Cache-Adaptive Algorithms

A Dissertation presented
by

Roozbeh Ebrahimi Soorchaei

to
The Graduate School
in Partial Fulfillment of the
Requirements
for the Degree of

Doctor of Philosophy
in
Computer Science

Stony Brook University

August 2015
We, the dissertation committee for the above candidate for the Doctor of Philosophy degree, hereby recommend acceptance of this dissertation.

Michael A. Bender - Dissertation Advisor
Associate Professor, Department of Computer Science

Joseph S. B. Mitchell - Dissertation Advisor
Professor, Department of Applied Mathematics and Statistics

Rob Johnson - Chairperson of Defense
Assistant Professor, Department of Computer Science

Jie Gao - Committee Member
Associate Professor, Department of Computer Science

Jeremy T. Fineman - External Committee Member
Assistant Professor, Department of Computer Science, Georgetown University

This dissertation is accepted by the Graduate School

Charles Taber
Dean of the Graduate School
Abstract of the Dissertation

Cache-Adaptive Algorithms

by

Roozbeh Ebrahimi Soorchaei

Doctor of Philosophy

in

Computer Science

Stony Brook University

2015

Memory efficiency and locality have substantial impact on the performance of programs, particularly when operating on large data sets. Thus, memory- or I/O-efficient algorithms have received significant attention both in theory and practice. The widespread deployment of multicore machines, however, brings new challenges. Specifically, since the memory is shared across multiple processes, the effective memory-size allocated to each process fluctuates over time.

This dissertation studies algorithms in the context of a memory allocation that changes over time, which we call the cache-adaptive setting. The cache-adaptive model applies to operating systems, databases, and other systems where the allocation of memory to processes changes over time.

Analytic techniques are provided for studying the behavior of recursive cache-oblivious algorithms in the cache-adaptive model. These techniques make analyzing algorithms in the cache-adaptive model almost as easy as in the Disk Access Model (DAM) [1]. These techniques are applied to analyze a wide variety of algorithms — Master-Method-style algorithms, Akra-Bazzi-style algorithms, collections of mutually recursive algorithms, and algorithms, such as the cache-oblivious FFT algorithm [31], that break problems of size $N$ into subproblems of size $\Theta(N^c)$.

While the cache-oblivious sorting algorithm Lazy Funnel Sort [15] does not have the former recursive structures, it is nonetheless proved to be optimally cache-adaptive.

Paging algorithms are studied in the context of cache-adaptive model. It is proved that the Least Recently Used (LRU) policy with resource augmentation is competitive with the
optimal offline strategy. Moreover, Belady’s Longest Forward Distance (LFD) [8] policy is shown to remain optimal even when the memory size changes.

Because cache-oblivious algorithms are well understood, frequently easy to design, and widely deployed, there is hope that provably good cache-adaptive algorithms can be deployed in practice.
Dedicated to my beloved wife, Golnaz, and dearest parents, Robab and Asad.
Contents

1 Introduction 1
   1.1 Memory Hierarchy Models ................................................. 3
   1.2 Previous Work ................................................................. 6
   1.3 Thesis Contributions and Organization ............................... 8

2 The Cache-Adaptive Model 11
   2.1 Optimality in the Cache-Adaptive Model ............................ 12
   2.2 Justification for the Cache-Adaptive Model .......................... 13

3 Cache-Adaptive Analysis 15
   3.1 Square Memory Profiles ..................................................... 15
   3.2 Progress Bounds and Progress Optimality in the Cache-Adaptive Model ... 17
   3.3 Square Profiles are Adequately Rich for Studying Progress Optimality .... 19
   3.4 Progress Optimality vs. Competitive Optimality .................... 21
   3.5 Recursions in the Cache-Adaptive Model .............................. 22

4 Optimality Criteria for Recursive Cache-Oblivious Algorithms 24
   4.1 Classes of Recursive Cache-Oblivious Algorithms .................. 24
   4.2 Structure of N-Fitting Square Profiles For Recursive Algorithms .... 26
   4.3 Optimality Criteria For Generalized Compositional Regular Algorithms ... 31
   4.4 Optimality of Generalized Regular Algorithms ........................ 37
   4.5 Optimality Criteria for (a, b, c)-Regular Algorithms ............. 38

5 Deriving Progress Bounds 43
   5.1 A Progress Bound for the Naïve Matrix Multiplication Problem ........ 46
   5.2 A Progress Bound for the Naïve All Pairs Shortest Paths Problem ....... 48
   5.3 Progress Bounds for the LCS and Edit Distance Problems ............ 50
   5.4 A Progress Bound for the Multipass Filter problem ................. 52
   5.5 A Progress Bound for the FFT Problem ............................... 53
# List of Figures

1.1 The DAM model of computation. ................................................. 4
1.2 The ideal-cache model of computation. ........................................ 5
1.3 The changes in memory size are attributed to the arrival/departure of other asynchronous process in the system. .................................. 6
1.4 The cache-adaptive model of computation. ...................................... 7

3.1 The inner square memory profile. ................................................. 16
3.2 The usable profile beneath each square profile. ............................... 20
3.3 Bottomed-out nodes in the cache-oblivious analysis. ....................... 22
3.4 Bottomed-out nodes in the cache-adaptive model. ............................ 23

5.1 The DAG of computation, $G_\psi$, for the LCS and Edit Distance problems. . . 51
5.2 The DAG of computation, $G_\eta$, for the Jacobi Multipass Filter problem. . . 53

9.1 Comparing states of the memory after original I/Os and swapped I/Os. . . 84
Acknowledgments

First, I would like to thank my incredible advisors, Professor Michael A. Bender and Professor Joseph S.B. Mitchell for all their support, patience, encouragement and guidance throughout the past six years.

I am deeply grateful to Michael for his great lessons on good academic writing, giving presentations, and thinking about research problems. I enjoyed our long chats about algorithms and beyond it: books, movies, cooking, and everything else.

I am also very grateful to Joe for being an incredible teacher; everything I know about computational geometry I have learned from him. I also learned from him the value of academic integrity, thoroughness and persistence in research.

I would like to express my sincere gratitude to Professor Jie Gao. Though Jie was not my academic advisor, through most of my PhD career, she acted like one to me. I am deeply indebted to her as I have learned from her how to seek interesting research problems, pursue them with persistence and efficiently write about them. She is undoubtedly one of the most professional, efficient and pleasant individuals I have had the privilege of knowing in my life.

I have also had the privilege of working with the incredibly smart and knowledgeable Professor Rob Johnson. This dissertation and the cache-adaptive project in general would have never happened if it wasn’t for him and his brilliant ideas. I am deeply grateful to him.

Michael Bender introduced me to the research problems concerning cache-adaptivity in the summer of 2012 and this dissertation is the outcome of a collaborative research project done with a fantastic group of researchers including him, Professor Jeremy Fineman, Golnaz Ghasemiesfeh, Professor Rob Johnson and Sam McCauley. I have learned from all of them and I would like to thank them all sincerely.

My deep gratitude goes to brilliant Professors Grant Schoenebeck, Ker-I Ko, Ben Moseley and Jing Chen, whom I have had the privilege of working with on projects that are not included in this dissertation. I have learned a lot about social networks and stochastic processes from Grant, computational complexity from Ker-I, scheduling algorithms from Ben and game theory from Jing.

I have been very fortunate to meet amazing friends at Stony Brook— Moussa, Neda, Mayank, Jon, Pablo and Akshay. They made my time here memorable and I hope that our friendship continues in the future. I also would like to thank all my friends from the algorithms lab and its frequent visitors. I had the privilege of working with some of them, and I am regretful that I didn’t get to work with the others. I enjoyed the lively and colorful environment of the lab and I hope it continues to be that way for the years to come.
I also want to thank Cynthia Scalzo, the extremely delightful graduate secretary of the computer science department, for all the help she has given me over the past six years.

Last but not least, I would like to express my deepest gratitude to my beloved Golnaz and dearest parents, Robab and Asad, for all the selfless love and support they have given me throughout my life. If it wasn’t for their encouragements and their beliefs in my abilities, I wouldn’t have done any of this. I am truly blessed to have them all in my life.

Golnaz and I endured very difficult times at Stony Brook and if it wasn’t for her numerous sacrifices in those circumstances, I wouldn’t have been be able to finish my studies. Golnaz also acted as my best colleague as she was the one who opened my eyes to research problems in social networks and we worked together on a couple of extremely fun papers.
Chapter 1

Introduction

Memory fluctuations are the norm on most computer systems. Each process’s share of memory changes dynamically as other processes start, stop, or change their own demands for memory. This phenomenon is particularly prevalent on multicore computers.

External-memory computations can especially suffer from these fluctuations. Examples include:

* joins and sorts in a database management system (DBMS),

* irregular, I/O-bound shared-memory parallel programs,

* cloud computing services running on shared hardware,

and essentially any external-memory computation running on a time-sharing system.

Database and scientific computing researchers and practitioners have recognized this problem for over two decades [18, 41, 42], and have developed many sorting and join algorithms [56, 43, 44, 57, 58, 32] that offer good empirical performance when memory changes size dynamically. However, most of these algorithms are designed to perform well in the common case, but they perform poorly in the worst case [6, 7].

In contrast to this reality, most of today’s performance models for external-memory computation assume a fixed internal memory size $M$ (see, e.g., [51]) and hence algorithms designed in these models cannot cope when $M$ changes. This means that most external-memory algorithms cannot take advantage of memory freed by the departure of other processes, and they can begin thrashing if the system takes back too much memory.

Thus, there is a gap between the state of the world, where memory fluctuations are the rule, and today’s tools for designing and analyzing external-memory algorithms, which assume fixed internal-memory sizes.
Barve and Vitter [6, 7] took the first major step towards closing this gap by showing that worst-case, external-memory bounds are possible in an environment where RAM changes size. Barve and Vitter generalized the DAM model [1] to allow the memory size $M$ to change periodically. They give worst-case optimal algorithms for sorting, FFT, matrix multiplication, LU decomposition, permutation, and buffer trees. However, their algorithms are quite complicated to describe, and possibly more complicated to implement.

On the other hand, the empirically efficient adaptive algorithms do not have theoretical performance bounds and are not commonly used in today’s DBMSs, even though the practical need for such algorithms has, if anything, increased. We attribute this lacuna to the difficulty of designing, analyzing, and implementing memory-adaptive algorithms.

We define the cache-adaptive (CA) model, an extension of the DAM [1] and ideal cache [31, 47] models. The CA model describes systems in which the available memory to a process/algorithm can change dynamically.

We show that cache adaptivity is sometimes achievable via more manageable algorithms, by leveraging the cache-oblivious technology [31, 47]. Cache-oblivious algorithms are not parameterized by the memory hierarchy, yet they often achieve provably optimal performance for any static hierarchy. We characterize how these algorithms adapt when the memory changes dynamically.

We put forward analytic tools that simplify the analysis of algorithms in the cache-adaptive model. We exhibit this simplicity by studying several classes of cache-oblivious algorithms in the cache-adaptive setting.

* We characterize the optimality criteria of recursive cache-oblivious algorithms that adhere to a Master-Method [25] style of recursion, which we call $(a, b, c)$-regularity.

Through this analysis, we prove the optimal cache-adaptivity of a number of most useful cache-oblivious algorithms such as the cache-oblivious in-place naïve matrix multiplication algorithm of Frigo et al. [31], the cache-oblivious Floyd-Warshall All Pairs Shortest Paths algorithm of Park et al. [45] and cache-oblivious matrix transpose algorithm of Frigo et al. [31].

* We generalize this analysis to handle non-homogeneous recursions of the Akra-Bazzi form [2] and even collections of multiple/mutually recursive algorithms. We offer a general recipe for figuring out whether a recursive algorithm is optimally cache-adaptive.

We use this recipe to show the optimal cache-adaptivity of mutually recursive algorithms such as the cache-oblivious dynamic programming algorithms of Chowdhury  

---

1 Also called the external-memory (EM) or I/O model.
and Ramachandran [21] for Longest Common Subsequence (LCS) and Edit Distance problems and the cache-oblivious Jacobi Multipass Filter Algorithm of Prokop [47].

* Our analytic techniques work for algorithms that do not adhere to the general recursion forms. As an example, we use them to exhibit that the cache-oblivious Fast Fourier Transform (FFT) algorithm of Frigo et al. [31] is a $O(\log \log)$ factor from being optimal in the cache-adaptive model.

* We also establish that the cache-oblivious Lazy Funnel Sort (LFS) algorithm of Brodal and Fagerberg [15], which falls outside the recursive classes described above, is optimally cache-adaptive.

* We establish that cache-obliviousness does not always lead to cache-adaptivity, by proving that a variation of the cache-oblivious naive matrix multiplication algorithm of Frigo et al. [31], MM-SCAN, that is optimal in the DAM model, is a $\Theta(\log N)$ factor away from being optimal in the cache-adaptive model when solving problem instances of size $N$.

Alongside studying algorithms, we study page replacement policies. We prove that the online Least Recently Used (LRU) policy with resource augmentation is competitive with the optimal policy in the CA model. We also establish that Belady’s Longest Forward Distance (LFD) [8] policy is an optimal offline page replacement policy in the CA model.

The memory-adaptive model of Barve and Vitter [6] allows algorithms to know about the future memory allocations a number of steps ahead. We characterizes the performance of optimal algorithms designed in their model in our cache-adaptive setting by showing that if those algorithms are given roughly twice as much knowledge about the future memory allocations, they remain optimal in the cache-adaptive setting.

Because cache-oblivious algorithms are well understood, frequently easy to design, and widely deployed, there is hope that provably good cache-adaptive algorithms can be deployed in practice. We hope that our analyses gives algorithm designers clear guidelines for creating optimally cache-adaptive algorithms.

1.1 Memory Hierarchy Models

Numerous memory models have been proposed to analyze the performance of external memory algorithms. Here, we give a summary of a few of the most relevant memory models. Then, we briefly introduce the cache-adaptive model of computation. A thorough exposition of this model is presented in chapter 2.
The DAM model of computation  In the DAM model of computation [1], the system consists of a two-level memory hierarchy, comprising an internal memory of size $M$ and an external memory that is arbitrarily large. Data is transferred between the memory and disk in chunks of fixed size $B$. Figure 1.1 gives a schematic view of the DAM model. A DAM algorithm manages its own page replacement, and the internal memory is fully associative. In the DAM model, in-memory computation comes for free, and performance is measured by the number of I/Os or block transfers, which serves as a proxy for running time. Thus, in each time step\(^2\) of the algorithm one block is transferred between memory and disk.

The DAM model has been the most successful model for analyzing external memory algorithms for nearly four decades. DAM is not only a simple and elegant theoretical framework that has harbored the conception of hundreds of algorithms and data structures, but also is a very good and practical approximation of the Input/Output (I/O) communication cost between internal and external memories. The practicality and widespread usage of external memory algorithms in real industrial computation systems is a testament to the later claim. For comprehensive surveys on external memory algorithms and data structures see [50, 51, 3, 4].

The ideal-cache model  A cache-oblivious algorithm [31] is analyzed using the ideal-cache model [31, 47], which is the DAM model augmented to include automatic, optimal page replacement. The algorithm is not parameterized by $M$ or $B$, but it is analyzed in terms of $M$ and $B$. Figure 1.2 gives a schematic view of the ideal-cache model. The beauty of this restriction is that an optimal cache-oblivious algorithm is simultaneously optimal for any fixed choice of $M$ and $B$.

\(^2\)Often when we write external-memory and cache-oblivious algorithms, we avoid the word “time” and express performance in terms of I/Os. In the cache-adaptive context, we prefer to use the word "time step" explicitly.
Automatic page replacement in the ideal-cache model is necessary because the algorithm has no knowledge of $B$ or $M$. The optimal page replacement assumption is justified since the Least Recently Used (LRU) page replacement policy is constant-competitive when given a constant memory augmentation [49].

Figure 1.2: The ideal-cache model of computation.


The memory-adaptive model Barve and Vitter [7] generalize the DAM model to allow the memory size $M$ to change periodically. They assume that the size of the available memory to the external memory algorithm changes in a sequence of allocation phases. In each allocation phase, $S_i$, the algorithm is given $S_i$ blocks of memory with $2S_i$ available I/Os. The algorithm is given explicit knowledge about the current and the next sizes of allocation phases, and the number of I/Os left in the current phase.

Unfortunately, since its conception, the memory-adaptive model of Barve and Vitter has not seen much follow-up work.

The cache-adaptive model The cache-adaptive (CA) model extends the DAM and ideal-cache models. In the CA model, the memory size is not fixed; it can change during an algorithm’s execution. We attribute the changes to the memory size to arrival/departure of other asynchronous processes in the system under consideration (See fig. 1.3).

As with the DAM model, computation is free and the performance of algorithms is measured in terms of I/Os. Figure 1.4 gives a schematic view of the cache-adaptive model. As in the ideal-cache model, when we consider a cache-oblivious algorithm in the CA model, we assume that the system manages the content of the memory automatically. But, non-oblivious algorithms can manage the content of memory on their own.
The **memory profile** is the function $m : \mathbb{N} \to \mathbb{N}$, which indicates the size, in blocks, of the memory at the time of the $t$th I/O.\(^3\) The **memory profile in machine words** is the function $M(t) = Bm(t)$.

We place no restrictions on how the memory size changes from one I/O to the next (unlike previous approaches \([6]\)). However, since an algorithm can only load one block into memory per time step, it may not be able to take advantage of all available memory immediately. Thus, we assume that memory only increases by 1 block per time step, i.e. that

$$m(t + 1) \leq m(t) + 1.$$  \hfill (1.1)

When the size of memory decreases, a large number of blocks may need to be written back to disk, depending on whether the blocks are dirty. In this works, we do not charge these write-backs to the application.

The cache-adaptive model assumes that an algorithm can query the current size of memory, $m(t)$, at each time step $t$.

### 1.2 Previous Work

Barve and Vitter \([7]\) generalize the DAM model to allow the memory size $M$ to change periodically. They assume that the size of the available memory to the external memory

---

\(^3\)Throughout, we use the terms *block* and *page* interchangeably.
algorithm changes in a sequence of allocation phases. In each allocation phase, \( S_i \), the algorithm is given \( S_i \) blocks of memory with \( 2S_i \) available I/Os. The algorithm is given explicit knowledge about three variables:

* the current size of memory, \( S_i \),
* how many I/Os are left in this allocation phase,
* and the size of memory in the next allocation phase, \( S_{i+1} \).

They give optimal sorting, matrix multiplication, LU decomposition, FFT, and permutation algorithms. They also present a dynamically optimal (in an amortized sense) memory-adaptive version of the buffer tree data structure [7]. Their algorithms and analyses are complicated. Probably due to this complexity, their work has seen very little follow-up work in more than 15 years.

Empirical studies of adaptivity have been carried out by Zhang and Larson [57] and Pang, Carey, and Livny [43, 44] on memory-adaptive sorting and join algorithms. Other papers discussing aspects of adaptivity include [6, 7, 18, 32, 41, 42, 44, 56, 58]. However, most of these algorithms are designed to perform well in the common case, but they perform poorly in the worst case as shown by Barve and Vitter [6, 7].

The notion of cache-obliviousness was proposed by Frigo et al. [47, 31]. Because cache-oblivious algorithms can be optimal without resorting to memory parameters, they can be uniformly optimal on unknown, multilevel memory hierarchies [31]. Consult [26] for a survey of cache-oblivious algorithms. See [53, 38, 17, 31, 12, 11, 22, 20, 54, 30, 19] for discussions of implementations and performance analysis of cache-oblivious algorithms. See [16, 10] for discussions on the limits of cache-obliviousness.

Cache-oblivious programming has been used as a framework to design algorithms for multicore systems. Blelloch et al. [13] define a class of recursive algorithms, HR, that per-
form well on multicore systems[13]. Cole and Ramachandran [23] define BP and HBP, two classes of recursive cache-oblivious algorithms that also behave well on multicores. Blelloch, Gibbons, and Simhadri [14] prove results for algorithms with low recursive depth. It is an open research question to determine which of these classes are also cache-adaptive.

There are also paging results in different models where the memory size fluctuates. Peserico [46] considers an alternative model for page replacement policies when the memory size fluctuates. Peserico’s page-replacement model does not apply to the cache-adaptive model, because the increases/decreases in the cache size appear at specific locations in the page-request sequence, rather than at specific points in times.

Katti and Ramachandran consider page replacement polices for multicore shared-cache environments [37]. Several authors have considered other aspects of paging where the size of internal memory or the pages themselves vary [35, 5, 55, 33, 40, 39] e.g., because several processes share the same cache. For example, [39] considers a model where the application itself adjusts the memory size, and [35, 55] consider a model where the page sizes vary.

1.3 Thesis Contributions and Organization

We formally introduce the cache-adaptive model in chapter 2. We define the notion of a memory profile that describes changes in allocations of memory to an algorithm over time. We then argue that competitive optimality is the the natural choice for studying optimality of algorithms in this model (see section 2.1).

Chapter 3 provides analytic tools for studying the behavior of algorithms in the cache-adaptive model. We introduce square memory profiles, a class of profiles that are well-behaved and easy to work with. We show that in most settings an analysis on square profiles transfers to all memory profiles. It should be noted that the memory adaptive model of Barve and Vitter [6] is completely defined on square profiles.

In section 3.2, we present an axiomatization of an algorithm’s progress in solving a problem in the cache-adaptive model. We state almost all of our optimality criteria theorems (in chapter 4) in terms of problems that have progress functions associated with them and these progress functions should satisfy our axiomatization. Section 3.2 also defines the notion of progress optimality. Progress optimality views a memory profile as distributing a resource (memory) over time. In this perspective, an algorithm is optimally progressing if it uses this resource at maximum (optimal) capacity most of the time.

Later in chapter 5, we provide tools to explicitly derive progress functions for problems. To this end, we use machinery from lower bound proof techniques of the DAM model, like the red-blue pebble game of Hong and Kung [34], the red pebble game of Savage [48], and
the information-tree lower bound technique of Aggarwal and Vitter [1]. The tools presented in chapter 5 allow for a seamless porting of optimality analysis in the DAM model to the optimality analysis in the cache-adaptive model.

In section 3.3, we prove that progress optimality on square profiles and progress optimality on general profiles are equivalent in the cache-adaptive model. Since studying optimality on square profiles is considerably simpler than studying it on general memory profiles, we restrict our analysis to square profiles.

In section 3.4, we prove that if an algorithm is optimally progressing, then it is optimally cache-adaptive in the CA model. However, it remains an open question whether all competitively optimal algorithms are also optimally progressing in the CA model.

In chapter 4, we characterize optimality criteria for several classes of recursive cache-oblivious algorithms. These classes include Master-Method-style recursions [25], Akra-Bazzi-style recursions [2] and classes of multiple/mutual recursions composed from Akra-Bazzi style functions. We offer a general recipe for figuring out whether a recursive algorithm is optimally cache-adaptive.

Chapter 6 applies the recipe theorems proved in chapter 4 together with explicit progress bounds derived in chapter 5 to exhibit the cache-adaptive optimality of several cache-oblivious algorithms:

* the cache-oblivious in-place naïve matrix multiplication algorithm of Frigo et al. [31],
* the cache-oblivious Floyd-Warshall All Pairs Shortest Paths algorithm of Park et al. [45],
* the cache-oblivious Jacobi Multipass Filter algorithm of Prokop [47],
* the cache-oblivious dynamic programming algorithms of Chowdhury and Ramachandran [21] for Longest Common Subsequence (LCS) and Edit Distance problems,
* and the cache-oblivious matrix transpose algorithm of Frigo et al. [31].

In chapter 7, we establish that cache-obliviousness does not always lead to cache-adaptivity. We prove that a variation of the cache-oblivious naïve matrix multiplication algorithm of Frigo et al. [31], the MM-SCAN algorithm, that is optimal in the DAM model, is a $\Theta(\log N)$ factor away from being optimal in the cache-adaptive model when solving problem instances of size $N$.

In section 7.2, we show that the analytic techniques of chapter 4 can be used to analyze algorithms that don’t fit in the general recipe theorem of chapter 4. As an example, we show that the cache-oblivious FFT algorithm of Frigo et al. [31] is a $O(\log \log N)$ factor away from being optimal when solving problem instances of size $N$. 
In chapter 8, we establish that the Lazy Funnel Sort algorithm of Brodal and Fagerberg [15], which falls outside the recursive classes we have studied thus far, is optimally progressing and cache-adaptive.

We study page replacement policies in the CA model in chapter 9. In section 9.1, we prove that the online Least Recently Used (LRU) policy with resource augmentation is competitive with the optimal policy in the CA model. We also show that Belady’s Longest Forward Distance (LFD) policy [8] is an optimal offline policy in the CA model, see section 9.2.

