.
$\lceil x \rceil$	The smallest integer greater or equal to x.
$\lfloor x \rfloor$	The greatest integer smaller or equal to x.
$\Pr[A]$	The probability that an event A occurs.
$\mathbb{E}[X]$	The expectation of a random variable X.
CPU	A central processing unit.
FFT	The discrete fast Fourier transformation algorithm.
I/O	An input/output operation.
ISO	The International Organization for Standardization.
LRU	The least recently used caching algorithm.
OPT	An optimal caching algorithm.
RAM	A random-access memory.