In chapter 10, we study the notion of foresight into future allocations of memory. The memory-adaptive model of Barve and Vitter [6] allows algorithm to know the future memory allocations a number of steps ahead. We formalize this look-ahead mechanism by the notion of \( k \)-prescience.

We show that any \( k \)-prescient algorithm that is optimal on square profiles is optimal on all profiles if it is given \( 2k + 1 \)-prescience. This result characterizes the performance of optimal algorithms designed in the Barve and Vitter’s memory-adaptive model in our cache-adaptive model.

Finally, in section 10.1 we exhibit that square profiles are adequately rich for studying optimality of cache-obliviuous algorithms in the cache-adaptive model.
Chapter 2
The Cache-Adaptive Model

The cache-adaptive (CA) model extends the DAM and ideal-cache models. As with the DAM model, computation is free and the performance of algorithms is measured in terms of I/Os. As in the ideal-cache model, when we consider a cache-oblivious algorithm in the CA model, we assume that the system manages the content of the memory automatically. But, non-oblivious algorithms can manage the content of memory on their own. In the CA model, the memory size is not fixed; it can change during an algorithm’s execution.

Definition 2.1. The memory profile is the function \( m : \mathbb{N} \rightarrow \mathbb{N} \), which indicates the size, in blocks, of the memory at the time of I/O number \( t \).\(^1\) The memory profile in machine words is the function \( M(t) = Bm(t) \).\(^2\)

We assume that the memory profile never drops to 0 blocks, meaning that for all \( t \), \( m(t) \geq 1 \) and \( M(t) \geq B \).

Definition 2.2. We say that a memory profile is usable if the increases are limited by 1 block per time step, i.e. that

\[
m(t + 1) \leq m(t) + 1.
\]

We place no restrictions on how the memory size changes from one I/O to the next (unlike previous approaches [6]). However, since an algorithm can only load one block into memory per time step, it may not be able to take advantage of all available memory immediately. Therefore, it suffices to work with usable profiles when analyzing algorithms in the cache-adaptive model.

When the size of memory decreases, a large number of blocks may need to be written back to disk, depending on whether the blocks are dirty. In this dissertation, we do not

\(^1\)Throughout, we use the terms block and page interchangeably.

\(^2\)Usually a machine word is considered to be the size of an integer number in the architecture of the machine. In most of today’s architectures, an integer is 4 bytes.
charge these write-backs to the application, but we believe that the model can be extended to do so.

In this work, we focus exclusively on memory-monotone algorithms, defined as follows.

**Definition 2.3.** An algorithm is **memory monotone** if it runs no more than a constant factor slower when given more memory.

Memory monotonicity is used to rule out degenerate algorithms that use a fast algorithm when given a non-square profile but a slow algorithm when given a square profile (see the next section for the definition of a square profile). All cache-oblivious algorithms and almost all “reasonable” DAM-model algorithms are memory monotone. Many paging algorithms like LRU and Belady’s optimal offline paging algorithm [8] are also memory monotone. One notable exception is the FIFO paging algorithm, which was recently shown not to be memory monotone [9, 29].

**Tall-Cache Assumption** The performance bounds for cache-oblivious algorithms commonly rely on a so-called **tall-cache** assumption, which means that there is a constant $H(B)$, polynomial in $B$, such that the memory size has to satisfy $M \geq H(B)$. For example, for cache-oblivious sorting or matrix transpose, $H(B) = \Theta(B^2)$ [31]. We support these kinds of analyses in the CA model as follows.

**Definition 2.4.** In the CA model, we say that a memory profile $M$ is **H-tall** if for all $t \geq 0$, $M(t) \geq H(B)$.

**Assumption 2.5.** When we consider a $H(B)$-tall memory profile $M$, we assume without loss of generality that $M(0) = H(B)$.

### 2.1 Optimality in the Cache-Adaptive Model

Optimality in the CA model captures the spirit of optimality in the DAM model, but accommodates the complications presented by the changing memory size. Roughly speaking, an algorithm is considered “asymptotically optimal” if its worst-case I/O performance is within a constant factor of the best possible. In the DAM model, the memory size is static, so the extra I/Os can go anywhere—granting the algorithm a constant factor more time and more speed are equivalent. In contrast, the effectiveness of each I/O in the CA model varies with the memory profile. Thus, granting an algorithm more time at the end is a nonstarter, because the memory profile could drop precipitously. The CA model instead defines optimality by granting the algorithm extra speed.
Definition 2.6. Giving an algorithm A \textbf{c-speed augmentation} means that A may perform \( c \) I/Os in each step of the memory profile.

Definition 2.7 \textbf{(Speed-augmented profiles)}. If \( m \) is any memory profile, under \textbf{\( c_1 \)-speed augmentation} \( m \) is scaled into the profile \( m'(t) = m([t/c_1]) \).

Definition 2.8. An algorithm A that solves problem \( P \) is \textbf{competitively optimal} in the cache-adaptive model if there exists a constant \( c \) such that on all memory profiles and all sufficiently large input size \( N \), the worst-case running time of a \( c \)-speed augmented \( A \) is better than the worst-case running time of any other (non-augmented) memory-monotone algorithm.

Note that this notion of optimality requires an algorithm to outperform all other memory-monotone algorithms, including non-oblivious algorithms that optimize cache usage by looking ahead in the profile.

As in the DAM model, memory augmentation is needed to show that LRU is constant competitive. We also use memory augmentation to simplify our analyses.

Definition 2.9. For any memory profile \( m \), we define a \textbf{\( c_2 \)-memory augmented} version of \( m \) as the profile \( m'(t) = c_2m(t) \). Running an algorithm A with \textbf{\( c_2 \)-memory augmentation} on the profile \( m \) means running A on the \( m' \).

2.2 Justification for the Cache-Adaptive Model

The CA model is intended to capture performance on systems in which memory size changes asynchronously in response to external events, such as the start or end of other tasks. Since these events are asynchronous, changes in memory size should be pegged to wall-clock time.

We construct the CA model as an extension of the DAM model, i.e., the DAM model is the special case that \( m(t) \) is static. The DAM model, however, has no explicit notion of time. Instead, performance is measured by the number of I/Os. This I/O counting can be reinterpreted as time, with an I/O taking unit time and computation taking 0 time. On real systems, I/Os dominate computation, so the number of I/Os is a good approximation to wall-clock time for I/O-bound algorithms.

In the CA model, we explicitly measure time in terms of I/Os. This is the same approach adopted by Barve and Vitter [6]. For example, the arrival of a new process at a particular time can be modeled by having memory drop after a certain number of I/O steps.
Page Replacement in the CA Model  The CA model assumes that page replacement is performed automatically by the system for cache-oblivious algorithms. We show that optimal replacement can be simulated by a resource-augmented online LRU policy in the CA model. We prove that LRU with 4-memory and 4-speed augmentation always completes sooner than OPT (see section 9.1).

For all other non-oblivious algorithms in the cache-adaptive model, we assume that the algorithm is in charge of performing the page replacement.
Chapter 3

Cache-Adaptive Analysis

In this chapter we provide analytic tools for studying the behavior of algorithms in the cache-adaptive model.

3.1 Square Memory Profiles

We introduce a class of memory profiles called square profiles. Proving cache-adaptive optimality on square profiles is easier and cleaner. Moreover, as we show multiple times during this thesis, optimality on square profiles is extendable to all profiles.

Definition 3.1. A memory profile \( m \) is square if there exist boundaries \( 0 = t_0 < t_1 < \ldots \) such that for all \( t \in [t_i, t_{i+1}) \), \( m(t) = t_{i+1} - t_i \). In other words, a square memory profile is a step function where each step is exactly as long as it is tall (see fig. 3.1).

We define an inner square profile \( m' \) for a profile \( m \) by placing maximal squares below \( m \), proceeding left to right, as illustrated in fig. 3.1. The inner square boundaries are the left/right boundaries of the squares in \( m' \).

Definition 3.2. For a memory profile \( m \), the inner square boundaries \( t_0 < t_1 < t_2 < \ldots \) of \( m \) are defined as follows: Let \( t_0 = 0 \). Recursively define \( t_{i+1} \) as the largest integer such that \( t_{i+1} - t_i \leq m(t) \) for all \( t \in [t_i, t_{i+1}) \). The inner square profile of \( m \) is the profile \( m' \) defined by \( m'(t) = t_{i+1} - t_i \) for all \( t \in [t_i, t_{i+1}) \).

The following lemma enables us to analyze algorithms even in profiles where the memory size drops precipitously. Intuitively, the lemma states that the \((i + 1)\)st interval in the inner square profile is at most twice as long as the \(i\)th interval, and the available memory in the original memory profile during the \((i + 1)\)st interval is at most four times the available memory in the \(i\)th interval of the inner square profile.
Lemma 3.3. Let $m$ be a usable memory profile (see definition 2.2). Let $t_0 < t_1 < \ldots$ be the inner square boundaries of $m$, and let $m'$ be the inner square profile of $m$.

1. For all $t$, $m'(t) \leq m(t)$.

2. For all $i$, $t_{i+2} - t_{i+1} \leq 2(t_{i+1} - t_i)$.

3. For all $i$ and $t \in [t_{i+1}, t_{i+2})$, $m(t) \leq 4(t_{i+1} - t_i)$.

Proof. 1. By construction of $m'$ in definition 3.2.

2. By construction of the inner square boundaries in definition 3.2, for each $i$ there exists a $t_i^* \in [t_i, t_{i+1}]$ such that $m(t_i^*) = m'(t_i) = t_{i+1} - t_i$. Since $m$ can only increase by one in each time step, $m(t_{i+1}) \leq m(t_i^*) + (t_{i+1} - t_i^*)$. Substituting for $m(t_i^*)$ and because $t_i^* \geq t_i$, we obtain $m(t_{i+1}) \leq 2(t_{i+1} - t_i)$. Also by construction the inner square boundaries, $t_{i+2} - t_{i+1} \leq m(t_{i+1})$, we must have $t_{i+2} - t_{i+1} \leq 2(t_{i+1} - t_i)$.

3. Similarly, for all $t \in [t_{i+1}, t_{i+2})$, $m(t) \leq m(t_i^*) + (t - t_i^*) \leq (t_{i+1} - t_i) + (t_{i+2} - t_i) = (t_{i+1} - t_i) + (t_{i+2} - t_{i+1}) + (t_{i+1} - t_i)$. Hence we get $m(t) \leq 4(t_{i+1} - t_i)$.

\[\square\]
3.2 Progress Bounds and Progress Optimality in the Cache-Adaptive Model

In this section, we present an axiomatization of the notion of an algorithm’s progress in the cache-adaptive model. In chapter 5, we show that lower bound proofs in the DAM model could be used to derive explicit functions, which satisfy our axiomatization, for various problems that we study. This allows for a seamless porting of DAM lower bounds into the cache-adaptive model.

A problem has a **progress bound** if there exists (1) a **progress requirement function** \( R(N) \) representing the amount of progress that an algorithm must make to solve a problem of size \( N \) and (2) a **progress limit function** \( \rho(M) \) representing the most progress that any algorithm can make when running on profile \( M \).

**Notation 3.4.** We represent a memory profile of finite duration \( n \) as a string of integers \( S_0S_1\cdots S_{n-1} \), meaning the profile has \( S_i \) machine words \( i \) time steps after its start. String concatenation is written as \( m_1 \parallel m_2 \). We write \( \square_N \) to represent the profile \( S_0\cdots S_{\lceil N/B \rceil-1} \) where \( S_i = N \) for all \( i \). We also treat a profile \( M \) as a function, i.e., we write \( M(t) \) to indicate the size of memory \( t \) time steps after the start of the profile. If \( t \geq n \), then \( M(t) = 0 \).

We define an ordering on the profiles of finite duration. We use this ordering to compare the progress of different memory profiles.

**Definition 3.5.** Let \( M \) and \( U \) be any two profiles of finite duration. We say that \( M \) is **smaller than** \( U \), \( M \prec U \), if there exists profiles \( L_1, L_2\ldots L_k \) and \( U_0, U_1, U_2\ldots U_k \), such that \( M = L_1\parallel L_2\ldots\parallel L_k \) and \( U = U_0\parallel U_1\parallel U_2\ldots\parallel U_k \), and for each \( 1 \leq i \leq k \),

(i) If \( d_i \) is the duration of \( L_i \), \( U_i \) is a profile with duration \( \geq d_i \).

(ii) As standalone profiles, \( L_i \) is always below \( U_i \). \(^1\)

**Definition 3.6.** A function \( \rho : \mathbb{N}^* \rightarrow \mathbb{N} \) is **monotonically increasing** if for any pair of profiles \( M \) and \( U \), if \( M \prec U \) then \( \rho(M) \leq \rho(U) \).

**Definition 3.7.** A monotonically increasing function \( \rho : \mathbb{N}^* \rightarrow \mathbb{N} \) is **square-additive** if

(i) \( \rho(\square_M) \) is bounded by a polynomial in \( M \),

(ii) \( \rho(\square_{M_1} \parallel \cdots \parallel \square_{M_k}) = \Theta(\sum_{i=1}^k \rho(\square_{M_i})) \).

\(^1\) A memory profile is a function of time. A function \( f \) is below function \( g \) on interval \( I \) if \( f \leq g \) on \( I \).
Definition 3.8. A problem has a **progress bound** if there exists a monotonically increasing polynomial-bounded **progress requirement function** $R : \mathbb{N} \to \mathbb{N}$ and a square-additive **progress limit function** $\rho : \mathbb{N}^* \to \mathbb{N}$ such that: For any profile $M$, if $\rho(M) < R(N)$, then no memory-monotone algorithm running under profile $M$ can solve all problem instances of size $N$.

We also refer to the progress limit function $\rho$ simply as the **progress function** or **progress bound**. The notion of progress is problem-specific.

**Example 3.9.** The external-memory sorting lower bound [1] says that, given $M$ memory, a comparison-based sorting algorithm can learn at most $O(BM \log M)$ bits of information per I/O and must learn $\Omega(N \log N)$ bits to sort. Thus, $R(N) = \Omega(N \log N)$ and $\rho(M) = O(\sum_{t=0}^{\infty} BM(t) \log M(t))$.

Some progress bounds, e.g., for sorting, bound the maximum possible progress per I/O, whereas others bound the maximum possible progress that can be made over multiple I/Os. For example, the progress bound for standard matrix multiplication states that, given $M$ memory and $M/B$ I/Os, no algorithm can perform more than $O(M^{3/2})$ elementary multiplications [34, 36], see chapter 5 for a complete derivation of these two progress bounds.

**Observation 3.10.** If algorithm $A$ has linear space complexity, then the amount of progress $R(N)$ that $A$ must complete in order to solve a problem of size $N$ is $O(\rho(\Box N))$.

In the cache-adaptive model, an algorithm is optimal if it can beat every other algorithm using a constant speed augmentation (see definition 2.8).

A memory profile can be seen as a distribution of a resource (memory) over time. In this interpretation, using the memory profile in an optimal capacity can be taken as a measure of optimality. We define an algorithm to be **optimally progressing** if it always makes within a constant factor of the maximum possible progress on a profile.

**Definition 3.11.** For an algorithm $A$ and problem instance $I$ we say a profile $M$ of length $\ell$ is **I-fitting** if $A$ requires exactly $\ell$ time steps to process input $I$ on profile $M$. A profile $M$ is **N-fitting** if $A$, given profile $M$, can complete its execution on all instances of size $N$, and there exists at least one instance $I$ of size $N$ for which $M$ is I-fitting.

**Definition 3.12.** An algorithm $A$ for problem $P$ is **optimally progressing with respect to $\rho$** (or simply optimally progressing if $\rho$ is understood) if, for every N-fitting profile $M$, $\rho(M) = O(R(N))$.

In section 3.4, we investigate the relationship between the competitive optimality (definition 2.8) and progress optimality (definition 3.12) notions in the cache-adaptive model. In most parts of the current thesis, we work with the latter notion, as it is more intuitive and much easier to work with.
3.3 Square Profiles are Adequately Rich for Studying Progress Optimality

In this section, we exhibit that the class of all square profiles is adequately rich to capture the intricacies of progress optimality analysis on general memory profiles. In other words, analyzing progress optimality in the cache-adaptive model can be restricted to square profiles.

We exhibit this richness by proving two results. We first prove that the progress bound of a profile $M$ and its inner square profile $M'$ are within a constant factor of each other (see theorem 3.13 below). An immediate corollary of the above result is that if an algorithm is optimally progressing on square profiles, then it is optimally progressing on all memory profiles.

However, as an observant reader might have noticed, not all square profiles are usable (see definition 2.2). We prove that for each square profile $M$ below $U$ (see definition 2.2). We prove that for each square profile $M$, there exists a usable memory profile $U$ below $M$ such that $\rho(U) = \Theta(\rho(M))$ (see theorem 3.15 below).

**Theorem 3.13.** If $\rho$ is square additive and $M$ is a profile with inner square profile $M'$, then $\rho(M) = \Theta(\rho(M'))$.

**Proof.** Since $\rho$ is monotonic and $M'(t) \leq M(t)$ for all $t$, $\rho(M') \leq \rho(M)$. Let $M'_{4,4}$ be the 4-speed and 4-memory augmented version of $M'$. Since $\rho$ is square-additive and $\rho(\square_N)$ is bounded by a polynomial in $N$, we have that $\rho(M'_{4,4}) = O(\rho(M'))$.

We prove that $M \prec M'_{4,4}$, and by monotonicity of $\rho$ we get that

$$\rho(M') \leq \rho(M) \leq \rho(M'_{4,4}) = O(\rho(M')),$$

which means that $\rho(M) = \Theta(\rho(M'))$.

Let $M[S_i]$ denote the profile $M$ restricted to the interval $S_i$. Let $k + 1$ be the number of squares in $M'$. Define $L_1 = M[S_1 \cup S_2], L_2 = M[S_3], \ldots, L_k = M[S_{k+1}]$, and note that $M = L_1 \parallel L_2 \parallel \ldots \parallel L_k$. Also, define $U_i$ to be a 4-speed 4-memory augmented version of square $S_i$ and allow $U'_k = U_k \parallel U_{k+1}$. Notice that $M'_{4,4} = U_1 \parallel U_2 \parallel \ldots \parallel U_{k-1} \parallel U'_k$.

In order to prove that $M \prec M'_{4,4}$, we show that $U'_k$ and each $U_i$, $1 \leq i \leq k - 1$ satisfies the two conditions of definition 3.5.

We start by considering $U_1$ and $L_1$. If $M$ is $H(B)$-tall, by assumption 2.5 we have that $M(0) = H(B)$. By definition 3.2, we have that $t_1 = H(B)$ and since $m(t + 1) \leq m(t) + 1$, we have that for all $t \in [0, t_1)$, $M(t) \leq 2t_1$. Moreover, by lemma 3.3, we know that $S_2$ is at most twice as long as $S_1$ and for all $t \in [t_1, t_2)$, $M(t) \leq 4(t_1 - t_0) = 4|S_1|$. Hence, $t_2 \leq 3t_1$, and for all $t \in [0, t_2)$, $M(t) \leq 4|S_1|$. Because $U_1$ is a 4-speed 4-memory augmented version of $S_1$, we have that (i) $U_1$ has a longer duration than $L_1 = M[S_1 \cup M_2]$, and that (ii) $L_1$ is below $U_1$. 

19
Similarly, for each $U_i$, $2 \leq i \leq k$, by lemma 3.3, we know that $S_{i+1}$ is at most twice as long as $S_i$ and for all $t \in [t_{i+1}, t_{i+2})$, $M(t) \leq 4(t_{i+1} - t_i) = 4|S_i|$. Because $U_i$ is a 4-speed 4-memory augmented version of $S_i$, we have that (i) $U_i$ has a longer duration than $L_i = M[S_{i+1}]$, and that (ii) $L_i$ is below $U_i$.

By repeating the above argument for $U_k$, we see that (i) $U_k$ has a longer duration than $L_k = M[S_{k+1}]$, and that (ii) $L_k$ is below $U_k$. This means that $U_k' = U_k \parallel U_{k+1}$ also satisfies both of the above conditions.

Therefore, we have shown that $M < M_4'$ and the statement of the theorem follows.

The immediate corollary of theorem 3.13 is that progress optimality on square profiles results in progress optimality on all memory profiles.

**Corollary 3.14.** If an algorithm is optimally progressing on square profiles, then it is optimally progressing on all memory profiles.

Other reincarnations of the argument of theorem 3.13 are used in chapter 9 in the analysis of the LRU paging algorithm, and also in section 10.1 to show that competitive optimality of cache-oblivious algorithms on square profiles transfers to competitive optimality on all profiles.

![Figure 3.2: The usable profile beneath each square profile.](image)

**Theorem 3.15.** Let $\rho$ be square additive. For every square memory profile $M$, there exists a usable memory profile $U$ below $M$ such that, $\rho(U) = \Theta(\rho(M))$.

**Proof.** Let $M = \square_M \parallel \square_M \parallel \ldots \parallel \square_M$ be any square profile. We construct a usable profile $U$ as follows. We allow $\square_{U_1} = \square_M$.
(i) For each \( i \geq 2 \) if \( M_i \leq M_{i-1} \), we let \( \Box U_i = \Box M_i \).

(ii) Otherwise if \( M_i > M_{i-1} \), we let \( U \) grow by 1 block at a time until it reaches \( M_i \). Afterwards, we allow \( U(t) = M_i \) until the boundary of \( \Box M_i \) ends.

See fig. 3.2 for an illustration. It is obvious that \( U \preceq M \), so by monotonicity of \( \rho \), we have that \( \rho(U) \leq \rho(M) \).

We now argue that \( \rho(U) = \Omega(\rho(M)) \). To exhibit this, we show that there exist mutually disjoint squares \( \Box W_i \), that all fit below \( U \) and for each \( i \), \( \Box W_i \) is at most 2 times shorter than \( \Box M_i \). Since each \( \Box W_i \) fits below \( U \), we have that \( W \preceq U \) where \( W = \Box W_1 \parallel \Box W_2 \parallel \ldots \parallel W_k \). On the other hand, since \( \rho \) is square-additive, \( \rho \) is bounded by a polynomial and thus \( \rho(\Box W_i) = \Theta(\rho(\Box M_i)) \). Square-additivity of \( \rho \) also means that \( \rho(W) = \Theta(\rho(M)) \). Since \( \rho(W) \leq \rho(U) \) the statement follows.

It remains to show that such \( \Box W_i \) exist for each \( i \). For each \( i \), if \( \Box U_i = \Box M_i \) (as in case (i) above), we allow \( \Box W_i = \Box M_i \). Otherwise (as in case (ii) above), we let \( \Box W_i \) be a square that is grown from the rightmost point of \( \Box M_i \) diagonally to left until it touches \( U \), see fig. 3.2. Note that because \( U \) increases linearly at the beginning of \( \Box M_i \) until it reaches \( M_i \), the point of \( \Box W_i \) intersecting \( U \) is always on or above the diagonal of \( \Box M_i \). Therefore, the height of \( W_i \) is at least \( 1/2 \) the height of \( M_i \).

\[ \square \]

3.4 Progress Optimality vs. Competitive Optimality

In this section, we prove that if an algorithm is optimally progressing, then it is optimally cache-adaptive (lemma 3.16) in the CA model. However, it remains an open question whether all competitively optimal algorithms are also optimally progressing in the CA model.

**Lemma 3.16.** If an algorithm \( A \) is optimally progressing, then it is optimally cache adaptive.

**Proof.** Let \( N \) be a sufficiently large input size. Suppose \( M \) is an \( N \)-fitting profile for some other algorithm \( E \). With some (unknown and possibly \( \Omega(1) \)) speed augmentation \( c \), \( A \) can solve all problems of size \( N \) on \( M' \), the inner square profile of \( M \). Let \( M'_c \) be the \( c \)-speed augmented version of \( M' \), where \( c \) is chosen to be as small as possible, so that \( M'_c \) is \( N \)-fitting for \( A \).

\( M'_c \) replaces each square in \( M' \) with \( c \) squares of the same height, therefore by square-additivity of \( \rho \) we have that

\[ \rho(M'_c) = c\Theta(\rho(M')). \]
Since $A$ is optimally progressing and $M_c'$ is $N$-fitting $\rho(M_c') = O(R(N))$. We have
\[
\rho(M_c') = c\Theta(\rho(M')) = O(R(N))
\]
\[
= c\Theta(\rho(M)) \quad \text{by theorem 3.13.}
\]

On the other hand, since $M$ is $N$-fitting for $\mathcal{E}$, we have that $\rho(M) \geq R(N)$, so it must be the case that $c = O(1)$.

We have that with $c = O(1)$ speed augmentation, $A$ can solve all problems of size $N$ in $M'$. Because $M$ is always above its inner square profile $M'$, and $A$ is memory-monotone (ref. to definition 2.3), $A$ on $M$ is no more than $f = O(1)$ times slower than $A$ on $M'$. Therefore, with $cf = O(1)$ augmentation, $A$ can solve all problems of size $N$ on $M$ and hence has a running time no worse than $\mathcal{E}$ on $M$.

3.5 Recursions in the Cache-Adaptive Model

We analyze different types of recursive algorithms in the cache-adaptive model. Recursive cache-oblivious algorithms have base cases of constant size. In contrast, their I/O complexity is expressed by a recurrence, where the base case is a function of $M$ or $B$.

The recurrence “bottoms out” at nodes in the recursion tree with input size at most $M$. This is because once a subproblem is brought fully into memory, subsequent recursive calls do not incur I/Os. We refer to such nodes as bottomed-out nodes; see fig. 3.3. The number of block transfers needed to complete a bottomed-out node is usually linear in the input size of the node.

Bottomed out nodes in a recursion tree in DAM are at the same depth (as long as the tree has a regular structure) since $M$ is fixed. In contrast, in the CA model, where the
memory size changes over time, the height of bottomed-out nodes can vary; see fig. 3.4. The running time of a recursive algorithm in the CA model is influenced by the height of these bottomed-out nodes in different periods of time.

When a cache-efficient recursive algorithm is not making recursive calls, the work it does must be I/O efficient. We refer to this work as a **linear scan**. Note that under our definition, a linear scan need not access a sequence of consecutive elements, as in a classical linear scan. However, it must be efficient—accessing $\Omega(B)$ useful locations on average, plus $O(1)$ additional I/Os.

**Definition 3.17.** *We say that an algorithm $L$ is a linear scan of size $\ell$ if it accesses $\ell$ distinct locations, it performs $\Theta(\ell)$ memory references, and its I/O complexity is $\Theta(1+\ell/B)$.***

This definition captures a wide variety of efficient cache-oblivious behaviors. Note that, in the definition, a linear scan may not access every element of its input (e.g., a search for a specific item in an array), it may not access the pages in sequential order (e.g., matrix transpose), and the order of accesses can be data-dependent (e.g., the merge operation from merge-sort).

Note further that the definition of a scan depends implicitly on the memory profile. For example, a matrix transpose is a linear scan only when $m$ is $h$-tall.

**Definition 3.18.** *Algorithm $A$ has space complexity $f(N)$ if for all problems of size $N$, the number of distinct memory locations accessed by $A$ while processing the input is $\Theta(f(N))$.***
Chapter 4

Optimality Criteria for Recursive Cache-Oblivious Algorithms

In this chapter, we study optimality criteria for recursive cache-oblivious algorithms. In section 4.1 we define several classes of recursive cache-oblivious algorithms. Later, we characterize the optimality criteria for these classes of recursive algorithms in the cache-adaptive model.

We prove algorithms are optimally (or sub-optimally) progressing by analyzing their performance on their worst-case profiles, as defined below.

Definition 4.1. Algorithm A’s worst-case profile for inputs of size \(N\) among all profiles that are \(\lambda\)-tall is

\[
W_{A,N,\lambda} = \text{argmax}\{\rho(M) \mid M \text{ is an } N\text{-fitting, } \lambda\text{-tall profile}\}.
\]

When \(\lambda\) is omitted, we assume that \(\lambda\) equals the tall-cache requirement for \(A\), \(H(B)\),

\[
W_{A,N} = W_{A,N,H} = \text{argmax}\{\rho(M) \mid M \text{ is an } N\text{-fitting, } H\text{-tall profile}\}.
\]

To bound the progress of \(W_{A,N}\), we bound the progress of its inner square profile. Theorem 3.13 shows that they have asymptotically the same progress.

The following observation shows that \(\rho(W_{A,N})\) can’t be too small.

Observation 4.2. If \(A\) has linear space complexity, then \(\rho(W_{A,N}) = \Omega(\rho(\square_N))\), because we can construct an \(N\)-fitting profile that contains at least one square of size \(\Theta(N)\).

4.1 Classes of Recursive Cache-Oblivious Algorithms

We now introduce \((a, b, c)\)-regular algorithms. This general class comprises any algorithm that recursively divides a problem of size \(N\) into \(a\) subproblems of size \(N/b\), and then does
a linear scan of size $\Theta(N^c)$.

**Definition 4.3.** Let $a \geq 1/b$, $0 < b < 1$, and $0 \leq c \leq 1$ be constants. An algorithm is \((a, b, c)\)-regular if, for inputs of sufficiently large size $N$, it makes

(i) exactly $a$ recursive calls on subproblems of size $bN$, and

(ii) performs $\Theta(1)$ linear scans before, in between or after recursive calls. The size of the biggest linear scan is $\Theta(N^c)$.

**Definition 4.4.** When the size of a linear scan in an invocation of an algorithm is $\leq B$, we refer to it as an **overhead** reference. An overhead reference costs $\Theta(1)$ I/Os.

**Observation 4.5.** As the recursion of a recursive algorithm deepens, the size of input decreases, and linear scans become smaller. Some of these linear scans might turn into overhead references at the deeper levels of the recursion.

Also, all linear scans of size $\Theta(N^0) = \Theta(1)$ are overhead references.

**Definition 4.6.** Let $f \geq 1$, $0 \leq c \leq 1$, $a_i > 0$, and $0 < b_i < 1$ be constants for $i = 1, 2, \ldots, f$. An algorithm $A$ is a **generalized regular (GR) algorithm** if, on an input of size $N$, makes

(i) exactly $a_i$ recursive calls to subproblems of size $b_iN$,

(ii) performs $\Theta(1)$ linear scans before, in between or after recursive calls. The size of the biggest linear scan is $\Theta(N^c)$.

We make use of the following notation throughout multiple proofs.

**Notation 4.7.** The linear scans of an invocation of a GR algorithm on an input of size $x$ can be categorized as:

- $L_1(x)$: $d_1 = \Theta(1)$ linear scans before any of the subcalls.

- $L_{2u}(x)$: $d_{2u} = \Theta(1)$ linear scans between subcall $u$ and subcall $u + 1$.

- $L_3(x)$: $d_3 = \Theta(1)$ linear scans at the end of all subcalls.

The following tweak of the definition 4.6 allows us to model classes of algorithms with multiple recursive subroutines that call each other. Examples of such algorithms include the cache-oblivious longest-common-subsequence (LCS) algorithm [21], the cache-oblivious dynamic programming edit-distance algorithm [21], and the Jacobi Multipass Filter algorithm [47].
Definition 4.8. Let $0 \leq c_j \leq 1$ and $f_j \geq 1$ be constants for $j = 1, \ldots, e$. Also let $a_{ji} > 0$, and $0 < b_{ji} \leq 1$ be constants for $j = 1, \ldots, e$, and $i = 1, \ldots, f_j$. Algorithms $A_1, \ldots, A_e$ are **generalized compositional regular (GCR) algorithms** if, for all $i$, $A_j$ on an input of size $N$ makes

(i) exactly $a_{ji}$ calls to algorithm $A_{ji}$ on subproblems of size $b_{ji}N$. Algorithm $A_{ji}$ is one of $A_1, \ldots, A_e$.

(ii) performs $\Theta(1)$ linear scans before, in between or after its calls. The size of the biggest linear scan is $\Theta(N^{c_j})$.

Algorithms $A_1, \ldots, A_e$ are **perfect generalized compositional regular (PGCR) algorithms**, if, for every $j$, the size of all of $A_j$’s linear scans is $\Theta(N^{c_j})$.

4.2 Structure of $N$-Fitting Square Profiles For Recursive Algorithms

The worst-case profile $W_{A,N}$, or its inner square profile, does not have to respect the recursive structure of $A$. For example, squares can cross recursive boundaries, cover multiple recursive invocations, span multiple linear scans, etc. Any analysis based solely on the recursive structure of the algorithm must handle the fact that the profile may not nicely line up with the algorithm.

To solve this problem, we first establish a mapping from squares of any $N$-fitting square profile to recursive calls and linear scans performed by $A$.

Definition 4.9. When $A$ executes on a square profile $M(t)$, we say a square $S$ of $M$ **overlaps** a linear scan $L$ if at least one memory reference of $L$ is served during $S$. Similarly, we say $S$ **encompasses** $A$’s execution on a subproblem if every memory reference $A$ makes while solving the subproblem is served during $S$. Finally, we say $S$ **contains** an overhead reference $R$ if at least half of the references of $R$ are served during $S$.

Definition 4.10. Let $A_1, \ldots, A_e$ be generalized compositional regular (GCR) algorithms all with linear space complexity. We say that a square profile $M$ of length $\ell$ is **$N$-chargeable with respect to $A_j$**, if every square $S$ of $M$ satisfies at least one of the following three properties when $M$ is considered with respect to $A_j$’s execution on any problem instance of size $N$ that takes exactly $\ell$ steps to process.

(i) $S$ encompasses an execution of any of $A_1, \ldots, A_e$ on a subproblem of size $\Theta(|S|)$. 

26
(ii) $S$ overlaps a linear scan of size $\Omega(|S|)$.

(iii) $S$ contains $\Theta(|S|/B)$ overhead references.

The progress of an $N$-chargeable square profile with respect to $A_j$ can be charged to the recursive entities in $A_j$’s execution on problem instances of size $N$.

The following fundamental lemma shows that $N$-fitting square profiles of linear space complexity regular algorithms (definition 4.3, definition 4.6 and definition 4.8) are $N$-chargeable. Thus, we can use a charging scheme to bound the progress of $W_{A_j,N}$ by charging the squares of $W_{A_j,N}$ to recursive entities in $A_j$’s execution on problem instances of size $N$; see theorem 4.14.

**Lemma 4.11.** Let $c$ be a constant and let $A_1, \ldots, A_c$ be perfect generalized compositional regular (PCGR) algorithms, all with linear space complexity. Then every $N$-fitting square profile for $A_j$ is $N$-chargeable with respect to $A_j$.

*Proof.* Let $M$ be an $N$-fitting profile for $A_j$ and let $S$ be a square of $M$ and let $\sigma$ be the sequence of memory references generated while solving a problem instance of size $N$ for which $M$ is $I$-fitting. We prove that $S$ must have one of the three properties in definition 4.10 with respect to $\sigma$.

Let $N'$ be such that every $A_1, \ldots, A_c$ can solve problems of size less than or equal to $N'$ using at most $|S|/3B$ I/Os and $|S|$ memory. Since every $A_1, \ldots, A_c$ has linear space complexity, $N' = \Theta(|S|)$.

Let $b = \min\{b_{ji}\} = \Theta(1)$. Note that every root-to-leaf path of the recursion tree must contain a subproblem whose size is in the range $[bN', N']$, so we can expand the recursion tree to subproblems of size between $bN'$ and $N'$. Let $E_1, \ldots, E_t$ be the leaves of this partially expanded recursion tree, so that each $E_i$ corresponds to an execution of an $A_i$ on a problem of size between $bN'$ and $N'$.

**General properties of linear scans** As before, we use notation 4.7 to describe different types of linear scans in the recursive structure of $A_j$. Let $\Phi$ be any subsequence of memory references that does not contain a complete execution of any $A_i$. Thus $\Phi$ can contain only

- references generated by linear scans performed at the end of an execution of one of the $A_i$s ($L_3$-type linear scans),
- references generated by an $L_{2a}$-type linear scan between two recursive calls,
- references generated by linear scans performed at the beginning of an invocation of one of the $A_i$s ($L_1$-type linear scans).
If $\Phi$ lies between two complete executions, then it may contain some number of $L_3$-type scans, followed by an $L_2$-type scan, followed by some number of $L_1$-type scans. If $\Phi$ consists of references in $\sigma$ before the first complete execution of any $A_i$, then it contain references from only $L_1$-type scans. If $\Phi$ follows the last complete execution of any $A_i$ in $\sigma$, then it will contain references from only $L_3$-type scans.

**Property (i)** If the square $S$ encompasses an execution of any of $A_1, \ldots, A_e$ on a problem of size at least $bN'$, then we are done, since $bN' = \Theta(N') = \Theta(|S|)$.

**Properties (ii) and (iii)** Suppose $S$ does not encompass an invocation of $A_j$ on a subproblem of size at least $bN'$. We show that $S$ must either satisfy property (ii) or property (iii). In this case, $S$ can intersect at most two of the leaves $E_i$ and $E_{i+1}$ of our partially expanded recursion tree (one at the beginning and one at the end). Furthermore, by the choice of $N'$, these executions can occupy at most 2/3rds of the I/Os of $S$. Thus at least 1/3 the I/Os of $S$ must be a contiguous sequence of memory references that does not contain a complete execution of any $A_i$. Call this subsequence $\Phi$.

Let $Z_1, Z_2, Z_3$ be the set of linear scans of type $L_1$, $L_2u$, and $L_3$, respectively, in $\Phi$. Since $S$ does not encompass a subproblem, the linear scans in $Z_1$ all belong to only one sequence of $L_1$-type slide-down moves on the recursion tree. Similarly, the linear scans in $Z_3$ all belong to only one sequence of $L_3$-type climb-up moves in the recursion tree.

Let $I(.)$ denote the I/O complexity of a set of linear scans and allow

$$z = \max\{I(Z_1), I(Z_2), I(Z_3)\}.$$  

Since at least 1/3 of I/Os in $S$ are overlapping linear scans, we have that $z \geq |S|/9B$. There are three cases to be considered.

**Case of $z = I(Z_2)$** In this case $Z_2$ is comprised of linear scans in only one $L_2u$ set. Since $A_1, \ldots, A_e$ are all perfect, all linear scans in $L_{2u}$ are of size $\Theta(N^y)$ for some constant $y$.

If $y = 0$, then all scans in $L_{2u}$ are overhead references and cost $\Theta(1)$ I/Os. Since, there are only $d_{2u} = \Theta(1)$ linear scans in $L_{2u}$, we deduce that $z = I(Z_2) = \Theta(1)$. Since $z \geq |S|/9B$, $S$ contains $d_{2u} = \Theta(|S|/B)$ overhead references and thus satisfies property (iii).

If otherwise $y > 0$, let $E$ be the biggest linear scan in $L_{2u}$. Because there are only $d_{2u} = \Theta(1)$ linear scans in $L_{2u}$, then $E$ is a linear scan of size $\Omega(z) = \Omega(|S|)$. Thus, $S$ would satisfy property (ii).
Case of $z = \mathcal{I}(Z_3)$  Here, $Z_3$ is comprised of linear scans of type $L_3$ from several invocations of (possibly) different $A_j$ algorithms. Let $L_3^j$ denote the set of all $L_3$ type linear scans in $Z_3$ that are executed in algorithm $A_j$’s invocations.

Let $L_3^m$ be the set among all $L_3^3$s with the biggest I/O cost and let $A_m$ be the algorithm that produced these scans. Since there are $e = \Theta(1)$ compositional algorithms, we have that $\mathcal{I}(L_3^m) \geq \mathcal{I}(Z_3)/e$.

Let $x_1 \leq \cdots \leq x_k$ be the problem sizes solved by each of the invocations of $A_m$ that generated one of the $L_3$-type linear scans in $Z_3$. Let $L_3^m(x_i)$ denote the linear scans generated by the invocation of $A_m$ on problem of size $x_m$. Let $q = \max\{b_{ji}\}$ (note that $0 < q < 1$). Note that, since $A_1, \ldots, A_e$ are PGCR algorithms and the sequence of linear scans in $L_3$ come from invocations of $A_1, \ldots, A_j$ in exactly one sequence of climb-up moves, $x_i \leq q^{k-i}x_k$ for all $i$. Also, we have that

$$L_3^m = L_3^m(x_1) \cup L_3^m(x_2) \cdots \cup L_3^m(x_k).$$

If $k = 1$, then the analysis in case ($z = \mathcal{I}(Z_2)$) shows that $S$ either satisfies property (ii) or property (iii).

So, we assume that $k > 1$. Since $A_m$ is perfect, all linear scans in each $L_3^m(x_i)$ are all of size $\Theta(x_i^{cm})$ for a constant $c_m$.

If $c_m = 0$, then all linear scans in $L_3^m$ are overhead references. Since each overhead reference costs $\Theta(1)$ I/Os and $\mathcal{I}(L_3^m) = \Omega(|S|/B)$ there must be $\Omega(|S|/B)$ overhead references in $L_3^m$ and $S$ satisfies property (iii).

Otherwise, assume that $c_m > 0$. By definition, each invocation of $A_m$ can only perform a constant number of $L_3$-type linear scans, so we can write $\mathcal{I}(L_3^m)$ as:

$$\mathcal{I}(L_3^m) = \mathcal{I}(L_3^m(x_1)) + \mathcal{I}(L_3^m(x_2)) + \cdots + \mathcal{I}(L_3^m(x_k))$$

$$= \Theta\left(1 + \frac{x_1^{cm}}{B}\right) + \Theta\left(1 + \frac{x_2^{cm}}{B}\right) + \cdots + \Theta\left(1 + \frac{x_k^{cm}}{B}\right).$$

Let $v$ be the biggest index such that the size of the biggest linear scan in $L_3^m(t)$ is $\leq B$ for each $1 \leq t \leq v$. We compare

$$\sigma_1 = \mathcal{I}(L_3^m(x_1)) + \cdots \mathcal{I}(L_3^m(x_v))$$

and

$$\sigma_2 = \mathcal{I}(L_3^m(x_{v+1})) + \cdots \mathcal{I}(L_3^m(x_k)).$$

(a) If $\sigma_1 \geq \sigma_2$, we argue that $S$ must satisfy property (iii). By definition of $v$, we have that all the linear scans in $\sigma_1$ are overhead references and cost $\Theta(1)$ I/Os. Since we have that $\sigma_1 \geq \mathcal{I}(L_3^m)/2 = \Omega(|S|/B)$, there must be $\Omega(|S|/B)$ overhead references in $L_3^m(x_1) \cup L_3^m(x_2) \cdots \cup L_3^m(x_v)$.
(b) If $\sigma_2 > \sigma_1$, we argue that $S$ must satisfy property (ii). In this case, we show that $\mathcal{I}(L^3_m(x_k)) = \Omega(|S|/B)$ and so $L^3_m(x_k)$ is a linear scan of size $\Omega(|S|)$ overlapped by $S$.

By definition of $v$, we have that the biggest linear scan in $L^3_m(x_{v+1})$ is of size $> B$. Because $x_{i+1} \geq x_i/q$ for each $i$, we have that the biggest linear scan in $L^3_m(x_t)$ has size bigger than $\Omega(B)$ for $v+1 \leq t \leq k$. Hence, we can write

$$\sigma_2 = \mathcal{I}(L^3_m(x_{v+1})) + \cdots + \mathcal{I}(L^3_m(x_k)) = \Theta\left(\frac{x_{v+1}^{cm}}{B}\right) + \cdots + \Theta\left(\frac{x_k^{cm}}{B}\right).$$

Consider the geometric series $\Psi = (x_k)^{cm} + (qx_k)^{cm} + \cdots + (q^{k-v}x_k)^{cm}$ with constant coefficient $(q)^{cm}$. Note that $\Psi \geq \sum_{t=v+1}^{k} x_t^{cm}$, because $x_i \leq q^{k-i}x_k$. We have that

$$|L^3_m(x_k)| = \Theta(x_k^{cm}) = \Omega(\Psi) = \Omega\left(\sum_{t=v+1}^{k} x_t^{cm}\right) = \Omega(B\sigma_2) = \Omega(|S|).$$

**Case of $z = \mathcal{I}(Z_1)$** The analysis in this case is identical to the case of $z = \mathcal{I}(Z_3)$.

We have established that each square $S$ satisfies one of the three properties and the proof is complete. \(\square\)

**Definition 4.12.** Let $A_1, \ldots, A_e$ be a set of generalized compositional regular algorithms. Let $S$ be an overhead-containing square of an $N$-fitting profile for $A_j$. For each overhead reference $r$ in $A_j$, we let $\text{caller}(r)$ be the call of any of $A_1, \ldots, A_e$ that created the reference $r$, and $\text{rank}(r)$ be the input size of caller($r$). Also we define

\[ E(S) = \{r | r \text{ is an overhead reference contained in } S\}, \]

\[ X(S) = \text{caller} \left( \arg\max_{r \in E(S)} \text{rank}(r) \right). \]

$X(S)$ is the highest rank invocation of any of $A_1, \ldots, A_e$ that produced any of the overhead references in $S$.

The following lemma bounds the size of squares of an $N$-chargeable profile which do not satisfy property (i) nor property (ii) in definition 4.10.

**Lemma 4.13.** Let $A_1, \ldots, A_e$ be a set of generalized compositional regular algorithms all with linear space complexity and let $q = \max\{b_{ji}\}$. Let $M$ be an $N$-chargeable profile with respect to $A_j$. Each square of $M$ that does not satisfy property (i) nor property (ii) in definition 4.10 has size $O\left(B \log_{1/q} X(S)\right)$. 

30
Proof. Let $S$ be a square of $M$ that does not satisfy property (i) nor property (ii). Because $M$ is $N$-chargeable with respect to $A_j$, we have that $S$ must contain $\Theta(|S|/B)$ overhead references.

Let $b = \min\{b_{ji}\} = \Theta(1)$, and expand the recursion to subproblems of size between $bN'$ and $N'$, where $N'$ is chosen so that every $A_1, \ldots, A_e$ can solve problems of size less than or equal to $N'$ using at most $|S|/3B$ I/Os and $|S|$ memory. Since every $A_1, \ldots, A_e$ has linear space complexity, $N' = \Theta(|S|)$. $S$ does not satisfy property (i), so it can not encompass a subproblem of size $N'$. The overhead references contained in $S$ belong to the sequence of references between at most two invocations of any of $A_1, \ldots, A_e$ on subproblems of size $N'$.

$X(S)$ is the highest rank invocation of any of $A_1, \ldots, A_e$ that produced any of the overhead references in $S$. The size of the biggest subcall for any $A_1, \ldots, A_e$ on a subproblem of size $X(S)$ is $qX(S)$. This means that between any two invocations of any of $A_1, \ldots, A_e$ on two subproblems that are overlapped by $S$ there are at most $O(\log_{1/q} X(S))$ linear scans, and consequently, overhead references.

Since $S$ contains $\Theta(|S|/B)$ overhead references, and there can be at most $O(\log_{1/q} X(S))$ overhead references contained in $S$, we have we have that $|S_1| = O(B\log_{1/q} X(S))$. \qed

# 4.3 Optimality Criteria For Generalized Compositional Regular Algorithms

In this section, we present theorems on the optimality criteria for generalized compositional regular (GCR) algorithms.

Theorem 4.14’s statement is complicated not because memory is changing size, but because it covers a wide variety of recursive forms, including algorithms that have several mutually recursive functions.

**Theorem 4.14.** Let $0 \leq c_j \leq 1$ and $f_j \geq 1$ be constants for $j = 1, \ldots, e$. Also let $a_{ji} > 0$, and $0 < b_{ji} < 1$ be constants for $j = 1, \ldots, e$, and $i = 1, \ldots, f_j$. Suppose $A_1, \ldots, A_e$ are generalized compositional regular algorithms all with linear space complexity, tall-cache requirement $H(B)$, and progress bound $\rho$.

Let $b = \max\{b_{ji}\}$ and $\lambda \geq H(B)$ be constants. Then there exist functions $T_1, \ldots, T_e; \mathcal{U}_1, \ldots, \mathcal{U}_e; \mathcal{V}_1, \ldots, \mathcal{V}_e$ such that the progress of the worst-case $\lambda$-tall profile for $A_j$, $\rho(W_{A_j,N,\lambda})$, is $O(T_j(N) + \mathcal{U}_j(N) + \mathcal{V}_j(N))$ and the $T_j, \mathcal{U}_j$ and $\mathcal{V}_j$ satisfy the recurrences
\[ T_j(N) = \begin{cases} \max \left\{ \rho(\Box N), \sum_{i=1}^{f_j} a_{ji} T_{ji}(b_{ji} N) \right\} & \text{if } \lambda < N \\ \Theta(\rho(\Box \lambda)) & \text{if } N \geq \lambda; \end{cases} \]

\[ U_j(N) = \begin{cases} \Theta(\rho(\Box N_{c_j})) + \sum_{i=1}^{f_j} a_{ji} U_{ji}(b_{ji} N) & \text{if } N = \Omega(\lambda) \text{ and } N_{c_j} = \Omega(\lambda) \\ \sum_{i=1}^{f_j} a_{ji} U_{ji}(b_{ji} N) & \text{if } N = \Omega(\lambda) \text{ and } N_{c_j} \neq \Omega(\lambda) \\ 0 & \text{if } N \neq \Omega(\lambda); \end{cases} \]

\[ V_j(N) = \begin{cases} \sum_{i=1}^{f_j} a_{ji} V_{ji}(b_{ji} N) & \text{if } B \log_{1/b} N = \Omega(\lambda) \text{ and } N_{c_j} > B \\ \Theta(\rho(\Box B \log_{1/b} N)) + \sum_{i=1}^{f_j} a_{ji} V_{ji}(b_{ji} N) & \text{if } B \log_{1/b} N = \Omega(\lambda) \text{ and } N_{c_j} \leq B \\ 0 & \text{if } B \log_{1/b} N \neq \Omega(\lambda). \end{cases} \]

where \( T_{ji}, U_{ji} \) and \( V_{ji} \) are one of \( T_1, \ldots, T_e; U_1, \ldots, U_e; V_1, \ldots, V_e \) depending on the structure of \( A_j \).

Theorem 4.15 tells us when the bound given in theorem 4.14 is tight.

**Theorem 4.15.** Suppose \( A_1, \ldots, A_e \) are generalized compositional regular algorithms with linear space complexity, tall-cache requirement \( H(B) \), and progress bound \( \rho \). Let \( \lambda \) be equal to \( \max\{H(B), (B \log_{1/b} B)^{1+\epsilon}\} \). When for all \( j, c_j = 1 \), we have that \( V_j(N) = 0 \) and \( \rho(W_{A_j,N,\lambda}) = \Theta(T_j(N) + U_j(N)) \).

Theorems 4.14 and 4.15 give us a characterization for when GCR algorithms are optimally progressing with respect to progress function \( \rho \) and progress requirement function \( R \) as follows:

- Solve the recurrence in theorem 4.14 to obtain \( T_j(N) + U_j(N) + V_j(N) \), an upperbound on the progress that any algorithm can make on \( A_j \)'s worst-case \( N \)-fitting profile.

- If \( T_j(N) + U_j(N) + V_j(N) = O(R(N)) \), then \( A_j \) is optimally progressing and, by lemma 3.16, optimally cache adaptive.

- If \( T_j(N) + U_j(N) + V_j(N) \neq O(R(N)) \) then, \( A_j \) is \( O((T_j(N) + U_j(N) + V_j(N))/R(N)) \) away from being optimally progressing and cache-adaptive. In this case, if all \( c_j = 1 \), then \( A_j \) is \( \Theta((T_j(N) + U_j(N))/R(N)) \) away from being optimally progressing.
Proof of Theorem 4.14

Lemma 4.11 shows that all $N$-fitting profiles for perfect generalized compositional regular algorithms are $N$-chargeable. However, algorithms $A_1, \ldots, A_j$ are not necessary perfect (according to definition 4.8), so lemma 4.11 does not apply to them.

We modify $A_1, \ldots, A_j$ to get padded algorithms $A'_1, \ldots, A'_e$ and exhibit that the padded algorithms are PGCR algorithms. Each $A'_j$ operates exactly in the same way as $A_j$, except that on in each invocation of size $Y$, $A'_j$ pads all linear scans of $A_j$ to be as big as the biggest linear scan in $A_j$’s invocation of size $Y$, $\Theta(Yc_j)$. The padding operation can be done by scanning an auxiliary array of appropriate size.

We argue that $\rho(W_{A'_j,N,\lambda}) \leq \rho(W_{A'_j,N,\lambda})$. We take $W_{A'_j,N,\lambda}$ as a profile and modify it to get a profile $\Phi_j$ and show that $\Phi_j$ is $N$-fitting for $A'_j$. Consider $W_{A'_j,N,\lambda}$ and in a bottom-up manner extend squares of $W_{A'_j,N,\lambda}$ so that these squares are big enough to serve all padded linear scans in $A'_j$. Since $W_{A'_j,N,\lambda}$ is $N$-fitting for $A_j$, by definition $Z'_j$ is $N$-fitting for $A'_j$. By definition 3.5 we have that $W_{A'_j,N,\lambda} \prec \Phi_j$. Hence, by definition 3.6

$$\rho(W_{A'_j,N,\lambda}) \leq \rho(\Phi_j) \leq \rho(W_{A'_j,N,\lambda}).$$ (4.1)

Let $M_j$ be any $\lambda$-tall $N$-fitting profile for $A'$. We bound the progress of $M_j$ thus bounding $\rho(W_{A'_j,N,\lambda})$ from above. Lemma 4.11 shows that $M_j$ is $N$-chargeable.

If $N \leq \lambda$, because $A'_j$ is PCGR and has linear space complexity, if one unrolls the recursion of $A'_j$ a constant number of times, whole executions of subproblems can be completed inside a single square of size $\lambda$. It follows that $M_j$ must consist of $\Theta(1)$ squares of size $\lambda$, so $\rho(M_j) = \Theta(\rho(\square_\lambda)) = \Theta(T_j(N))$ for any $j = 1, \ldots, e$.

Now, let $N > \lambda$. Because $M_j$ is $N$-chargeable, every square $S$ in $M_j$ satisfies one of the three properties in definition 4.10.

Charging the subproblem-encompassing squares First, we charge the progress of each subproblem-encompassing square to the covered subproblem. When a square $S$ is charged to a subproblem $Z$, all subproblems of $Z$ are encompassed by $S$. Because $A'_j$ has linear space complexity, $S$ has size $\Theta(|Z|)$. Therefore, the progress of all squares charged to subproblems is bounded by $\Theta(T_j(N))$ where $T_j(N)$ satisfies the recurrence

$$T_j(N) = \begin{cases} 
\max \left\{ \rho(\square_N), \sum_{i=1}^{f_j} a_{ji} T_{ji}(b_{ji}N) \right\} & \text{if } \lambda < N \\
\Theta(\rho(\square_\lambda)) & \text{if } N \leq \lambda.
\end{cases}$$
Charging the linear-scan-overlapping squares

If a square $S$ overlaps a linear scan $L$ of size $\Omega(|S|)$ executed by the top-level invocation of $A'_j$ and $L$ is the biggest linear scan overlapped by $S$, we charge it to $L$. At the top-level invocation of $A'_j$, all linear scans are of size $\Theta(N)$.

We develop a recursive relation $U_j(N)$ that bounds the total progress of all linear-scan-overlapping squares for an invocation of $A'_j$ on a problem instance of size $N$.

Let $N_0$ be the size of input in an invocation for $A'_j$. If $N_0 \neq \Omega(\lambda)$, then because $M_j$ is $\lambda$-tall, no square in $M_j$ can be charged to a linear scan executed in any part of the subproblem $A'_j(N_0)$, as squares are much bigger than these linear scans. Therefore, $U_j(N_0) = 0$.

Now consider an input size of $N_1 = \Omega(\lambda)$ for $A'_j$. If $N_1 \neq \Omega(\lambda)$, then because $M_j$ is $\lambda$-tall, no square in $M_j$ can be charged to a linear scan executed in the top level invocation of $A'_j(N_1)$. However, other subcalls of $A'_j$ might execute linear scans that are large enough. Therefore, $U_j(N_1) = \sum_{i=1}^{f_j} a_{ji} U_{ji}(b_{ji} N_1)$.

For larger $N$, multiple squares may have their progress charged to a single linear scan of size $|L|$, but all but at most two of those squares will be contained entirely within the linear scan. Thus, the total size of all the squares charged to a single linear scan of size $|L|$ will be $\Theta(|L|)$. Suppose $S_1, \ldots, S_k$ are the squares charged to $L$. Since $\rho(\square_X) = \Omega(X)$, we must have that $\sum \rho(S_i) = O(\rho(\square \sum |S_i|)) = O(\rho(\square |L|))$. Thus the progress of all the squares charged to linear scans executed by the top-level invocation of $A'_j$ on a problem instance of size $N$ can be upper-bounded by $\Theta(\rho(\square N^{c_j}))$.

Therefore, the progress of all squares charged to linear scans is upperbounded by $U_j(N)$ where

$$U_j(N) = \begin{cases} 
\Theta(\rho(\square N^{c_j})) + \sum_{i=1}^{f_j} a_{ji} U_{ji}(b_{ji} N) & \text{if } N = \Omega(\lambda) \text{ and } N^{c_j} = \Omega(\lambda) \\
\sum_{i=1}^{f_j} a_{ji} U_{ji}(b_{ji} N) & \text{if } N = \Omega(\lambda) \text{ and } N^{c_j} \neq \Omega(\lambda) \\
0 & \text{if } N \neq \Omega(\lambda).
\end{cases}$$

Charging the overhead-containing squares

Finally, we charge each overhead-containing square $S_1$ to the recursive call that corresponds to the subproblem $X(S_1)$, the highest rank subproblem that produced any of the overhead references in $S_1$ (see definition 4.12). By lemma 4.13, the size of each overhead-containing square, $S_1$, of $M_j$ that is not subproblem-encompassing nor linear-scan-overlapping is $O(B \log_{1/b} X(S_1))$.

If $c_j = 0$, then all linear scans of $A'_j$ are overhead references. Otherwise, $A'_j$’s linear scan convert to overhead references when $N^{c_j} \leq B$. 

34
We develop a recursive relation $V_j(N)$ that bounds the total progress of all overhead-containing squares for an invocation of $A'_j$ on a problem instance of size $N$. At the top-level invocation of $A'_j$ of size $N$, we only account for overhead-containing squares whose highest rank overhead are produced at the current invocation. Hence, their size is $O(B \log \frac{1}{b} N)$.

Let $N_3$ be the size of input in an invocation for $A'_j$. If $B \log \frac{1}{b} N_3 \neq \Omega(\lambda)$, then because $M_j$ is $\lambda$-tall, no square of $M_j$ can be charged to the recursive call $A'_j(N_3)$, because squares are much bigger than a series of overhead references whose highest ranked reference is executed in $A'_j(N_3)$. Therefore, $V_j(N_3) = 0$.

Now consider $N_4$ to be an input size for $A'_j$, such that $B \log \frac{1}{b} N_4 = \Omega(\lambda)$. If $N_c j \neq O(B)$, then the linear scans in the top-level invocation of $A'_j$ are not overhead references. However, other subcalls of $A'_j$ might execute overhead references. Hence, $V_j(N_4) = \sum_{i=1}^{f_j} a_{ji} V_{ji}(b_{ji} N_4)$.

Since each invocation of $A'_j$ executes $\Theta(1)$ overhead references, the progress of all overhead-containing squares that contain an overhead executed by the top-level invocation of $A'_j$ can be upper-bounded by $\Theta(\rho(\Box_{B \log \frac{1}{b} N}))$. Therefore, the progress of all overhead-containing squares is upperbounded by $V_j(N)$ where

$$V_j(N) = \begin{cases} 
\sum_{i=1}^{f_j} a_{ji} V_{ji}(b_{ji} N) & \text{if } B \log \frac{1}{b} N = \Omega(\lambda) \text{ and } N^{c_j} > B \\
\Theta(\rho(\Box_{B \log \frac{1}{b} N})) + \sum_{i=1}^{f_j} a_{ji} V_{ji}(b_{ji} N) & \text{if } B \log \frac{1}{b} N = \Omega(\lambda) \text{ and } N^{c_j} \leq B \\
0 & \text{if } B \log \frac{1}{b} N \neq \Omega(\lambda). 
\end{cases}$$

We have charged squares of every type in $M_j$. Therefore, by square-additivity of $\rho$ we have that $\rho(M_j) = O(T_j(N) + U_j(N) + V_j(N))$.

Since, $M_j$ can be any $N$-fitting profile for $A'_j$, we have shown that

$$\rho(W_{A_j, N, \lambda}) \leq \rho(W'_{A'_j, N, \lambda}) \leq O(T_j(N) + U_j(N) + V_j(N)),$$

by eq. (4.1),

which finishes the proof of the theorem.

\qed

**Proof of Theorem 4.15**

We first prove that when $c_j = 1$, and $M_j$ is $\lambda$-tall we have that for all $j$, $V_j(N) = 0$. Note that when $N^1 \leq B$, $B \log \frac{1}{b} N \leq B \log \frac{1}{b} B$. Because $\lambda \geq (B \log \frac{1}{b} B)^{1+\epsilon}$, we have that $B \log \frac{1}{b} N \neq \Omega(\lambda)$. Therefore the middle case in $V_j(N)$ never happens for any of $V_j(N)$s. Thus for all $j$, $V_j(N) = 0$.
We describe how to build an $N$-fitting profile for $A_j$, $M_j$, such that $\rho(M_j) = \Theta(T_j(N) + U_j(N))$. Consider $\Theta(T_j(N) + U_j(N))$ and among the two terms in the $\Theta$, pick the term that is bigger.

First, assume that $U_j(N)$ is bigger. We construct an $N$-fitting profile $W^j_N$ such that $\rho(W^j_N) = \Theta(U_j(N))$ as follows. Memory starts at $\lambda$. Whenever the algorithm begins a linear scan of size $L \geq 2\lambda$ with memory size $\lambda$, it will necessarily incur at least $(L - \lambda)/B = \Theta(L/B)$ cache misses, no matter how memory changes size during the linear scan. Thus we can increase the size of memory to $O(L)$ at the first page fault during the linear scan and decrease it to size $\Theta(n)$ on the last page fault during the linear scan, which will be $\Theta(L/B)$ time steps later. Thus, $W^j_N$ will contain a square of size $\Theta(L)$ for every linear scan of size $L = \Omega(\lambda)$ performed during the execution of $A_j$ on a problem of size $N$. When all linear scans of $A_j$ are finished, the profile $W^j_N$ continues at height $\lambda$ until it becomes $N$-fitting for $A_j$.

Since each $A_i$ performs linear scans of size $N^{e_i}$ on inputs of size $N$, square-additivity and monotonicity of $\rho$ entail that $\rho(W^j_N) = \Omega(U_j(N))$.

Now assume that $T_j(N)$ is bigger. We first construct a slightly different recurrence on functions $T'_j(N)$, show that $T'_j(N) = \Theta(T_j(N))$, and then construct a profile $W^j_N$ such that $\rho(W^j_N) = \Theta(T'_j(N))$.

Let $s_j(N)$ be the space complexity of $A_j$ on problems of size $N$. Let $b = \min\{b_{j_i}\}$. Let

$$T'_j(N) = \begin{cases} \max \left\{ \rho(\square_N, \sum_{i=1}^{f_j} a_{ji} T'_{ji}(b_{ji}N)) \right\} & \text{if } s_j(N) \geq 2\lambda/b \\ \Theta(\rho(\square_\lambda)) & \text{otherwise.} \end{cases}$$

Since $T_j$ and $T'_j$ differ only in a constant factor of the sizes of their base cases, $T'_j(N) = \Theta(T_j(N))$.

$T'_j(N)$ is the max among a finite number of terms. Take any term, $\Phi$, that is maximal, i.e. $T'_j(N) = \Phi$. Because $T'_j(N)$ is based on the recursive structure of $A_j$, $\Phi$ is the sum of terms $\rho(\square_{n_1}) + \cdots + \rho(\square_{n_t})$ corresponding to different subproblems of $A_1, \ldots, A_e$s of sizes $n_1, \ldots, n_t$ such that no two subproblems are included in one another. Note that, for each of these problems, their space complexity is at least $2\lambda$, by the definition of $T'_j$.

Given these non-overlapping subproblems of size $n_1, \ldots, n_t$, we construct $W^j_N$ as follows. Memory starts at size $\lambda$. Whenever the algorithm begins solving a problem of size $n_i$ with $\lambda$ memory, it will necessarily incur $\Theta(n_j/B)$ page faults because the problem’s space complexity is linear (and at least $2\lambda$ by definition of $T'_j(N)$). Thus we can increase memory to size $\Theta(n_j)$ on the first page fault during the algorithms execution on the problem, and decrease it to size $\lambda$ on the last page fault during the algorithm’s execution on the problem, which must be at least $\Theta(n_j/B)$ time steps later. After all subproblems in $\Phi$ are finished,
profile $W^j_N$ is continued at height $\lambda$ until $W^j_N$ becomes $N$-fitting for $A_j$.

By square-additivity and monotonicity of $\rho$, we have that

$$\rho(W^j_N) = \Omega(\rho(\square_{n_1}) + \cdots + \rho(\square_{n_t})) = \Omega(T'_j(N)) = \Omega(T_j(N)).$$

4.4 Optimality of Generalized Regular Algorithms

As a corollary of theorem 4.14, we get the following theorem for generalized regular (GR) algorithms.

**Theorem 4.16.** Let $0 \leq c \leq 1$ and $f \geq 1$, $a_i > 0$, and $0 < b_i < 1$ be constants for $i = 1, \ldots, f$. Suppose $A$ is a generalized regular algorithm with linear space complexity, tall-cache requirement $H(B)$, and progress bound $\rho$.

Let $b = \max\{b_i\}$ and $\lambda \geq H(B)$ be constants. Then there exist functions $T, U, V$ such that the progress of the worst-case $\lambda$-tall profile for $A$, $\rho(W_{A,N,\lambda})$, is $O(T(N) + U(N) + V(N))$ and the $T, U$ and $V$ satisfy the recurrences

$$T(N) = \begin{cases} 
\max \left\{ \rho(\square_N), \sum_{i=1}^{f} a_i T(b_i N) \right\} & \text{if } \lambda < N \\
\Theta(\rho(\square_N)) & \text{if } N \leq \lambda; \\
\end{cases}$$

$$U(N) = \begin{cases} 
\Theta(\rho(\square_{N^e})) + \sum_{i=1}^{f} a_i U(b_i N) & \text{if } N = \Omega(\lambda) \text{ and } N^c = \Omega(\lambda) \\
\sum_{i=1}^{f} a_i U(b_i N) & \text{if } N = \Omega(\lambda) \text{ and } N^c \neq \Omega(\lambda) \\
0 & \text{if } N \neq \Omega(\lambda); \\
\end{cases}$$

$$V(N) = \begin{cases} 
\sum_{i=1}^{f} a_i V(b_i N) & \text{if } B \log_{1/b} N = \Omega(\lambda) \text{ and } N^c > B \\
\Theta(\rho(\square_{B \log_{1/b} N})) + \sum_{i=1}^{f} a_i V(b_i N) & \text{if } B \log_{1/b} N = \Omega(\lambda) \text{ and } N^c \leq B \\
0 & \text{if } B \log_{1/b} N \neq \Omega(\lambda). \\
\end{cases}$$

Moreover, when $c = 0$, $U(N) = 0$.

**Proof.** The statement of the theorem is derived directly from theorem 4.14.

We show that when $c = 0$, $U(N) = 0$. Because $\lambda \geq H(B) \geq B$, when $c$ is equal to 0 we have that $N^c \neq \Omega(\lambda)$ and the first case in $U(N)$ does not happen. Hence $U(N) = 0$. \qed
The following theorem is a corollary of theorem 4.15.

**Theorem 4.17.** Suppose $A$ is a generalized regular algorithm with linear space complexity, tall-cache requirement $H(B)$, and progress bound $\rho$. Let $\lambda = \max\{H(B), (B \log_{1/b} B)^{1+\epsilon}\}$. When $c = 1$, $\forall(N) = 0$ and $\rho(W_{A,N,\lambda}) = \Theta(T(N) + U(N))$.

### 4.5 Optimality Criteria for $(a, b, c)$-Regular Algorithms

In this section we exhibit that when $\rho$ is determined, for example when $\rho(\square_X) = \Theta(X^p)$ for a constant $p$, it is easy to utilize theorem 4.16 to derive an explicit term for the progress of worst-case profiles for $(a, b, c)$-regular algorithms.

This characterization can be used to figure out the optimality criteria for $(a, b, c)$-regular algorithms in the cache-adaptive model.

We make use of the following two lemmas, which we prove at the end of this section.

**Lemma 4.18.** Let $A$ be an $(a, b, c)$-regular algorithm with linear space complexity and tall-cache requirement $H(B)$. Suppose also that, in the DAM model, $A$ is optimally progressing for a problem with progress bound $\rho(\square_X) = \Theta(N^p)$, for constant $p$. Then, $p = \log_{1/b} a$.

**Lemma 4.19.** Assume that $B \geq 4$. Pick a $\delta \in (0, 0.1)$, and let $d = 3(1 + \delta)$. If $Z$ is bigger than $(dB \log B)^{1+\delta}$, we have that $Z^{1/(1+\delta)} > B \log Z$.

**Theorem 4.20.** Let $A$ be an $(a, b, c)$-regular algorithm with linear space complexity and tall-cache requirement $H(B)$. Suppose also that, in the DAM model, $A$ is optimally progressing for a problem with progress bound $\rho(\square_X) = \Theta(N^p)$, for constant $p$. Assume that $B \geq 4$, pick an $\epsilon \in (0, 0.1)$, and let $d = 3(1 + \epsilon)$ and $\lambda = \max\{H(B), (dB \log_{1/b} B)^{1+\epsilon}\}$.

Then, $\rho(W_{A,N,\lambda})$ is bounded by $O(\mathcal{X}(N))$, where

\[
\mathcal{X}(N) = \begin{cases} 
\Theta(N^{\log_{1/b} a \log_{1/b} \frac{B}{X}}) & \text{if } c = 1 \text{ and } a = 1/b \\
\Theta(N^{\log_{1/b} a}) & \text{otherwise}.
\end{cases}
\]

**Proof.** By lemma 4.18, we have that because $A$ is optimally progressing in the DAM model,
\[ p = \log_{1/b} a. \] By theorem 4.14, \( \rho(W_{A,N,\lambda}) = O(\mathcal{T}(N) + \mathcal{U}(N) + \mathcal{V}(N)) \). where

\[
\begin{align*}
\mathcal{T}(N) &= \begin{cases} 
\max \{ \Theta \left( N^{\log_{1/b} a} \right), a\mathcal{T}(bN) \} & \text{if } \lambda < N \\
\Theta \left( \lambda^{\log_{1/b} a} \right) & \text{if } N \leq \lambda;
\end{cases} \\
\mathcal{U}(N) &= \begin{cases} 
\Theta \left( N^{\log_{1/b} a} \right) + a\mathcal{U}(bN) & \text{if } N = \Omega(\lambda) \text{ and } N^c = \Omega(\lambda) \\
a\mathcal{U}(bN) & \text{if } N = \Omega(\lambda) \text{ and } N^c \neq \Omega(\lambda) \\
0 & \text{otherwise};
\end{cases} \\
\mathcal{V}(N) &= \begin{cases} 
a\mathcal{V}(bN) & \text{if } B \log_{1/b} N = \Omega(\lambda) \text{ and } N^c > B \\
\Theta \left( (B \log_{1/b} N)^{\log_{1/b} a} \right) + a\mathcal{V}(bN) & \text{if } B \log_{1/b} N = \Omega(\lambda) \text{ and } N^c \leq B \\
0 & \text{if } B \log_{1/b} N \neq \Omega(\lambda).
\end{cases}
\end{align*}
\]

Solving the recursion for \( \mathcal{T}(N) \) using the Master method we get

\[ \mathcal{T}(N) = \Theta \left( N^{\log_{1/b} a} \right). \]

As for \( \mathcal{U}(N) \), we note that \( a(bN)^{\log_{1/b} a} = a^{1-c} N^{c \log_{1/b} a} \). When \( 0 < c < 1 \), we have that \( \mathcal{U}(N) \) becomes a geometric series and solves to \( \Theta \left( N^{\log_{1/b} a} \right) \). When \( c = 1 \), we have that \( \mathcal{U}(N) \) is the summation of \( \log_{1/b} N/\lambda \) terms, each of them equal to \( \Theta \left( N^{\log_{1/b} a} \right) \). And when \( c = 0, N^c \neq \Omega(\lambda) \). Hence,

\[ \mathcal{U}(N) = \begin{cases} 
\Theta \left( N^{\log_{1/b} a} \log_{1/b} N \right) & \text{if } c = 1 \\
0 & \text{if } c = 0 \\
\Theta \left( N^{\log_{1/b} a} \right) & \text{otherwise}.
\end{cases} \]

We bound \( \mathcal{V}(N) \) from above. Note that as long as \( N_0 > \lambda \geq (dB \log_{1/b} B)^{1+\epsilon} \), by lemma 4.19 we have that

\[ \frac{N_0^{1/(1+\epsilon)}}{\log_{1/b} N_0} > B \Rightarrow N_0^{1/(1+\epsilon)} > B \log_{1/b} N_0. \]

And when \( N_1 \leq \lambda \), we have \( B \log_{1/b} N_1 \leq B \log_{1/b} \lambda < \lambda^{1/(1+\epsilon)} < \lambda \) by another application of lemma 4.19 because \( \lambda \geq (dB \log_{1/b} B)^{1+\epsilon} \). Therefore \( \mathcal{V}(N) = O(\mathcal{Z}(N)) \) where

\[
\mathcal{Z}(N) = \begin{cases} 
\Theta \left( N^{\log_{1/b} a} \right) + a\mathcal{Z}(bN) & \text{if } N > \lambda \\
\Theta \left( \lambda^{\log_{1/b} a} \right) & \text{if } N \leq \lambda.
\end{cases}
\]

\[ = O \left( N^{\log_{1/b} a} \right) \]

by applying the Master method.

The theorem statement follows from summing the terms for \( \mathcal{T}(N), \mathcal{U}(N) \) and \( \mathcal{V}(N) \). □
Theorem 4.21. Suppose $A$ is an $(a, b, c)$-regular algorithm with tall-cache requirement $H(B)$ and linear space complexity. Suppose also that, in the DAM model, $A$ is optimally progressing for a problem with progress bound $\rho(\square_N) = \Theta(N^p)$, for constant $p$. Assume that $B \geq 4$, pick an $\epsilon \in (0, 0.1)$, and let $d = 3(1 + \epsilon)$ and $\lambda = \max\{H(B), (dB \log_{1/b} B)^{1+\epsilon}\}$.

1. If $c < 1$, then $A$ is optimally progressing an optimally cache-adaptive among all $\lambda$-tall profiles.

2. If $c = 1$, then $A$ is $\Theta\left(\log_{1/b} \frac{N}{\lambda}\right)$ away from being optimally progressing and $O\left(\log_{1/b} \frac{N}{\lambda}\right)$ away from being optimally cache-adaptive.

Proof. Suppose $M$ is a square $N$-fitting profile for $A$. By lemma 4.18, we have that $p = \log_{1/b} a$. If $c < 1$, then by theorem 4.20 the maximum possible progress that any algorithm can make on $M$ is $\rho(M) = \Theta(N^p \log_{1/b} M)$. Since $A$ has linear space complexity, $R(N) = O(\rho(\square_N)) = O(N^p \log_{1/b} M)$ by observation 3.10. Therefore, $\rho(M) = \Theta(R(N))$.

If $c = 1$, then by theorems 4.17 and 4.20 $A$ is a factor of $\Theta(\log_{1/b} N/\lambda)$ away from being optimally progressing. Hence, with $O(\log_{1/b} N/\lambda)$ speed augmentation $A$ can out outperform any other memory-monotone algorithm. \qed

Proof of Lemma 4.18

Let $M_1 = H(B)$ and let $M_2 \geq H(B)$ be an arbitrary constant. Since $A$ is optimally progressing in the DAM model, by lemma 3.16 it is optimally cache-adaptive on both profiles $M_1$ and $M_2$. However in the DAM model, optimally cache-adaptive is equivalent to the usual worst-case optimality.

Let $e_{A,M_1,N}$ be the minimum time required for $A$ to solve all instances of size $N$ given $M_1$, and $e_{A,M_2,N}$ the equivalent time with respect to $M_2$. Define

$$M_1'(t) = \begin{cases} M_1 & \text{if } t \leq e_{A,M_1,N}, \\ 0 & \text{otherwise}; \end{cases} \quad M_2'(t) = \begin{cases} M_2 & \text{if } t \leq e_{A,M_2,N}, \\ 0 & \text{otherwise}. \end{cases}$$

We compute $e_{A,M_1,N}$ and $e_{A,M_2,N}$.

$$e_{A,M_1,N} = \Theta\left(\frac{M_1}{B} \left(a^\log_{1/b} \frac{N}{M_1}\right)\right) = \Theta\left(\frac{M_1}{B} \left(\left(\frac{N}{M_1}\right)^{\log_{1/b} a}\right)\right).$$

$$e_{A,M_2,N} = \Theta\left(\frac{M_2}{B} \left(a^\log_{1/b} \frac{N}{M_2}\right)\right) = \Theta\left(\frac{M_2}{B} \left(\left(\frac{N}{M_2}\right)^{\log_{1/b} a}\right)\right).$$

Since $\rho$ is square-additive, we can compute the progress of $\rho$ on $M_1'$ and $M_2'$ as the sum of $\rho$ on squares $\square_{M_1}$ and $\square_{M_2}$ respectively. Remember that each square $\square_X$ is $X$ words tall and
First, as\footnote{\emph{Proof of Lemma 4.19}} \(X/B\) time steps wide. Hence, \(M'_1\) fits between \(\left[\frac{e_A M_1 B}{M_1 B}\right] - 1\) and \(\left[\frac{e_A M_1 B}{M_1 B}\right]\) of \(\square M_1\)'s. Similarly, \(M'_2\) fits between \(\left[\frac{e_A M_2 B}{M_2 B}\right] - 1\) and \(\left[\frac{e_A M_2 B}{M_2 B}\right]\) of \(\square M_2\)’s.

\[
\rho(M'_1) = \Theta\left(\sum_{i=1}^{\frac{e_A M_1 B}{M_1 B}} \rho(\square M_1)\right) = \Theta\left(\frac{e_A M_1 B}{M_1 B} \rho(\square M_1)\right) = \Theta\left(\left(\frac{N}{M_1}\right) \log_{1/b} a \rho(\square M_1)\right)
\]

\[
\rho(M'_2) = \Theta\left(\sum_{i=1}^{\frac{e_A M_2 B}{M_2 B}} \rho(\square M_2)\right) = \Theta\left(\frac{e_A M_2 B}{M_2 B} \rho(\square M_2)\right) = \Theta\left(\left(\frac{N}{M_2}\right) \log_{1/b} a \rho(\square M_2)\right).
\]

Since \(A\) is optimally progressing on both \(M_1\) and \(M_2\), \(\rho(M'_1) = O(R(N))\) and \(\rho(M'_2) = O(R(N))\). Since \(M'_1\) and \(M'_2\) are \(N\)-fitting profiles for \(A\), we get that \(\rho(M'_1) = \Theta(R(N))\) and \(\rho(M'_2) = \Theta(R(N))\).

Therefore, \(\rho(M'_1) = \Theta(\rho(M'_2))\) and we get

\[
\rho(M'_1) = \Theta\left(\left(\frac{N}{M_1}\right) ^{\log_{1/b} a} \rho(\square M_1)\right) = \Theta\left(\left(\frac{N}{M_2}\right) ^{\log_{1/b} a} \rho(\square M_2)\right) = \rho(M'_2).
\]

By simplifying eq. (4.2), we get that

\[
\rho(\square M_2) = \Theta\left(\left(\frac{M_2}{M_1}\right) ^{\log_{1/b} a} \rho(\square M_1)\right) = \Theta\left(M_2 ^{\log_{1/b} a} \frac{\rho(\square M_1)}{M_1 ^{\log_{1/b} a}}\right).
\]

Since \(M_2\) is an arbitrary constant, we can allow \(M_2 = (M_1)^2\) and then we will have

\[
\rho(\square M_2) = \Theta\left(M_2 ^{\log_{1/b} a} \rho(\square M_1)\right).
\]

Since \(\rho(\square X) = \Theta(X^p)\), we have that \(\rho(\square X^2) = \Theta(\rho(\square X)^2)\). Thus,

\[
\rho(\square M_2^2) = \Theta\left(M_1 ^{\log_{1/b} a} \rho(\square M_1)\right) = \Theta\left(\rho(\square M_1)^2\right) \Rightarrow \rho(M_1) = \Theta\left(M_1 ^{\log_{1/b} a}\right).
\]

Therefore, we have shown that for an arbitrary \(M_2\)

\[
\rho(\square M_2) = \Theta\left(M_2 ^{\log_{1/b} a} \frac{\rho(\square M_1)}{M_1 ^{\log_{1/b} a}}\right) = \Theta\left(M_2 ^{\log_{1/b} a}\right).
\]

\[\square\]

**Proof of Lemma 4.19**

First, as \(\epsilon\) is in \((0, 0.1)\), we have that \(d = 3(1 + \delta) < 3.3 < B\).
Second, notice that as long as $\delta \in (0, 0.1)$, $f(x) = x^{1/(1+\delta)}/\log x$ is a strictly increasing function in $x$ on $[e, \infty)$. Let $x_0 = (dB \log B)^{1+\delta}$, and observe that

$$\log x_0 = 3(1 + \delta) (\log d + \log B + \log \log B) < 3(1 + \delta) \log B.$$ 

Therefore,

$$\frac{x_0^{1/(1+\delta)}}{\log x_0} = \frac{dB \log B}{\log x_0} > B \quad \text{since } d = 3(1 + \delta).$$

Since $x^{1/(1+\delta)}/\log x$ is strictly increasing on $[e, \infty)$ and $Z > (dB \log B)^{1+\delta}$, we get that $Z^{1/(1+\delta)} > B \log Z$. 

$\square$
Chapter 5

Deriving Progress Bounds

In this chapter, we exhibit that one can derive progress bounds, which satisfy the axioms of definition 3.8, for several important problems:

• the naïve matrix multiplication problem,
• the naïve all pairs shortest paths problem,
• the longest common subsequence (LCS) and edit distance problems,
• the multipass filter problem,
• the fast fourier transform (FFT) problem,
• the comparison-based sorting problem.

These derivations allow us to apply the general results of chapter 4 to determine the cache-adaptivity of algorithms which solve these problems (see chapter 6).

We utilize the powerful red-blue pebble game technique of Hong and Kung [34] together with the similar red pebble game technique of Savage [48], and also the information tree lower bound techniques of Aggarwal and Vitter [1] to define appropriate functions that satisfy the axioms of definition 3.8.

We begin by describing a computation DAG [34, 48] which is an abstract way of describing computational dependencies of algorithms.

Definition 5.1. Computational dependencies of an algorithm $A$ can be described by a computation DAG, $G$. $G$ is comprised of input/output nodes that have no incoming/outgoing edges. These nodes correspond to the input/output of the problem.

Each internal node of $G$ represents a computation step by the algorithm. Node $u$ is connected to node $v$ via a directed edge $(u \rightarrow v)$, if computing node $v$ requires data/information from node $u$. 

43
We adopt a notion of progress that bears similarity to the *red pebble game* approach of [48]. We define $R$ and $\rho$ with respect to DAGs of computation.

**Definition 5.2.** Given a DAG of computation, $G$, for a problem instance of size $N$ we define $R(N)$ to be the number of nodes in $G$.

We let $\rho(\square_S)$ to be the maximum number of nodes of $G$ computable using a fixed memory of size $S$ and $S/B$ I/Os, maximized over all initial memory contents, and also maximized over all problem instances.

Similarly, for a memory profile $M$ we allow $\rho(M)$ to be the maximum number of nodes of $G$ computable using profile $M$, maximized over all initial memory contents, and also maximized over all problem instances.

We can also define $R$ and $\rho$ for families of DAGs.

**Definition 5.3.** For a family of DAGs, $\{G_i\}$, such that the number of nodes in all $G_i$s is the same, we define $\rho(\square_S) = \max_i \{\rho(\square_S, G_i)\}$. Similarly, for a profile $M$, we let $\rho(M) = \max_i \{\rho(M, G_i)\}$.

As before, we allow $R(N)$ to be the number of nodes of any of $G_i$ for a problem instance of size $N$.

The above definition is a natural generalization of the notion of an *$S$-span* which is defined on a *red pebble game* [48].

**Definition 5.4** (From [48]). Given a DAG of computation, $G$, the *red pebble game* is played using the following rules.

*(Initialization)* A pebble can be placed on an input node at any time.

*(Computation Step)* A pebble can be placed on (or moved to) any non-input node only if all its immediate predecessors carry pebbles.

*(Pebble Deletion)* A pebble can be removed at any time.

*(Goal)* Each output node must be pebbled at least once.

The red pebble game is an abstraction of data transfer and computation in a two-level memory hierarchy. A pebble placement on an input/output node is akin to reading/writing a portion of the input/output data. A pebble placement on an internal nodes corresponds to a computation step in the DAG $G$ that is done in memory. Removal of a pebble models the erasure or overwriting of the value associated with the node on which the pebble resides from memory.
Definition 5.5 (From [48]). Given a computation DAG, $G$, the S-span of $G$, is the maximum number of nodes of $G$ that can be pebbled with $S$ pebbles in the red pebble game maximized over all initial placements of $S$ red pebbles. (The initialization rule is disallowed.)

Many DAM lower bound proofs use the machinery of information speed function from [34] to lower bound the I/O complexity of algorithms.

Definition 5.6 (From [34]). Consider a DAG of computation $G$. We refer to node-disjoint paths from input nodes to output nodes as lines.

We say that the information speed function is $\Omega(F(d))$ if for any two nodes $u$, $v$ on the same line that are at least $d$ apart, there are $F(d)$ nodes in $G$ satisfying the following two properties.

(a) None of these nodes belong to the same line.

(b) Each of these nodes belongs to a path connecting $u$ and $v$.

In the following lemma, we give a tool to transform the bounds on the information speed function into bounds on the S-span. Its proof is a restructuring of the argument of Theorem 5.1 in [34]. This transformation allows us to seamlessly port some DAM lower bounds to progress bounds in the cache-adaptive model. Later, we exhibit this porting for LCS (see lemma 5.23) and Jacobi Multipass Filter (see lemma 5.28) problems.

Lemma 5.7. For any DAG, $G$, where all input nodes can reach all output nodes through lines (definition 5.6), if the information speed function is $\Omega(F(d))$ where $F$ is monotonically increasing and $F^{-1}$ exists, then S-span of $G$ is $O(SF^{-1}(S))$.

Proof. Let $I_S$ be any initial placement of $S$ pebbles, and let RPG($I_S$) denote the nodes that could be pebbled in the red pebble game using the $S$ pebbles in $I_S$.

First, note that the nodes of RPG($I_S$) must be on at most $S$ lines, because there are initially $S$ pebbles on nodes and lines are node-disjoint.

Claim 5.8. RPG($I_S$) has at most $F^{-1}(S) + 1$ nodes on any line.

Proof. Suppose that the claim is false for some line. Then on this line there are two nodes $u$ and $v$ in RPG($I_S$) that are $F^{-1}(S) + 1$ apart. WLOG assume that $v$ is a decedent of $u$.

Hence, there should be $z = F(F^{-1}(S) + 1)$ nodes satisfying properties (a) and (b) in definition 5.6. Because $F$ is monotone increasing, we have that $z > S$.

We argue that all of these $z$ nodes must be in RPG($I_S$). Because they satisfy property (a) they are on some path from $u$ to $v$. But any valid pebbling in the red pebble game should pebble all the nodes on all paths that connect $u$ and $v$ to be able to pebble $v$. 

45
However, by property (a), these $z$ nodes must belong to distinct lines. But this is a contradiction, because $z > S$ and nodes of $\text{RPG}(I_S)$ can be on at most $S$ lines.

The lemma follows from the above claim, because nodes of $\text{RPG}(I_S)$ are on at most $S$ lines. So $|\text{RPG}(I_S)| = O(SF^{-1}(S))$. 

\section{A Progress Bound for the Naïve Matrix Multiplication Problem}

The matrix multiplication problem is formally defined as follows.

\textbf{Definition 5.9.} The matrix multiplication problem is concerned with computing $C = A \times B$, where

$$C_{ij} = \sum_k A_{ik} \times B_{kj}. \quad (5.1)$$

For simplicity we assume that $A$, $B$, and $C$ are all $\sqrt{N} \times \sqrt{N}$ matrices.

\section*{A Cache-Oblivious Matrix Multiplication Algorithm}

Frigo et al. [31] give a cache-oblivious recursive matrix multiplication algorithm, MM-Inplace. The MM-Inplace algorithm computes eight sub-products of quadrants of matrices $A$ and $B$ “in place”, adding the results of the elementary multiplication into the output matrix. We refer the interested reader for a complete exposition to [31]; we give the pseudo-code for this algorithm in section 6.1 – see algorithm 1.

We argue that the matrix multiplication problem has a progress function $\rho$ and progress requirement function $R(N)$ which together constitute a progress bound for the matrix multiplication problem.

Frigo et al. [31] showed that the MM-Inplace algorithm is optimal in the DAM model among algorithms that multiply two $\sqrt{N} \times \sqrt{N}$ matrices using just inner products to compute entries in the product matrix , i.e. each $C_{ij}$ is computed by using eq. (5.1). The additions in these inner products are allowed to be performed in any order. We refer to the class of these algorithms as Naïve-MM.

Several authors have established a lower bound for the Naïve-MM algorithms. Hong and Kung [34] give a lower bound on the I/O complexity of Naïve-MM algorithms by analyzing the powerful red-blue pebble game technique on the computation DAG of any Naïve-MM algorithm. Savage [48] also gives a similar lower bound using the red pebble game.
Finally, Irony et al. [36] give the same lower bound using a geometric argument that bounds the maximum number of elementary multiplications that can be done in a fixed memory of size $Z$ and $Z/B$ I/Os.

**Definition 5.10.** We define $\rho_\mu$ and $R_\mu$ to be the $\rho$ and $R$ of definition 5.2 defined on the family of DAGs of Naïve-MM algorithms.

Savage [48] proves the following lemma which upperbounds the size of an $S$-span for Naïve-MM algorithms.

**Lemma 5.11 (From [48]).** If $G$ is a DAG of any Naïve-MM algorithm, an $S$-span of $G$ is at most $2S^{3/2}$, when $S < N^2$ (the input matrices do not fit in memory).

**Lemma 5.12.** We have that $\rho_\mu(\square_S) = \Theta(S^{3/2})$ and $R_\mu(N) = \Theta(N^{3/2})$. Moreover, $\rho_\mu$ and $R_\mu$ constitute a progress bound for the Naïve-MM problem.

*Proof.* We first show that $\rho_\mu(\square_S) = \Theta(S^{3/2})$ for the class of Naïve-MM algorithms. Let $G_i$ be the family of DAGs in Naïve-MM. By lemma 5.11, we have that $S$-span of any of DAGs in $G_i$ is at most $2S^{3/2}$. Since any computation of nodes of a DAG corresponds to a pebbling strategy in the red pebble game, we have that $\rho_\mu(\square_S) \leq 2S^{3/2}$. As exhibited by the MM-Inplace algorithm, some DAGs in $G_i$ correspond to recursive evaluations of eq. (5.1). Consider one of these DAGs, $F$. We argue that $\rho_\mu(\square_S, F) = \Omega(S^{3/2})$. Since, $\rho$ is maximized over all DAGs in $G_i$, we have that $\rho_\mu(\square_S) \geq \rho_\mu(\square_S, F) = \Omega(S^{3/2})$.

Consider a recursive evaluation of eq. (5.1) and a level of recursion in which two sub-matrices $A_1$ and $B_1$ of size $(S^{1/2}/3) \times (S^{1/2}/3)$ are multiplied. Both $A_1$ and $B_1$ can be brought into memory using $\leq 2S/9B$ I/Os and they fit together in a memory of size $S$. Multiplying them in memory takes no extra I/Os and writing the inputs and the result back takes at most $S/3B$ I/Os. To perform the multiplication of $A_1$ and $B_1$ $\Theta\left(\left(S^{1/2}/3\right)^3\right) = \Theta(S^{3/2})$ operations of $F$ must be completed. Therefore, we have that $\rho_\mu(\square_S, F) = \Omega(S^{3/2})$.

**Functions $\rho_\mu$ and $R_\mu$ constitute a progress bound** We have that the size of each DAG of computation in $G_i$ for a problem of size $N$ is the same number $R_\mu(N) = \Theta(N^{3/2})$, because eq. (5.1) contains $\Theta(N^{3/2})$ multiplications and additions.

Consider any DAG $E$ in $G_i$. If $\rho_\mu(M, E) < R_\mu(N)$ for a profile $M$, then no algorithm whose DAG is $E$ can compute all nodes of the $E$ in $M$. Thus, if $\rho_\mu(M) = \max\{\rho_\mu(\square_S, G_i)\}$ is less than $R_\mu(N)$, no algorithm can solve all problem instances of size $N$ in $M$.

We now prove square-additivity (definition 3.7) and monotonicity (definition 3.6) of $\rho_\mu$. 

47
Square-additivity  We have already established that $\rho_{\mu}(\square S) = \Theta(S^{3/2})$, therefore $\rho_{\mu}(\square S)$ is bounded by a polynomial in $S$.

Consider the profile $Z = \square z_1 \| \ldots \| \square z_k$. We first argue that $\rho_{\mu}(Z) \leq \sum_\ell \rho_{\mu}(\square z_\ell)$. The memory available at the beginning of $\square z_{\ell+1}$ is at most $z_{\ell+1}$. By definition, no matter the initial content of cache after the execution of an algorithm on $\square z_1 \| \ldots \| \square z_\ell$, the maximum number of nodes of any of $G_i$s computable in $\square z_{\ell+1}$ by using $z_{\ell+1}$ memory and $z_{\ell+1}/B$ I/Os is $\rho_{\mu}(\square z_{\ell+1})$. Therefore, we have that $\rho_{\mu}(Z) \leq \sum_\ell \rho_{\mu}(\square z_\ell)$.

Next, we argue that $\rho_{\mu}(Z) = \Omega(\sum_\ell \rho_{\mu}(\square z_\ell))$. Remember that $F$ is a DAG in $G_i$ which corresponds to a recursive evaluation of eq. (5.1). We already have argued that $\rho_{\mu}(\square z_i, F) = \Omega(S^{3/2}/2) = \Omega(\rho_{\mu}(\square z_i))$.

Now, we show that $\rho_{\mu}(Z, F) = \Omega(\sum_\ell \rho_{\mu}(\square z_\ell, F)) = \Omega(\sum_\ell \rho_{\mu}(\square z_\ell))$. Because $\rho_{\mu}$ is defined to be the maximum over all DAGs in $G_i$, we have that $\rho_{\mu}(Z) \geq \rho_{\mu}(Z, F) = \Omega(\sum_\ell \rho_{\mu}(\square z_\ell))$.

For each square $\square z_\ell$, we consider the recursive evaluation of eq. (5.1) at a level in which two submatrices $A_\ell$ and $B_\ell$ of size $(z^{1/2}_{\ell}/3) \times (z^{1/2}_{\ell}/3)$ are multiplied. As before, one can see that multiplying $A_\ell$ and $B_\ell$ can be done in $\square z_\ell$ and it includes computing $\Omega(z^{3/2}_{\ell})$ nodes of $F$. Since $\rho_{\mu}$ is also taken as the maximum among all problem instances, there exists a (possibly huge) problem instance, $I$, such that for each $\square z_\ell$, there are subproblems ($A_\ell$ and $B_\ell$) in $I$ that are untouched (not computed) in any of any of the previous squares ($\square z_t$ for $t < \ell$). Therefore, we get that $\rho_{\mu}(Z, F) = \Omega(\sum_\ell \rho_{\mu}(\square z_\ell, F)) = \Omega(\sum_\ell \rho_{\mu}(\square z_\ell))$.

Monotonicity  Let $M \prec U$ be two memory profiles. Then, we have $M = L_1 \| \ldots \| L_k$ and $U = U_0 \| U_1 \| \ldots \| U_k$, such that $U_i$ is both above $L_i$ and has a longer duration. Since for each $U_i$ is above $L_i$ and has a longer duration than it, it must be the case that maximum number of nodes computable in $U$ is bigger or equal than those in $M$.

5.2 A Progress Bound for the Naïve All Pairs Shortest Paths Problem

We first define the All Pairs Shortest Paths (APSP) problem.

Definition 5.13. Given a weighted graph $Q$, that does not have negative cycles, the All Pairs Shortest Paths (APSP) problem is concerned with finding the length (sum of weights on edges) of shortest paths between all pairs of nodes.
The dynamic programming solution of Floyd-Warshall [28, 52] to the APSP problem is based on the following recursive relation.

Allow \( SP(i, j, k) \) be the shortest possible path from \( i \) to \( j \) using nodes only from the set \( \{1, 2, \ldots, k\} \) as intermediate points along the way. Let \( w(i, j) \) be the weight of the edge between nodes \( i \) and \( j \). \( SP(i, j, k) \) satisfies the recursive relation

\[
SP(i, j, 0) = w(i, j) \\
SP(i, j, k + 1) = \min\{SP(i, j, k), SP(i, k + 1, k) + SP(k + 1, j, k)\}.
\] (5.2)

For simplicity we assume that \( Q \) has \( \sqrt{N} \) nodes and the adjacency matrix of \( Q \) occupies \( N = \sqrt{N} \times \sqrt{N} \) space.

A Cache-Oblivious Floyd-Warshall APSP Algorithm

Park et al. give a recursive cache-oblivious algorithm for the APSP problem. We refer the interested reader for a complete exposition of this algorithm to their paper [45]. We present the pseudo-code for the \( FW-APSP \) algorithm in section 6.1 – see algorithm 2.

Park et al. exhibit that \( FW-APSP \) is optimal in the DAM model among all algorithms that solve the APSP problem by computing the \( \Theta\left(\sqrt{N}^3\right) = \Theta(N^{3/2}) \) operations needed to implement the type of computation defined by eq. (5.2). We refer to the class of such algorithms as \( Naïve-APSP \).

Definition 5.14. We define \( \rho_\alpha \) and \( R_\alpha \) to be the \( \rho \) and \( R \) of definition 5.2 defined on the family of DAGs of \( Naïve-APSP \) algorithms.

Lemma 5.15. Family of DAGs produced by algorithms in \( Naïve-APSP \) is a subfamily of DAGs produced by algorithms in \( Naïve-MM \).

Proof. We claim that the data dependencies of eq. (5.2) is equivalent to a particular ordering on the additions and multiplications in eq. (5.1). To see this, map the \( addition/min \) nodes in DAGs of Naïve-APSP to the \( multiplication/addition \) nodes in DAGs of Naïve-MM respectively. In this interpretation eq. (5.2) is a computation of eq. (5.1), where \( A_{it} \times B_{ij} \) must be computed and added to the previous value for \( C_{ij} \) before any of \( A_{it} \times B_{lj} \) multiplications for all \( \ell > t \).

This means that the each DAG produced by a Naïve-APSP algorithm has a corresponding DAG in Naïve-MM. \( \Box \)

Lemma 5.16. If \( G \) is a DAG of any \( Naïve-APSP \) algorithm, an \( S \)-span of \( G \) is at most \( O(S^{3/2}) \), when \( S < N^2 \) (the input matrices do not fit in memory).
Proof. The statement is a direct corollary of lemma 5.15 and lemma 5.11.

Lemma 5.17. We have that $\rho_\alpha(\square s) = \Theta(S^{3/2})$ and $R_\alpha(N) = \Theta(N^{3/2})$. Moreover, $\rho_\alpha$ and $R_\alpha$ constitute a progress bound for the Naïve-APSP problem.

Proof. We have that the size of each DAG of computation in Naïve-APSP for a problem of size $N$ is the same number $R_\alpha(N) = \Theta(N^{3/2})$, because eq. (5.2) contains $\Theta(\sqrt{N^3}) = \Theta(N^{3/2})$ additions and min operations.

The rest of the proof is almost identical to the proof of lemma 5.12 except that we utilize lemma 5.16 to bound $\rho_\alpha(\square s)$ from above.

5.3 Progress Bounds for the LCS and Edit Distance Problems

We begin by a short exposition of the Longest Common Subsequence problem.

Definition 5.18. A sequence $X = (x_1, \ldots, x_m)$ is a subsequence of sequence $Y = (y_1, \ldots, y_n)$ if there exists a strictly increasing function $f : [1, m] \rightarrow [1, n]$ such that for all $i \in [1, m]$, $x_i = y_{f(i)}$.

Definition 5.19. Given two sequences $X$ and $Y$, the Longest Common Subsequence (LCS) problem is concerned with finding the longest subsequence of $X$ that is also a subsequence of $Y$.

The dynamic programming solution [25] to the LCS problem is based on the following recursive relation. Given sequences $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots, y_n)$, define $c[i, j]$ to be the length of the LCS for $(x_1, \ldots, x_i)$ and $(y_1, \ldots, y_j)$. $c[i, j]$ can be computed from the following recursive relation

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{c[i - 1, j], c[i, j - 1]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases} \quad (5.3)$$

Once the value of $c[m, n]$ is determined and all entries of the $c[i, j]$ are available, one can trace back the sequence of decisions that led to the value computed for $c[m, n]$, and thus retrieve the elements on an LCS of $X$ and $Y$.

A naive implementation of the above recursion computes the table $c$ in a row-major or column-major order and is very cache inefficient, as it incurs $\Theta(mn/B)$ I/Os.

We give a short description of the Edit Distance problem as well.
Definition 5.20. Given two sequences $X$ and $Y$, the Edit Distance problem is concerned with finding the smallest cost edit sequence that transforms $X$ to $Y$.

The edit operations are: delete($x_i$) of cost $D(x_i)$ that deletes $x_i$ from $X$, insert($y_j$) of cost $I(y_j)$ that inserts $y_j$ into $X$, and substitute($x_i, y_j$) of cost $S(x_i, y_j)$ that replaces $x_i$ with $y_j$ in $X$.

The Recursive LCS (and Edit Distance) Algorithm

Chowdhury and Ramachandran in [21] describe a cache-oblivious optimal recursive algorithm for the LCS problem that also works for the Edit Distance problem by changing the character transformation cost function. We refer the interested reader for a complete exposition of this algorithm to [21]; we only give a high-level picture in section 6.3. They prove that Recursive-LCS is optimal in DAM model among all algorithms that execute the $\Theta(N^2)$ operations needed to implement the type of computation defined by eq. (5.3).

All the algorithms in this class share a unique DAG of computation, $G_\psi$ that is described by the data dependencies of eq. (5.3), see fig. 5.1.

Chowdhury and Ramachandran also prove that the Edit-Distance algorithm, which is derived by replacing the cost function of Recursive-LCS, is optimal in DAM among a similar class of algorithms. Thus, both problems share a unique DAG of computation, $G_\psi$.

![Figure 5.1: The DAG of computation, $G_\psi$, for the LCS and Edit Distance problems.](image)

Definition 5.21. We define $\rho_\psi$ and $R_\psi$ to be the $\rho$ and $R$ of definition 5.2 defined on $G_\psi$.

To prove the an upper bound for the $S$-span of $G_\psi$, we use the following bound by Chowdhury and Ramachandran [21] together with lemma 5.7.
Lemma 5.22 (From [21]). The information speed function for $G_\psi$ satisfies $F_{G_\psi}(d) = \Omega(d)$. The inverse of $F_{G_\psi}(d)$ exists, and $F^{-1}_{G_\psi}(d) = O(d)$.

Lemma 5.23. An $S$-span of $G_\psi$ is at most $O(S^2)$, when $S < N^2$ (the dynamic programming table does not fit in memory).

Proof. The bound follows from lemma 5.7 and the upper bound on $F^{-1}_{G_\psi}(S)$.

Lemma 5.24. We have that $\rho_\psi(\Box_S) = \Theta(S^2)$ and $R_\psi(N) = \Theta(N^2)$. Moreover, $\rho_\psi$ and $R_\psi$ constitute a progress bound for the LCS problem.

Proof. Computing LCS from eq. (5.3) for a problem of size $N$ requires $\Theta(N^2)$ operations, so $R_\psi(N) = |G_\psi(N)| = \Theta(N^2)$.

The rest of the proof is a repetition of the argument of lemma 5.12, except that here we only have a unique DAG $G_\psi$, and we utilize lemma 5.23 to bound $\rho_\psi(\Box_S)$ from above.

5.4 A Progress Bound for the Multipass Filter problem

We first define the multipass filter problem.

Definition 5.25. A one-dimensional multipass filter on an array $A$ of size $N$ is comprised of computing values of generations $A^{t+1}$ from values at generation $t$ according to some update rule. A typical update function is

\[
A^{t+1}_i \leftarrow \frac{(A^t_{i-1} + A^t_i + A^t_{i+1})}{3} \quad (5.4)
\]

We consider computing $N$ generations of the update rule on $A$ that has $N$ elements.

Multipass filters are used in Jacobi iteration for solving heat-diffusion equations and simulation of lattice gases with cellular automata.

The Cache-Oblivious Recursive Jacobi Multipass Filter Algorithm

Prokop [47] gives a recursive cache-oblivious Jacobi Multipass Filter algorithm. We refer the interested reader to [47] for a complete exposition; we only give the pseudo-code in section 6.2.

Prokop [47] proves that JACOBI is optimal in DAM model among all algorithms that execute the $\Theta(N^2)$ operations needed to implement the type of computation defined by eq. (5.4).

All the algorithms in this class share a unique DAG of computation, $G_\eta$ that is described by the data dependencies of eq. (5.4), see fig. 5.2.
Definition 5.26. We define $\rho_\eta$ and $R_\eta$ to be the $\rho$ and $R$ of definition 5.2 defined on $G_\eta$.

To prove the an upper bound for the $S$-span of $G_\eta$, we use the following bound by Prokop [47] together with lemma 5.7.

Lemma 5.27 (From [47]). The information speed function for $G_\eta$ satisfies $F_{G_\eta}(d) = \Omega(d)$. The inverse of $F_{G_\eta}(d)$ exists, and $F_{G_\eta}^{-1}(d) = O(d)$.

Lemma 5.28. An $S$-span of $G_\eta$ is at most $O(S^2)$, when $S < N^2$ (the $N$ generations of computing the array does not fit in memory).

Proof. The bound follows from lemma 5.7 and the upper bound on $F_{G_\eta}^{-1}(S)$.

Lemma 5.29. We have that $\rho_\eta(\Box_S) = \Theta(S^2)$ and $R_\eta(N) = \Theta(N^2)$. Moreover, $\rho_\eta$ and $R_\eta$ constitute a progress bound for the multipass filter problem.

Proof. Computing $N$ generations of a multipass filter from 5.4 on an array of size $N$ requires $\Theta(N^2)$ operations, so $R_\eta(N) = |G_\eta(N)| = \Theta(N^2)$.

The rest of the proof is a repetition of the argument of lemma 5.12, except that here we only have a unique DAG $G_\eta$, and we utilize lemma 5.28 to bound $\rho_\eta(\Box_S)$ from above.

5.5 A Progress Bound for the FFT Problem

We first define the Discrete Fourier Transform (DFT) problem.
Definition 5.30. Let $X$ be an array of $N$ complex numbers. The **Discrete Fourier Transform (DFT)** of $X$ is an array $Y$ defined by the formula

$$Y[i] = \sum_{j=0}^{N-1} X[j] w_N^{-ij} \quad i = 0, \ldots, N-1; \quad (5.5)$$

where $w_N = e^{2\pi\sqrt{-1}/N}$.

DFT is the most important discrete transform used to perform Fourier analysis in many practical applications such as digital signal processing, image processing, solving partial differential equations, or even multiplying large integers. Directly computing eq. (5.5) requires $\Theta(N^2)$ operations.

A **Fast Fourier Transform (FFT)** algorithm is a recursive method of computing a DFT by computing only $\Theta(N \log N)$ operations.

Definition 5.31. Let $N = N_1 N_2$ be any integer factorization of $N$. A **Cooley-Tukey Fast Fourier Transform (FFT)** algorithm [24] A computes eq. (5.5) by computing

$$Y[i_1 + i_2 N_1] = \sum_{j_2=0}^{N_2-1} \left( \sum_{j_1=0}^{N_1-1} X[j_1 N_2 + j_2] w_N^{-i_1 j_1} \right) w_N^{-i_2 j_2}; \quad (5.6)$$

when $N$ is $\Omega(1)$. When $N = O(1)$, A computes eq. (5.5) directly.

The $w_N^{-i_2 j_2}$ terms are called **twiddle factors** [27].

Note, that the inner and outer summations in eq. (5.6) are both DFTs. This observation helps FFT algorithms to recursively compute DFTs.

**The Cache-Oblivious FFT Algorithm**

Frigo et al. give a recursive cache-oblivious FFT algorithm, CO-FFT. We refer the interested reader for a complete exposition of this algorithm to [31]. We present the pseudo-code and a high-level description for the CO-FFT algorithm in section 7.2 – see algorithm 8.

The class of Cooley-Tukey FFT algorithms (definition 5.31) describes a unique DAG of computation $G_\phi$. Frigo et al. [31] prove that CO-FFT is optimal in DAM model among all Cooley-Tukey FFT algorithms by using a lower bound given by Hong and Kung [34], who study $G_\phi$.

Definition 5.32. We define $\rho_\phi$ and $R_\phi$ to be the $\rho$ and $R$ of definition 5.2 defined on $G_\phi$.

Savage [48] proves the following lemma which upperbounds the size of an $S$-span for $G_\phi$. 

54
Lemma 5.33 (From [48]). The $S$-span of $G_{\phi}$ on $N$ input nodes is at most $2S\log S$ (when $S \leq n$).

Lemma 5.34. We have that $\rho_{\phi}(\square S) = \Theta(2S\log S)$ and $R_{\phi}(N) = \Theta(N\log N)$. $\rho_{\phi}$ and $R_{\phi}$ constitute a progress bound for the Cooley-Tukey FFT problem.

Proof. A Cooley-Tukey FFT algorithm needs to compute $R_{\phi}(N) = |G_{\phi}(N)| = \Theta(N\log N)$ operations from 5.6 to solve a problem of size $N$.

The rest of the proof is a repetition of the argument of lemma 5.12, except that here we only have a unique DAG $G_{\phi}$, and we utilize lemma 5.33 to bound $\rho_{\phi}(\square S)$ from above. \qed

5.6 A Progress Bound for the Sorting Problem

We use the external memory sorting lower bound of Aggarwal and Vitter [1] to define a progress bound for the sorting problem.

Definition 5.35. Given an array $A$ of $N$ elements and a total ordering defined on the elements of $A$, the comparison-based sorting problem is concerned with producing an output array of the elements of $A$ in ascending order.

Definition 5.36. We define $R_{\gamma}(N)$ to be the number of bits information that any comparison-based algorithm should learn to be able to sort all input instances of size $N$.

We define $\rho_{\gamma}(\square X)$ to be the maximum number of bits of information that could be learned about the ordering of elements in the input array using a fixed memory of $X$ and $X/B$ I/Os, maximized over all initial memory contents and also maximized over all problem instances.

Similarly, we define $\rho_{\gamma}(M)$ to be the maximum number of bits of information that could be learned about the ordering of elements in the input array in the profile $M$, maximized over all initial memory contents and also maximized over all problem instances.

Lemma 5.37. We have that $\rho_{\gamma}(\square S) = \Theta(S\lg S)$ and $R_{\gamma}(N) = \Theta(N\lg N)$. Moreover, $\rho_{\gamma}$ and $R_{\gamma}$ constitute a progress bound for the comparison-based sorting problem. \cite{1}

Proof. We rephrase the argument of Aggarwal and Vitter from [1]. Any comparison-based sorting algorithm must learn enough bits to be able to distinguish and generate all the $N!$ possible outcomes of the sort. Since each comparison has two possibles outcomes, it is one bit of information. Therefore, we have that $N\lg N/2 \leq R_{\gamma}(N) = \lg N! \leq N\lg N$.

When the memory size is $S$, we have that each time an algorithm reads in a block, it can compare every element in that block with every other element in memory. These comparisons

\cite{1}We use the notation $\lg = \log_2$. 

55
are not all independent though, so the total number of distinct outcomes learnable for each block is bounded by \((\frac{S}{B})!\).

Thus, using \(S/B\) I/Os, the number of bits learnable by any algorithm maximized over all initial memory contents is bounded by

\[
\rho_\gamma(\square_s) \leq \frac{S}{B} \log \left( \left( \frac{S}{B} \right)! \right) \leq \frac{S}{B} \log \left( \left( \frac{eS}{B} \right)^B \right) \leq O(S \log S).
\]

We now argue that \(\rho_\gamma(\square_s) = \Omega(S \log S)\). Consider an input array \(C\) of size \(S/2\). Since \(C\) fits in memory, an external sorting algorithm, like the external merge-sort can load \(C\) into memory, sort it and write it back using a memory of size \(S\) and \(S/B\) I/Os. Since \(C\) can range over all inputs of size \(S/2\), the algorithm should be able to distinguish between all the possible \(S!\) outcomes. Therefore, we have that \(\rho_\gamma(\square_s) = \Omega(R_\gamma(S/2)) = \Omega(S \log S)\).

It follows from definition that if \(\rho_\gamma(M) < R_\gamma(N)\), no comparison-based sorting algorithm can solve all problems of size \(N\).

Monotonicity of \(\rho_\gamma\) follows from definition. We argue that \(\rho_\gamma\) is square-additive. Consider the profile \(Z = \square_{z_1} \| \ldots \| \square_{z_k}\). A repetition of the argument of lemma 5.12 shows that \(\rho_\gamma(Z) \leq \sum \rho_\gamma(\square_{z_\ell})\).

Assume that \(I\) is in arbitrary (and possibly huge) problem instance. For each square \(\square_{z_\ell}\), we focus on a subarray of \(z_\ell/2\) consecutive elements in \(I\), which are untouched by previous squares \(\square_{z_t}\) for all \(t < \ell\). Like, above, these \(z_\ell/2\) elements can be loaded into a memory of size \(z_\ell\), sorted inside memory and written back using \(z_\ell/B\) I/Os.

Since \(\rho_\gamma\) is taken as the maximum among all problem instances, the subarrays for each \(\square_{z_\ell}\) can range over all possible inputs of size \(z_\ell/2\) as well. This means that for each \(\ell\), it must be the case that \(\rho_\gamma(\square_{z_\ell}) = \Omega(R_\gamma(z_\ell/2)) = \Omega(z_\ell \log z_\ell)\) and \(\rho_\gamma(Z) = \Omega(\sum \rho_\gamma(\square_{z_\ell}))\).

\(\square\)
Chapter 6

Optimal Recursive Cache-Adaptive Algorithms

In this chapter, we exhibit several applications of the optimality criteria theorems of chapter 4 to prove optimality of algorithms in the cache-adaptive model.

6.1 Optimal Matrix Multiplication and Floyd-Warshall APSP

We present the pseudo-code for the MM-INPLACE and FW-APSP algorithms (see algorithm 1 and algorithm 2 respectively). For complete descriptions refer to [31] and [45] respectively.

We can now apply theorem 4.21 to show that MM-INPLACE and FW-APSP algorithms are optimally progressing.

Theorem 6.1. For all $\Theta(B^2)$-tall memory profiles, the MM-INPLACE algorithm [31] and the FW-APSP algorithm [45] are optimally progressing and cache-adaptive for the Naïve-MM and Naïve-APSP classes respectively.

Proof. Both MM-INPLACE and FW-APSP require a $\Theta(B^2)$-tall cache to be optimal in the DAM model.

By lemma 5.12, we have that $\rho_\mu(\square_X) = \Theta(X^{3/2})$ and $R_\mu(N) = \Theta(N^{3/2})$ constitute a progress bound for the naïve matrix multiplication problem. And by lemma 5.17, we have that $\rho_\alpha(\square_X) = \Theta(X^{3/2})$ and $R_\alpha(N) = \Theta(N^{3/2})$ constitute a progress bound for the naïve APSP problem.
Algorithm 1 The cache-oblivious matrix multiply with O(1) additive overhead [31]. In this code, A, B, and C are passed by reference.

1: function MM-Inplace(N,i,j,k,C,A,B)  
2:     if N = O(1) then  
3:         C[i][k] ← A[i][j] × B[j][k]  
4:     else  
5:         i' ← i + N/2, j' ← j + N/2, k' ← k + N/2  
6:         MM-Inplace(N/2,i,j,k,C,A,B)  
7:         MM-Inplace(N/2,i,j,k',C,A,B)  
8:         MM-Inplace(N/2,i',j,k,C,A,B)  
9:         MM-Inplace(N/2,i',j,k',C,A,B)  
10:        MM-Inplace(N/2,i,j',k,C,A,B)  
11:        MM-Inplace(N/2,i,j',k',C,A,B)  
12:        MM-Inplace(N/2,i',j',k,C,A,B)  
13:        MM-Inplace(N/2,i',j',k',C,A,B)  
14:     end if  
15: end function

Since both MM-Inplace and FW-APSP are (a,b,c)-regular algorithms, and c = 0 we have that by theorem 4.21, they are optimally progressing and cache-adaptive on all Θ(B^2)-tall memory profiles. Note that there exists an ε ∈ (0,0.1) such that max{Θ(B^2), (3(1 + ε)B log_2 B)^1+ε} = Θ(B^2).

6.2 Optimal Jacobi Multipass Filter

The recursive cache-oblivious JACOBI algorithm [47] is described as the composition of three mutually recursive functions: JACOBI△, JACOBI▽ and JACOBI. We present the pseudo-code for the JACOBI algorithm (see algorithm 4). For a complete description refer to [47].

We use theorem 4.14 to prove that the JACOBI algorithm (algorithm 4) is optimally progressing and optimally cache-adaptive.

Theorem 6.2. Assume that B ≥ 4. Pick an ε ∈ (0,0.1) arbitrary close to 0, and let d = 3(1 + ε) and λ = (dB log_2 B)^1+ε. For all λ-tall memory profiles, the cache-oblivious JACOBI algorithm of [47] is optimally progressing and optimally cache-adaptive among all algorithms that execute the Θ(N^2) operations needed to implement the type of computation defined by eq. (5.4).
Algorithm 2 The cache-oblivious Floyd-Warshall APSP algorithm with $O(1)$ additive overhead [45]. The initial call to the recursive algorithm passes the entire input adjacency matrix of the graph as each argument (passed by reference).

Let $A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}$, $B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$ and $C = \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix}$.

1: function FW-APSP($A,B,C$)
2: if $\sqrt{N} = O(1)$ then Iterative-FW($A,B,C$).
3: else
4: FW-APSP($A_1,B_1,C_1$)
5: FW-APSP($A_2,B_1,C_2$)
6: FW-APSP($A_3,B_3,C_1$)
7: FW-APSP($A_4,B_3,C_2$)
8: FW-APSP($A_4,B_4,C_4$)
9: FW-APSP($A_3,B_4,C_3$)
10: FW-APSP($A_2,B_2,C_4$)
11: FW-APSP($A_1,B_2,C_3$)
12: end if
13: end function

Proof. The JACOBI algorithm is computing $N$ generations of the update rule in definition 5.25, on an array $A$ of size $N$. And the algorithm in [47] is described as if, it computes the whole $N \times N$ matrix of $N$ generations. However, JACOBI can be adapted to use only one auxiliary array of size $N$ in addition to the original array $A$. Thus, the space complexity of all the subroutines in the JACOBI algorithm is linear.

Lemma 5.29 shows that $\rho_{\eta}(\Box X) = \Theta(X^2)$ and $R_{\eta}(N) = \Theta(N^2)$ constitute a progress bound for the multipass filter problem among all algorithms that execute the $\Theta(N^2)$ operations needed to implement the type of computation defined by eq. (5.4).

Since all three subroutines of the algorithm are computing the computations of eq. (5.4), the progress bound applies to all three of the functions.

We apply theorem 4.14 by allowing $\lambda = (dB \log_2 B)^{1+\epsilon}$. Let $A_1, A_2, A_3$ represent JACOBIΔ, JACOBI∇ and JACOBI respectively.
Algorithm 3 The Iterative-FW subroutine.

1: \textbf{function} Iterative-FW(A,B,C) \hspace{1em} \triangleright \text{Let the matrices } A, B, C \text{ be of size } \sqrt{N} \times \sqrt{N}.
2: \hspace{1em} \textbf{for} k = 1 \text{ to } \sqrt{N} \text{ do}
3: \hspace{2em} \textbf{for} i = 1 \text{ to } \sqrt{N} \text{ do}
4: \hspace{3em} \textbf{for} i = 1 \text{ to } \sqrt{N} \text{ do}
5: \hspace{4em} A[ij] = \min\{A[ij], B[ik] + C[kj]\}.
6: \hspace{3em} \textbf{end for}
7: \hspace{2em} \textbf{end for}
8: \hspace{1em} \textbf{end for}
9: \textbf{end function}

By simultaneously solving \( T_1(N) \) and \( T_2(N) \), we get that

\[
T_1(N) = \begin{cases} 
\max \{ \Theta(N^2), 3T_1(N/2) + T_2(N/2) \} & \text{if } \lambda < N \\
\Theta(\lambda^2) & \text{if } N \leq \lambda.
\end{cases}
\]

\[
T_2(N) = \begin{cases} 
\max \{ \Theta(N^2), 3T_2(N/2) + T_1(N/2) \} & \text{if } \lambda < N \\
\Theta(\lambda^2) & \text{if } N \leq \lambda.
\end{cases}
\]

Therefore, for \( T_3(N) \), we have

\[
T_3(N) = \begin{cases} 
\max \{ \Theta(N^2), 2T_1(N/2) + 2T_2(N/2) \} & \text{if } \lambda < N \\
\Theta(\lambda^2) & \text{if } N \leq \lambda
\end{cases}
\]

\[= \Theta(N^2).\]

Since \( c_1 = c_2 = c_3 = 0 \), we have that \( N^{c_i} \neq \Omega(\lambda) \) for \( i = 1, 2, 3 \). Therefore,

\[
\mathcal{U}_1(N) = \begin{cases} 
3\mathcal{U}_1(N/2) + \mathcal{U}_2(N/2) & \text{if } N = \Omega(\lambda) \text{ and } N^0 \neq \Omega(\lambda) \\
0 & \text{otherwise}.
\end{cases}
\]

\[
\mathcal{U}_2(N) = \begin{cases} 
3\mathcal{U}_2(N/2) + \mathcal{U}_1(N/2) & \text{if } N = \Omega(\lambda) \text{ and } N^0 \neq \Omega(\lambda) \\
0 & \text{otherwise}.
\end{cases}
\]
Algorithm 4 The Jacobi Multipass Filter algorithm [47].

1: function JACOBI(A,N)
2: \textbf{JACOBI}Δ (A, N, 0, N, 0).
3: \textbf{JACOBI}∇ (A, N, N/2, N, 0).
4: \textbf{JACOBI}Δ (A, N, 0, N, 0).
5: \textbf{JACOBI}∇ (A, N, 0, N, N/2).
6: \textbf{end function}

1: function JACOBI∇(A,N,s,w,τ)
2: \textbf{if} w > 2 \textbf{then}
3: \textbf{JACOBI}∇ (A, N, (s + w/4), w/2, τ).
4: \textbf{JACOBI}Δ (A, N, (s + w/4), w/2, (τ + w/4)).
5: \textbf{JACOBI}∇ (A, N, s, w/2, (τ + w/4)).
6: \textbf{JACOBI}∇ (A, N, (s + w/2), w/2, (τ + w/4)).
7: \textbf{end if}
8: \textbf{end function}

\[ \mathcal{U}_3(N) = \begin{cases} 2\mathcal{U}_1(N/2) + 2\mathcal{U}_2(N/2) & \text{if } N = \Omega(\lambda) \text{ and } N^0 \neq \Omega(\lambda) \\ 0 & \text{otherwise.} \end{cases} \]

We conclude that \( \mathcal{U}_1(N) = \mathcal{U}_2(N) = \mathcal{U}_3(N) = 0. \)

Since \( c_1 = c_2 = c_3 = 0 \), we have that \( N^{c_i} = O(B) \) for \( i = 1, 2, 3 \). Therefore,

\[ \mathcal{V}_1(N) = \begin{cases} 3\mathcal{V}_1(N/2) + \mathcal{V}_2(N/2) & \text{if } B \log_{1/b} N = \Omega(\lambda) \text{ and } N^0 > B \\ \Theta \left( (B \log_{1/b} N)^2 \right) + 3\mathcal{V}_1(N/2) + \mathcal{V}_2(N/2) & \text{if } B \log_{1/b} N = \Omega(\lambda) \text{ and } N^0 \leq B \\ 0 & \text{if } N \neq \Omega(\lambda); \end{cases} \]

\[ \mathcal{V}_2(N) = \begin{cases} 3\mathcal{V}_2(N/2) + \mathcal{V}_1(N/2) & \text{if } B \log_{1/b} N = \Omega(\lambda) \text{ and } N^0 > B \\ \Theta \left( (B \log_{1/b} N)^2 \right) + 3\mathcal{V}_2(N/2) + \mathcal{V}_1(N/2) & \text{if } B \log_{1/b} N = \Omega(\lambda) \text{ and } N^0 \leq B \\ 0 & \text{if } N \neq \Omega(\lambda); \end{cases} \]

\[ \mathcal{V}_3(N) = \begin{cases} 2\mathcal{V}_1(N/2) + 2\mathcal{V}_2(N/2) & \text{if } B \log_{1/b} N = \Omega(\lambda) \text{ and } N^0 > B \\ \Theta \left( (B \log_{1/b} N)^2 \right) + 2\mathcal{V}_2(N/2) + 2\mathcal{V}_1(N/2) & \text{if } B \log_{1/b} N = \Omega(\lambda) \text{ and } N^0 \leq B \\ 0 & \text{if } N \neq \Omega(\lambda). \end{cases} \]

Note that as long as \( N_0 > \lambda \geq (dB \log_2 B)^{1+\epsilon} \), by lemma 4.19 we have that

\[ \frac{N_0^{1/(1+\epsilon)}}{\log_2 N_0} > B \Rightarrow N_0^{1/(1+\epsilon)} > B \log_2 N_0. \]
Algorithm 5 The JACOBI\(\Delta\) subroutine.

1: \textbf{function} JACOBI\(\Delta\)(\(A,N,s,w,\tau\))
2: \hspace{1em} \textbf{if} \(w > 2\) \textbf{then}
3: \hspace{2em} JACOBI\(\Delta\) (\(A, N, s, w/2, \tau\)).
4: \hspace{2em} JACOBI\(\Delta\) (\(A, N, (s + w/2), w/2, \tau\)).
5: \hspace{2em} JACOBI\(\nabla\) (\(A, N, (s + w/4) + 1, w/2, \tau\)).
6: \hspace{2em} JACOBI\(\Delta\) (\(A, N, (s + w/4), w/2, \tau + w/4\)).
7: \hspace{1em} \textbf{else}
8: \hspace{2em} \(p \leftarrow \tau \mod 2\).
9: \hspace{2em} \(q \leftarrow (\tau + 1) \mod 2\).
10: \hspace{2em} \(A[p][s \mod N] \leftarrow \frac{(A[q][(s - 1) \mod N] + A[q][s \mod N] + A[q][(s + 2) \mod N])}{3}\).
11: \hspace{2em} \(A[p][(s + 1) \mod N] \leftarrow \frac{(A[q][s \mod N] + A[q][(s + 2) \mod N])}{3}\).
12: \hspace{1em} \textbf{end if}
13: \textbf{end function}

And when \(N_1 \leq \lambda\), we have \(B \log_{1/b} N_1 \leq B \log_{1/b} \lambda < \lambda^{1/(1+\epsilon)} < \lambda\) by another application of lemma 4.19 because \(\lambda \geq (dB \log_{1/b} B)^{1+\epsilon}\).

Therefore, we have that \(V_1(N) = O(Z_1(N))\) and \(V_2(N) = O(Z_2(N))\) where

\[
Z_1(N) = \begin{cases} 
\Theta \left( N^{2/(1+\delta)} \right) + 3Z_1(N/2) + Z_2(N/2) & \text{if } N > \lambda \\
\Theta(\lambda^2) & \text{if } N \leq \lambda 
\end{cases}
\]

\[
Z_2(N) = \begin{cases} 
\Theta \left( N^{2/(1+\delta)} \right) + 3Z_2(N/2) + Z_1(N/2) & \text{if } N > \lambda \\
\Theta(\lambda^2) & \text{if } N \leq \lambda 
\end{cases}
\]

By manually solving the recursions for \(Z_1(N)\) and \(Z_2(N)\) simultaneously, we get that \(Z_1(N) = O(N^2)\) and \(Z_2(N) = O(N^2)\).

Finally, we have that \(V_3(N) = O(Z_3(N))\), where

\[
Z_3(N) = \begin{cases} 
\Theta \left( N^{2/(1+\delta)} \right) + 2V_1(N/2) + 2V_2(N/2) & \text{if } N > \lambda \\
\Theta(\lambda^2) & \text{if } N \leq \lambda 
\end{cases}
\]

\[= O \left( N^2 \right) \].
Therefore, $\rho_\eta(W_{\text{Jacobi},N,\lambda}) = O(T_3(N) + U_3(N) + V_3(N)) = \Theta(N^2)$. Since $\rho_\eta(W_{\text{Jacobi},N,\lambda}) = O(N^2) = O(R_\eta(N))$, \textsc{Jacobi} is optimally progressing and optimally cache-adaptive. 

### 6.3 Optimal LCS and Edit Distance

The \textsc{Recursive-LCS} dynamic programming algorithm of Chowdhury and Ramachandran [21] is described as the composition of two recursive functions. We describe a high-level description of these mutually functions when the size of $|X| = |Y| = N$. A complete exposition can be found in [21].

The first function, \textsc{LCS-Output-Boundary}, is a recursive procedure that computes the LCS length at the boundary of the subproblem being considered. \textsc{LCS-Output-Boundary} on a problem of size $N$

(i) makes 4 recursive calls on subproblems of size $N/2$;

(ii) and besides recursive calls makes $O(1)$ additional memory references.

The second function, \textsc{Recursive-LCS}, is a recursive algorithm that besides computing the LCS length, computes an actual LCS. \textsc{Recursive-LCS} on a problem of size $N$

(i) makes 3 recursive calls to \textsc{Recursive-LCS} on subproblems of size $N/2$;

(ii) makes 3 calls to \textsc{LCS-Output-Boundary} on subproblems of size $N/2$;

(iii) and makes a linear scan of size $\Theta(N)$.

\textsc{Recursive-LCS} is optimal in the DAM model and unlike previous algorithms, it does not require a tall cache assumption.

**Theorem 6.3.** Assume that $B \geq 4$. Pick an $\epsilon \in (0,0.1)$ arbitrary close to 0, and let $d = 3(1 + \epsilon)$ and $\lambda = (dB \log_2 B)^{1+\epsilon}$. For all $\lambda$-tall memory profiles, the cache-oblivious \textsc{Recursive-LCS} and \textsc{Edit-Distance} algorithms [21] are optimally progressing and optimally cache-adaptive among all algorithms that execute the $\Theta(N^2)$ operations needed to implement the type of computation defined by eq. (5.3).

**Proof.** Both the \textsc{Recursive-LCS} and \textsc{LCS-Output-Boundary} functions have linear space complexity as they recycle the auxiliary space required by the dynamic programming table.

By lemma 5.24, $\rho_\psi(\square_X) = \Theta(X^2)$ and $R_\psi(N) = \Theta(N^2)$ constitute a progress bound for the LCS problem among all algorithms that execute the $\Theta(N^2)$ operations needed to implement the type of computation defined by eq. (5.3).
Since both of the subroutines of the algorithm compute the LCS length, the progress bound applies to both of the functions.

We apply theorem 4.14 by allowing \( \lambda = (dB \log B)^{1+\epsilon} \). Suppose \( A_1, A_2 \) signify the LCS-\textsc{Output-Boundary} and \textsc{Recursive-LCS} functions respectively.

We first solve \( T_1, U_1 \) and \( V_1 \). Since \( c_1 = 0 \) and LCS-\textsc{Output-Boundary} does not have a subcall to \textsc{Recursive-LCS}, we have that:

\[
U_1(N) = \begin{cases} 
\Theta(\rho(\Box N)) + 4U_1(N/2) & \text{if } N = \Omega(\lambda) \text{ and } N^0 = \Omega(\lambda) \\
4U_1(N/2) & \text{if } N = \Omega(\lambda) \text{ and } N^0 \neq \Omega(\lambda) \\
0 & \text{if } N \neq \Omega(\lambda);
\end{cases}
\]

As for \( \mathcal{T}_1(N) \), we have that

\[
\mathcal{T}_1(N) = \begin{cases} 
\max \{ \Theta(N^2), 4\mathcal{T}(N/2) \} & \text{if } \lambda < N \\
\Theta(\lambda^2) & \text{if } N \leq \lambda;
\end{cases}
\]

By another application of lemma 4.19 because \( \lambda \geq (dB \log B)^{1+\epsilon} \).

Therefore, \( V_1(N) = O(Z_1(N)) \) where

\[
Z_1(N) = \begin{cases} 
\Theta(N^{2/(1+\delta)}) + 4Z_1(N/2) & \text{if } N > \lambda \\
\Theta(\lambda^2) & \text{if } N \leq \lambda.
\end{cases}
\]

\( = O(N^2) \) by applying the Master method.
We have that $c_2 = 1$. We now solve $U_2$:

\[
U_2 = \begin{cases} 
\Theta(N^2) + 3U_2(N/2) + 3U_1(N/2) & \text{if } N = \Omega(\lambda) \\
0 & \text{otherwise.}
\end{cases}
\]

\[
= \begin{cases} 
\Theta(N^2) + 3U_2(N/2) & \text{if } N = \Omega(\lambda) \\
0 & \text{otherwise.}
\end{cases}
\]

As for $V_2(N)$ we get

\[
V_2(N) = \begin{cases} 
3V_2(N/2) + 3V_1(N/2) & \text{if } B \log_{1/b} N = \Omega(\lambda) \text{ and } N^1 > B \\
3V_2(N/2) + 3V_1(N/2) + \Theta((B \log_{1/b} N)^2) & \text{if } B \log_{1/b} N = \Omega(\lambda) \text{ and } N^1 \leq B \\
0 & \text{otherwise.}
\end{cases}
\]

Note that when $N^1 \leq B$, $B \log_2 N \leq B \log_{1/b} B$. Because $\lambda \geq (dB \log_2 B)^{1+\epsilon}$, we have that $B \log_2 N \neq \Omega(\lambda)$. Therefore the middle case in $V_2(N)$ never happens.

\[
V_2(N) = \begin{cases} 
3V_2(N/2) + 3V_1(N/2) & \text{if } B \log_{1/b} N = \Omega(\lambda) \text{ and } N^1 > B \\
0 & \text{if } B \log_{1/b} N \neq \Omega(\lambda).
\end{cases}
\]

\[
= O(N^2) \quad \text{by applying the Master method.}
\]

Finally, we show that $T_2(N) = \Theta(N^2)$.

\[
T_2(N) = \begin{cases} 
\max \{ \Theta(N^2), 3T_2(N/2) + 3T_1(N/2) \} & \text{if } N > \lambda \\
\Theta(\lambda^2) & \text{if } N \leq \lambda.
\end{cases}
\]

We also have that

\[
T_2(N) < X(N) = \begin{cases} 
3X(N/2) + \Theta(N^2) + 3T_1(N/2) & \text{if } N > \lambda \\
\Theta(\lambda^2) & \text{if } N \leq \lambda.
\end{cases}
\]

\[
= \begin{cases} 
3X(N/2) + \Theta(N^2) & \text{if } N > \lambda \\
\Theta(\lambda^2) & \text{if } N \leq \lambda.
\end{cases}
\]

\[
= O(N^2).
\]
Hence, we have concluded that \( T_2(N) + U_2(N) + V_2(N) = \Theta(N^2) \). Therefore, \( \rho_\psi(W_{LCS,N,\lambda}) = O(N^2) \).

Since \( \rho_\psi(W_{LCS,N,\lambda}) = O(N^2) = O(R_\psi(N)) \), \textsc{Recursive-LCS} is optimally progressing and optimally cache-adaptive. Since \textsc{Edit-Distance} has the same progress bound, the above argument holds for it as well.

### 6.4 Optimal Matrix Transpose

**Definition 6.4.** Given a matrix \( A \), the **matrix transpose** problem is concerned with computing the matrix \( A^T \), where \( [A^T]_{ij} = [A]_{ji} \).

We describe a cache-oblivious algorithm, \textsc{M-Transpose}, for the matrix transpose problem. For a matrix \( A \), allow \( A_1, A_2, A_3, A_4 \) to be the four sub-quadrants of \( A \). We have that the transpose of \( A \), \( A^T \), satisfies

\[
A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \Rightarrow A^T = \begin{pmatrix} A_1^T & A_3^T \\ A_2^T & A_4^T \end{pmatrix}. \tag{6.1}
\]

The \textsc{M-Transpose} algorithm is a variant of the cache-oblivious algorithm of Frigo et al. [31], see algorithm 6. Like the algorithm in [31], \textsc{M-Transpose} requires a tall cache of size \( \Theta(B^2) \) to be optimal in the DAM model and incurs \( \Theta(N/B) \) I/Os.

**Algorithm 6** The cache-oblivious matrix transpose [31]. The algorithm writes \( A^T \) into the output matrix \( B \).

1: **function** \textsc{M-Transpose}(\( A, B \))
2: \hspace{1em} **if** size(\( A \)) = 1 \hspace{1em} **then** Write \( A \) into \( B \).
3: \hspace{1em} **else**
4: \hspace{2em} \textsc{M-Transpose}(\( A_1, B_1 \)).
5: \hspace{2em} \textsc{M-Transpose}(\( A_2, B_3 \)).
6: \hspace{2em} \textsc{M-Transpose}(\( A_3, B_2 \)).
7: \hspace{2em} \textsc{M-Transpose}(\( A_4, B_4 \)).
8: \hspace{1em} **end if**
9: **end function**

**Theorem 6.5.** For all \( \Theta(B^2) \)-tall memory profiles, the \textsc{M-Transpose} algorithm is optimally progressing and cache-adaptive and it takes \( \Theta(N/B) \) I/Os to finish.
Proof. Let \( \rho_r(\square_S) = S/B \) be the duration of a square \( \square_S \) and \( R_r(N) = N \) be the size of the input. \( \rho_r \) is trivially monotone and square-additive. Furthermore, since all algorithms that solve all inputs of size \( N \), must read the input, \( \rho_r \) and \( R_r \) constitute a progress bound for the matrix transpose problem.

M-TRANSPOSE is an \((a,b,c)\)-regular algorithm with \( c = 0 \) and \( a = 4 = 1/b \). Therefore, by theorem 4.21 it is optimally progressing and cache-adaptive. Note that there exists an \( \epsilon \in (0,0.1) \) such that \( \max\{\Theta(B^2), (3(1 + \epsilon)B \log_4 B)^{1+\epsilon}\} = \Theta(B^2) \).

The I/O complexity of M-TRANSPOSE is evident from the analysis of Frigo et al. [31]. \( \square \)
Chapter 7

Suboptimal Cache-Adaptive Algorithms

In this chapter, we establish that cache-obliviousness does not always lead to cache-adaptivity. We prove that a variation of the cache-oblivious naïve matrix multiplication algorithm of Frigo et al. [31], MM-Scan, which is optimal in the DAM model, is a $\Theta(\log)$ factor away from being optimal in the cache-adaptive model.

In section 7.2, we show that the analytic techniques of chapter 4 can be used to analyze algorithms that don’t fit the Akra-Bazzi form. As an example, we show that the cache-oblivious FFT algorithm of Frigo et al. [31] is a $O(\log \log N)$ factor away from being optimal when solving problem instances of size $N$.

7.1 Matrix Multiply: A Tale of Two Algorithms

The cache-oblivious matrix multiplication algorithm of Frigo et al. [31], MM-INPLACE has another variation, MM-SCAN (see algorithm 7), which performs the additions different than MM-INPLACE.

Both algorithms divide each input matrix into four submatrices and perform eight recursive calls to compute submatrix products. Both algorithms run in $O(N^{3/2}/\sqrt{MB})$ I/Os in the DAM model, which is optimal [34, 36]. Both algorithms require a tall cache of $\Theta(B^2)$.

Remarkably, only MM-INPLACE is optimally progressing and cache adaptive; and MM-SCAN is a $\Theta(\log(N/B^2))$ factor away from being optimal in the cache-adaptive model.

The two matrix multiplication algorithms differ in how they combine the eight matrix sub-products. Algorithm MM-SCAN adds the eight matrix sub-products in one final linear scan, yielding a recurrence of $T(N) = 8T(N/4) + \Theta(1 + N/B)$ in DAM. Algorithm MM-INPLACE computes the eight matrix sub-products “in place,” adding the results of elementary
multiplications into the output matrix, and yielding a recurrence of \( T(N) = 8T(N/4) + O(1) \) in DAM. Both recurrences have the same asymptotic solution in DAM.

**Algorithm 7** The cache-oblivious matrix multiply with \( \Theta(1 + N/B) \) linear scan [31].

Let \( A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \), \( B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix} \), \( X = \begin{pmatrix} X_1 & X_2 \\ X_3 & X_4 \end{pmatrix} \) and \( Y = \begin{pmatrix} Y_1 & Y_2 \\ Y_3 & Y_4 \end{pmatrix} \).

1: function \( \text{MM-Scan}(N, A, B) \)
2: if \( N = O(1) \) then
3:     return \( A \times B \)
4: else
5:     \( X_1 \leftarrow \text{MM-Scan}(N/2, A_1, B_1) \)
6:     \( X_2 \leftarrow \text{MM-Scan}(N/2, A_1, B_2) \)
7:     \( X_3 \leftarrow \text{MM-Scan}(N/2, A_3, B_1) \)
8:     \( X_4 \leftarrow \text{MM-Scan}(N/2, A_3, B_2) \)
9:     \( Y_1 \leftarrow \text{MM-Scan}(N/2, A_2, B_3) \)
10:    \( Y_2 \leftarrow \text{MM-Scan}(N/2, A_2, B_4) \)
11:    \( Y_3 \leftarrow \text{MM-Scan}(N/2, A_4, B_3) \)
12:    \( Y_4 \leftarrow \text{MM-Scan}(N/2, A_4, B_4) \)
13:                                                                                       ▷ Linear scan
14:    \( C \leftarrow X + Y \)
15:    return \( C \)
16: end if
17: end function

**Theorem 7.1.** On all \( \Theta(B^2) \)-tall memory profiles, the CO matrix multiplication algorithm \( \text{MM-Scan} \) is a \( \Theta(\log(N/B^2)) \) factor away from being optimal when solving problem instances of size \( N \).

**Proof.** By lemma 5.12, we have that \( \rho_\mu(\square X) = \Theta(X^{3/2}) \) and \( R_\mu(N) = \Theta(N^{3/2}) \) constitute a progress bound for the naïve matrix multiplication problem.

\( \text{MM-Scan} \) is an \((8, 1/4, 1)\)-regular algorithm. Hence, by theorem 4.21 it is a \( \Theta(\log_4(N/B^2)) \) factor away from being optimally progressing.

Since \( \text{MM-INPLACE} \) is optimally progressing, \( \text{MM-Scan} \) requires a speed augmentation of \( \Theta(\log_4(N/B^2)) \) to be able to beat \( \text{MM-INPLACE} \) in solving problem instances of size \( N \).
MM-Scan is a $\Theta(\log(N/B^2))$ factor away from being optimal.

7.2 Cache-Oblivious Fast Fourier Transform

We present the pseudo-code for the cache-oblivious FFT algorithm of Frigo et al. [31], CO-FFT (see algorithm 8). Note that in the base case, when $N = O(1)$, the algorithm computes the DFT directly. For a complete description we refer the interested reader to [31].

CO-FFT is not a GR algorithm because its recursive calls are on subproblems of size $\sqrt{N}$, rather than on subproblems a constant factor smaller. Consequently, a square $S$ may contain subproblems of size only $\Theta(\sqrt{|S|})$ instead of $\Theta(|S|)$, so the charging scheme of theorem 4.14 does not work “out of the box”.

<table>
<thead>
<tr>
<th>Algorithm 8 The CO-FFT algorithm [31].</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: function CO-FFT($R$, $N$)</td>
</tr>
<tr>
<td>2: Pretend that the input is a row-major $\sqrt{N} \times \sqrt{N}$ matrix $R$. Perform an in-place matrix transpose operation by calling M-TRANSPOSE($R$) (see algorithm 6). ▷ At this stage, the inner sum corresponds to a DFT of the $\sqrt{N}$ rows of the transposed matrix.</td>
</tr>
<tr>
<td>3: for each row $R_i$ of the matrix do</td>
</tr>
<tr>
<td>4: CO-FFT($R_i$, $\sqrt{N}$).</td>
</tr>
<tr>
<td>5: end for</td>
</tr>
<tr>
<td>6: A linear scan to multiple each element of $R$ by a by the twiddle factors (can be computed on the fly).</td>
</tr>
<tr>
<td>7: M-TRANSPOSE($R$), so that the inputs to the next stage are arranged in contiguous locations.</td>
</tr>
<tr>
<td>8: for each row $R_i$ of the matrix do</td>
</tr>
<tr>
<td>9: CO-FFT($R_i$, $\sqrt{N}$).</td>
</tr>
<tr>
<td>10: end for ▷ Compute $\sqrt{N}$ DFTs of the rows of the matrix recursively.</td>
</tr>
<tr>
<td>11: M-TRANSPOSE($R$), in place so as to produce the correct output order.</td>
</tr>
<tr>
<td>12: end function</td>
</tr>
</tbody>
</table>

However, a square $S$ must either intersect a linear scan of size $\Omega(|S|)$ or contain enough subproblems of size $\Omega(\sqrt{|S|})$ so that their total size sums to $\Theta(|S|)$. Lemma 5.34 shows that $\rho_\phi(\square_X) = \Theta(X \log X)$ and $R_\phi(N) = \Theta(N \log N)$ constitute a progress bound for Cooley-Tuckey FFT algorithms. We divide the progress of $S$ among all the subproblems contained
in \( S \). CO-FFT uses the cache-oblivious matrix-transpose algorithm, M-Transpose (algorithm 6), as a subroutine, so it has a tall-cache requirement, i.e. \( H(B) = \Theta(B^2) \).

**Theorem 7.2.** On all \( \Theta(B^2) \)-tall memory profiles, CO-FFT is a \( \Theta(\log \log (N/B^2)) \) factor away from being optimally progressing, and \( O(\log \log (N/B^2)) \) factor away from being optimally cache-adaptive.

The following lemma is a modified version of lemma 4.11 tailored for the specific application of the CO-FFT algorithm. We prove lemma 7.3 at the end of this section.

**Lemma 7.3.** If \( M \) is an \( N \)-fitting profile for CO-FFT, then every square, \( S \), of \( M \) satisfies at least one of the following two properties

(i) \( S \) encompasses \( k \) consecutive executions of CO-FFT on subproblems of size \( \ell \), where \( k\ell = \Theta(|S|) \).

(ii) \( S \) overlaps a linear scan of size \( \Omega(|S|) \).

**Proof of Theorem 7.2**

Consider a “canonical bad profile” for CO-FFT which has large boxes whenever CO-FFT performs a linear scan. The recursion for this profile is then

\[
M_{\text{FFT},N} = \begin{cases} 
\Box_N \| M_{\text{FFT},\sqrt{N}}^N \| \Box_N \| M_{\text{FFT},\sqrt{N}}^N \| \Box_N & \text{if } N \geq \Theta(B^2) \\
\Theta(\Box_{B^2}) & \text{otherwise.}
\end{cases}
\]

We have that \( \rho(\Box_N) = \Theta(N \log N) \), Therefore \( \rho(M_{\text{FFT},N}) \) satisfies the recurrence

\[
\rho(M_{\text{FFT},N}) = \begin{cases} 
2\sqrt{N}\rho(M_{\text{FFT},\sqrt{N}}^N) + \Theta(N \log N) & \text{if } N \geq \Theta(B^2) \\
\Theta(B^2 \log B) & \text{otherwise,}
\end{cases}
\]

which solves to \( \rho(M_{\text{FFT},N}) = \Theta(N \log N \log \log(N/B^2)) \). Thus, CO-FFT can be at least a \( \log \log(N/B^2) \) factor away from being optimally progressing.

To prove that this bound is tight, let \( W_{\text{FFT},N} \) be CO-FFT’s worst case profile for inputs of size \( N \). We will show that \( \rho(W_{\text{FFT},N}) = O(N \log N \log \log(N/B^2)) \).

Lemma 7.3 implies that every box \( S \) of \( W_{\text{FFT},N} \) either intersects a linear scan of size \( \Omega(|S|) \) or encompasses \( k \) consecutive executions of the FFT algorithm on subproblems of size \( \ell \), where \( k\ell = \Theta(|S|) \). We will account for the progress of the two cases separately.
Charging the linear-scan-overlapping squares  If a square $S$ overlaps a linear scan $L$ of size $\Omega(|S|)$ executed by the top-level invocation of CO-FFT and $L$ is the biggest linear scan overlapped by $S$, we charge it to $L$. We develop a recursive relation $U(N)$ that bounds the total progress of all linear-scan-overlapping squares for an invocation of CO-FFT on a problem instance of size $N$.

Let $N_0$ be the size of input in an invocation for CO-FFT. If $N_0 \neq \Omega(B^2)$, then because $W_{FFT,N}$ is $\Theta(B^2)$-tall, no square of $W_{FFT,N}$ can be charged to a linear scan executed in any part of the subproblem CO-FFT($N_0$), as squares are much bigger than these linear scans. Therefore, $U(N_0) = 0$.

For larger $N$, multiple squares may have their progress charged to a single linear scan of size $|L|$, but all but at most two of those squares will be contained entirely within the linear scan. Thus, the total size of all the squares charged to a single linear scan of size $|L|$ will be $\Theta(|L|)$. Suppose $S_1, \ldots, S_k$ are the squares charged to $L$. Since $\rho(\Box_X) = \Theta(X \log X) = \Omega(X)$, we must have that $\sum \rho(S_i) = O(\rho(\Box \sum |S_i|)) = O(\rho(\Box |L|))$. Thus, the progress of all the squares charged to linear scans executed by the top-level invocation of CO-FFT on a problem instance of size $N$ can be upper-bounded by $\Theta(\rho(\Box_N)) = \Theta(N \log N)$.

Therefore, the progress of all squares charged to linear scans of CO-FFT is upper-bounded by $U(N)$ where

$$U(N) = \begin{cases} 2\sqrt{N}U(\sqrt{N}) + \Theta(N \log N) & \text{if } N = \Omega(B^2) \\ 0 & \text{otherwise} \end{cases},$$

which has the solution $U(N) = O(N \log N \log \log(N/B^2))$.

Charging the subproblem-encompassing squares  Let $T(N)$ denote the total progress of all the subproblem-encompassing squares in $W_{FFT,N}$.

Consider a box $S$ encompassing CO-FFT’s execution on $k$ successive subproblems of size $\ell$. The total possible progress on $S$ is bounded by $\Theta(k\ell \log k\ell)$. However, from the structure of the CO-FFT algorithm, we must have that $k \leq l$, so $k\ell \log k\ell = \Theta(k\ell \log \ell)$, which is the total progress from a single box placed over each of the subproblems covered by $S$. Since a box that covers a subproblem must cover all subproblems of that subproblem, we can bound $T(N)$ by finding a set of non-overlapping subproblems that cover all subproblems of the input. Hence, $T(N)$ can be bounded by

$$T(N) = \max_{0 \leq i \leq \log \log \frac{N}{\ell N}} O(2^i N^{1-2^{-i}} N^{2^{-i}} \log N^{2^{-i}})$$

$$= O(N \log N) \quad (7.1)$$
Therefore, by square-additivity of $\rho_\phi$, we have that $\rho_\phi(W_{\text{FFT},N}) = O(U(N) + T(N)) = O(N \log N \log \log(N/B^2))$. Since $R_\phi = \Theta(N \log N)$, CO-FFT is a $\Theta(\log \log(N/B^2))$ factor away from being optimally progressing and a $O(\log \log(N/B^2))$ factor away from being optimally cache-adaptive.

**Proof of Lemma 7.3**

Let $M$ be an $N$-fitting profile for CO-FFT and let $S$ be a square of $M$ and let $\sigma$ be the sequence of memory references generated while solving a problem instance of size $N$ for which $M$ is $I$-fitting. We prove that $S$ must have one of the two properties (i) or (ii) with respect to $\sigma$.

Let $N'$ be such that CO-FFT can solve problems of size $\leq N'$ using at most $|S|/3B$ I/Os and $|S|$ memory. Since CO-FFT has linear space complexity, $N' = \Theta(|S|)$.

Note that every root-to-leaf path of the recursion tree must contain a subproblem whose size is in the range $[\sqrt{N'}, N']$, so we can expand the recursion tree to subproblems of size between $\sqrt{N'}$ and $N'$. Let $E_1, \ldots, E_t$ be the leaves of this partially expanded recursion tree, so that each $E_i$ corresponds to an execution CO-FFT on a subproblem of size $\ell \in [\sqrt{N'}, N']$.

**General properties of linear scans** As before, we use notation 4.7 to describe different types of linear scans in the recursive structure of CO-FFT.

If $\Phi$ is a subsequence of memory references between two complete executions of CO-FFT, then it may contain some number of $L_3$-type scans produced at the end of recursive calls, followed by an $L_2$-type scan produced between recursive calls, followed by some number of $L_1$-type scans produced at the beginning of recursive calls. If $\Phi$ consists of references in $\sigma$ before the first complete execution of CO-FFT, then it contain references from only $L_1$-type scans. If $\Phi$ follows the last complete execution of CO-FFT in $\sigma$, then it contains references from only $L_3$-type scans.

We first argue that overhead references cannot comprise a significant part of $S$. The size of linear scans in CO-FFT is $\Theta(N)$ on inputs of size $N$. When these linear scans become so small so that their size is less than $B$, it must be the case that the input size $N_1$ is $O(B)$. Thus, based on the recursive structure of the CO-FFT algorithm, the biggest sequence of consecutive overhead references in $\sigma$ can have a length of at most $B \log \log N_1 = O(B \log \log B)$. Since $M$ is $\Theta(B^2)$-tall, even the biggest sequence of consecutive overhead references cannot comprise a $\Omega(|S|)$ portion of square $S$.

**Property (i)** If the square $S$ encompasses $k$ consecutive executions of CO-FFT on subproblems of size $\ell \in [\sqrt{N'}, N']$, where $k\ell = \Theta(|S|)$ then we are done.

73
Property (ii) Suppose $S$ does encompass $k \geq 0$ consecutive executions of CO-FFT on subproblems of size $\ell \in [\sqrt{N'}, N']$, but $k\ell \neq \Omega(|S|)$. We show that $S$ must satisfy property (ii).

Note that $S$ can intersect at most two separate sequences of consecutive executions of CO-FFT on subproblems of size $\ell$, because if a third intersection occurs it must be in the middle section of $S$ and $S$ has to encompass all the $\ell$ consecutive subproblems of size $\ell$; see the structure of CO-FFT. Due to the choice of $N'$ and $\ell$, it follows that $\ell\ell = \Theta(|S|)$ which is in contradiction of our assumption that $S$ does not satisfy property (i).

Since $S$ can intersect at most two consecutive sequence of leaves of $E_1, \ldots, E_t$ of our partially expanded recursion tree (one at the beginning and one at the end) and neither sequence of leaves comprise a $\Theta(|S|)$ portion of $S$, at least $\Omega(|S|)$ I/Os of $S$ must be a contiguous sequence of memory references that does not contain a complete execution of CO-FFT. Call this subsequence $\Phi$.

Let $Z_1, Z_2, Z_3$ be the set of linear scans of type $L_1, L_2,$ and $L_3$, respectively, in $\Phi$. Since $S$ does not encompass a subproblem, the linear scans in $Z_1$ all belong to only one sequence of $L_1$-type slide-down moves on the recursion tree. Similarly, the linear scans in $Z_3$ all belong to only one sequence of $L_3$-type climb-up moves in the recursion tree.

Let $I(\cdot)$ denote the I/O complexity of a set of linear scans and allow

$$z = \max\{I(Z_1), I(Z_2), I(Z_3)\}.$$ 

Since at least $\Omega(|S|)$ of I/Os in $S$ are overlapping linear scans, we have that $z = \Omega(|S|/B)$. There are three cases to be considered.

Case of $z = I(Z_2)$ Since $\Phi$ does not encompass a subproblem, $Z_2$ is comprised of a single linear scan $\mathcal{L}$ in one $L_2$ set. The size of this linear scan is linear in the size of the invocation call. Therefore, $|\mathcal{L}/B| = \Omega(z) = \Omega(|S|/B)$ and $|\mathcal{L}| = \Omega(|S|)$. Thus, $S$ satisfies property (ii).

Case of $z = I(Z_3)$ In this case $Z_3$ is comprised of linear scans from several invocations of CO-FFT, where each invocation has produced exactly one linear scan of type $L_3$. Let $x_1 \leq \cdots \leq x_k$ be the problem sizes solved by each of these invocations.

Since the linear scans of CO-FFT are linear in the size of input call, we have that

$$I(Z_3) = \Theta(1 + x_1/B) + \ldots + \Theta(1 + x_k/B).$$

We have already argued that a sequence of consecutive overhead references can at most create $\log \log B$ I/Os. Since $I(Z_3) = \Omega(z) = \Omega(|S|/B) = \Omega(B)$, we have that overhead
references cannot comprise a significant part of $\mathcal{I}(Z_3)$. Assume that $x_b$ is the smallest problems size which produces a non-overhead linear scan in $Z_3$. We must have that

$$\Theta(x_b/B) + \Theta(x_{b+1}/B) + \ldots + \Theta(x_k/B) = \Omega(\mathcal{I}(Z_3)).$$

On the other hand, due to the structure of CO-FFT, we have that $x_i = (x_{i-1})^2$ for all $i = 2, \ldots, k$. This means that the above sum is a power series and no matter the values of $k$ and $b$, the I/O cost of the biggest linear scan dominates the sum, that is

$$\Theta(x_k/B) = \Omega(\Theta(x_b/B) + \Theta(x_{b+1}/B) + \ldots + \Theta(x_k/B))$$
$$= \Omega(\mathcal{I}(Z_3)) = \Omega(|S|/B).$$

Thus, $x_k = \Omega(|S|)$ and $S$ has property (ii).

**Case of $z = \mathcal{I}(Z_1)$** The analysis in this case is identical to the case of $z = \mathcal{I}(Z_3)$.

We have established that each square $S$ satisfies one of the two properties and the proof is complete. □
Chapter 8

Optimal Cache-Adaptive Sorting

In this chapter we study the comparison-based sorting problem. The cache-oblivious sorting algorithm of Brodal and Fagerberg [15], Lazy Funnel Sort (LFS), is a simpler variation of the Funnel Sort algorithm in [31]. LFS does not adhere to the recursive forms that we have studied thus far. Nonetheless, we prove that the LFS algorithms is optimally progressing and optimally cache-adaptive.

8.1 The Lazy Funnel Sort (LFS) Algorithm

We start with a short exposition of the LFS algorithm [15]. At the core of the LFS algorithm lies the concept of a $k$-merger, a perfectly balanced binary tree with $k$ leaves and a binary merger at each internal node. Each leaf has a sorted input stream, and the root has an output stream with capacity $k^d$, where $d \geq 2$ is a tuning parameter. The size of buffers between internal nodes are defined recursively: Consider a horizontal cut in the tree at half its full height: $D_0 = \lceil \lg(k)/2 \rceil$. The buffers between nodes of depth $D_0$ and $D_0 + 1$ have size $\lceil k^d/2 \rceil$. The subtree above depth $D_0$ is the top tree and all the subtrees rooted below are bottom trees. The sizes of the buffers in the top and bottom trees are defined recursively.

Upon each invocation, a $k$-merger merges $k^d$ elements into its output buffer. We call a complete invocation of a $k$-merger, producing $k^d$ elements in the output buffer, a round of execution of that $k$-merger. A $k$-merger together with its internal buffer is linearized in a recursive Van Emde Boas layout. First, the top subtree is laid out in a contiguous array and then all the bottom subtrees are laid out in contiguous arrays. The work flow of a $k$-merger is based on recursive calls to the underlying binary merger tree. A call is made to the root of a $k$-merger to fill its output stream by merging its two input buffers. When one of the buffers runs out of elements, a recursive call is made to the child node to fill it. For a complete description and analysis we refer the interested reader to [15].
LFS cannot use a single $N$-merger to sort the input array, since an $N$-merger would have superlinear size. Instead, LFS calls itself recursively to produce $N^{1/d}$ sorted streams of size $N^{1-1/d}$ and then merges these using an $N^{1/d}$-merger.

We use the following lemma by Brodal and Fagerberg [15].

**Lemma 8.1** (From [15]). Let $d \geq 2$. The size of a $k$-merger (excluding its output buffer) is bounded by $p k^{(d+1)/2}$ for a constant $p \geq 1$. Assuming that $M$ is $2pB^{(d+1)/(d-1)}$-tall, if a $k$-merger together with one block from each of its input buffers fit in memory, it performs $O(k^d/B + k)$ I/Os to output $k^d$ elements to its output buffer.

**Theorem 8.2.** Let $C = (4p)^{2d/(d+1)}$, where $p \geq 1$ is the constant from lemma 8.1 and allow $M$ to be any $CB^{2d/(d-1)}$-tall memory profile. An LFS algorithm based on $k$-mergers that have output buffers of size $k^d$ is optimally progressing and cache-adaptive on $M$ if $d \geq 2$.

**Proof.** By lemma 5.37, we have that $\rho_\gamma(\square S) = \Theta(S \lg S)$ and $R_\gamma(N) = \Theta(N \lg N)$ constitute a progress bound for the comparison-based sorting problem.

We show that if $M = \square_{S_1} \| \ldots \| \square_{S_j}$ is an $N$-fitting square profile for LCS, then $\rho_\gamma(M) = O(R_\gamma(N))$.

In the LFS algorithm, each input element must pass through $\Theta(\lg N)$ binary mergers to reach to the buffer at the root of the tree. As such, the total task of LFS is for all $N$ elements to climb $C(N) = \Theta(\lg N)$ binary mergers. Let $\phi(\square_{S_i})$ denote the total number of individual climbs that the LFS algorithm achieves during $\square_{S_i}$. Because $M$ is $N$-fitting for LFS we must have that

$$\sum_{i=1}^{j} \phi(\square_{S_i}) = NC(N) = \Theta(N \lg N). \tag{8.1}$$

Later in this section, we prove the following lemma.

**Lemma 8.3.** Let $C = (4p)^{2d/(d+1)}$, where $p \geq 1$ is the constant from lemma 8.1 and allow $M = \square_{S_1} \| \ldots \| \square_{S_j}$ to be a $CB^{2d/(d-1)}$-tall square memory profile. We have that the total number of individual climbs on the binary mergers that the LFS algorithm achieves during $\square_{S_i}$, $\phi(\square_{S_i})$, is $\Omega(S_i \lg S_i)$.

The statement of theorem 8.2 follows from lemma 8.3, because if $M$ is $N$-fitting for LFS,
it must be the case that \( \sum_i \phi(\square S_i) = C(N)N \). Therefore,

\[
\rho_\gamma(M) = \Theta\left( \sum_i \rho_\gamma(\square S_i) \right) \\
= O\left( \sum_i \phi(\square S_i) \right) \quad \text{by lemma 8.3 and the fact that } \rho_\gamma(\square S) = \Theta(S \lg S) \\
= O(C(N)N) = O(R_\gamma(N)),
\]

and thus LFS is optimally progressing. \(\square\)

**Proof of Lemma 8.3.** We are going to argue that certain \( x \)-mergers, which we refer to as **active trees**, are operational in each \( \square S_i \).

The recursive definition of buffer sizes also defines recursive subtrees. We roll out the recursive definition of these subtrees until we reach an \( x \)-merger such that \( x \) is the biggest value for which

\[
x^d \leq S_i/C. \tag{8.2}
\]

Remember that \( C = (4p)^{2d/(d+1)} \geq 4 \) and \( p \) is the same constant from lemma 8.1. We refer to such a subtree as an **active tree**. Note that since \( x \) is the biggest such value, we have that

\[
x^{2d} > S_i/C. \tag{8.3}
\]

An active tree is a little different than the notion of **base tree** in [15], because it is not the biggest \( x \)-merger that fits in memory during \( \square S_i \), but rather the biggest \( x \)-merger that can perform at least one round during \( \square S_i \) while fitting in memory. This difference is essential in our analysis, since we measure finished rounds of \( x \)-mergers in \( \square S_i \).

**Lemma 8.4.** Let \( C = (4p)^{2d/(d+1)} \), where \( p \geq 1 \) is the constant from lemma 8.1. If \( S_i \) is \( CB^{2d/(d-1)} \)-tall, an active tree with one block from each of its input buffers fits in a memory of size \( S_i \). We have that \( x^{d-1} > B \) and it takes an active tree \( \leq 4px^d/B \) I/Os to perform one round of operation.

By lemma 8.4, we have that one round of an active tree operations takes at most \( 4px^d/B \) I/Os. Since \( \square S_i \) contains \( S_i/B \) I/Os, we have that in \( \square S_i \), \( \alpha_i \geq S_i/4px^d \) rounds of operations of active trees finish.

We exhibit that \( \alpha_i \geq 2 \). This means that even with the worst alignment of a square and the operations of the active tree, LFS completes at least one round of operation of an **active tree** during \( \square S_i \).
We have

\[ \alpha_i \geq \frac{S_i}{4px^d} \geq \frac{C_{x^d}}{4px^d} \quad \text{because } \frac{S_i}{C} \geq x^d \text{ by eq. (8.2)} \geq \frac{C}{4p} = \frac{(4p)^{2d/(d+1)}}{4p} = (4p)^{(d-1)/(d+1)} \geq 2 \quad \text{because } d \geq 2 \text{ and } p \geq 1. \]

During each round of operation of an active tree, \( x^d \) elements climb up \( \lg x \) binary mergers. Therefore, the total number of climbs in \( \Box S_i \) by LFS, \( \phi(\Box S_i) \), is bounded from below:

\[
\phi(\Box S_i) = \Omega \left( \frac{S_i}{4px^d} \right) x^d \lg x \\
= \Omega \left( S_i \frac{1}{8dp} \lg \frac{S_i}{C} \right) \quad \text{Since } x > (S_i/C)^{1/2d} \text{ by eq. (8.3)} \\
= \Omega(S_i \lg S_i).
\]

It only remains to prove lemma 8.4.

**Proof of Lemma 8.4.** We show that the \( x \)-merger itself and the set of input buffer blocks both take at most \( S_i/4 \) memory.

The size of an active tree (an \( x \)-merger) is bounded by \( px^{(d+1)/2} \) by lemma 8.1. We have:

\[
px^{\frac{d+1}{2}} \leq p(S_i/C)^{\frac{d+1}{2d}} \quad \text{since } x \leq (S_i/C)^{\frac{1}{2d}} \text{ by eq. (8.2)} \\
\leq \frac{p}{4p} (S_i)^{\frac{d+1}{2d}} \quad \text{since } C = (4p)^{2d/(d+1)} \\
\ll S_i/4 \quad \text{since } d \geq 2.
\]

Since an \( x \)-merger has \( x \) leaves, one block from each input buffer will occupy a total of \( x \) blocks of memory. Therefore

\[
Bx \leq B(S_i/C)^{\frac{1}{2d}} \quad \text{since } x \leq (S_i/C)^{\frac{1}{2d}} \text{ by eq. (8.2)} \\
\leq (S_i/C)^{\frac{d-1}{2d}} (S_i/C)^{\frac{1}{2d}} \quad \text{since } S_i \geq CB^{2d/(d+1)} \text{ by tallness of } S_i \\
\leq (S_i/C)^{\frac{1}{2}} \\
\ll S_i/4 \quad \text{since } C \geq 4.
\]

which gives us the desired space bound.
To finish one round, an active tree needs to load the $x$-merger structure together with one block from each of its input buffers into memory which takes at most \((1 + px^{(d+1)/2}/B) + x\) I/Os.

By definition, the $x$-merger outputs $x^d$ elements to its output buffer. Since reading/writing of data is performed in batches and is I/O efficient, we have that reading and writing back these elements takes at most \(2(1 + x^d/B)\) I/Os.

We show that \(x^{d-1} > B\):

\[
x^{d-1} > \left( (S_i/C)^{1/2} \right)^{d-1}\quad \text{since } x > (S_i/C)^{1/2} \text{ by eq. (8.3)} \quad (8.4)
\]
\[
\geq (S_i/C)^{d-1}\quad \text{since } S_i \geq CB\frac{x^d}{2}\text{ by tallness of } S_i
\]
\[
\geq B.
\]

Equation (8.4) means \(x < x^d/B\). On the other hand, since \(d \geq 2\), we have that \(px^{(d+1)/2}/B \leq px^d/B\).

Therefore, the total number of I/Os required for an active tree to finish one round of its operation is bounded by

\[
2(1 + x^d/B) + (1 + px^{(d+1)/2}/B) + x \leq 3 + 3px^d/B
\]
\[
\leq 4px^d/B\quad \text{since } x^d/B > x \geq 2.
\]

Note that \(x \geq 2\), because there can be no 1-merger.
Chapter 9

Page Replacement Policies in the Cache-Adaptive Model

This section gives competitive analyses for page replacement when the size of memory changes over time. Since we measure time in terms of I/Os, two paging algorithms may become out-of-sync when one has a page fault and the other has a page hit. Thus, two algorithms may have wildly different amounts of memory available to them when they service the same request. This section shows how to deal with this alignment issue.

We first show that LRU with resource augmentation is competitive with OPT in section 9.1. Then we prove that Belady’s Longest Forward Distance (LFD) policy [8] is optimal when the size of cache changes over time in section 9.2.

We start by inspecting a formal view of a page replacement policy in the cache-adaptive model. We model the state of the memory as the set $C$ of pages currently in the memory and the size of the memory. Between each page request, the page replacement algorithm may perform some sequence of I/Os and specify a set of pages to be evicted from the memory (since memory changes, we may need to evict more than one page). We name an I/O operation by an ordered pair $(D, p)$, where $D$ is the set of pages to be deleted from memory and $p$ is the page to be loaded after performing the deletions. Note that when memory grows, $D$ may be the empty set. As in the standard model, no block from disk can have more than one paged copy in cache at a time.

In the CA model, a page replacement policy $P$ is a sequence of I/O operations $P = (D_1, p_1), (D_2, p_2), \ldots$ for a sequence of page requests $\sigma = (\sigma_1, \sigma_2, \ldots)$. A page replacement policy is feasible if $\sigma_i$ is always in memory when it is accessed and $C$ always fits in memory.

A policy is on-demand if: 1) When $\sigma_{i+1}$ is not in memory after $\sigma_i$ is accessed, it immediately performs one I/O to bring $\sigma_{i+1}$ into memory; 2) The policy makes no other I/Os. We require that all policies are feasible.
9.1 Constant Competitiveness of a Resource-Augmented LRU

We utilize square profiles, lemma 3.3 and an inductive charging approach to prove competitiveness of LRU with a variable-sized cache. We show that LRU with 4-memory and 4-speed augmentation is competitive with the optimal page replacement algorithm. This shows that, given some resource augmentation, real LRU-based systems can implement the CA model.

**Notation 9.1.** For a sequence of page accesses $\sigma = (\sigma_1, \ldots, \sigma_n)$, memory profile $m(t)$, and page replacement algorithm $P$, let $C_P(m, \sigma)$ be the number of I/Os required to process $\sigma$ in $m$ while using page replacement algorithm $P$. Let $C(m, \sigma)$ denote $C_{OPT}(m, \sigma)$.

The following lemma enables us to convert simultaneously between LRU and OPT and square and general profiles.

**Lemma 9.2.** Let $m$ be any memory profile. Let $m'$ be the inner square profile of $m$. Let $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ be any sequence of page accesses. Then $C_{LRU}(m'_4, \sigma) \leq 4C_{OPT}(m, \sigma)$.

**Proof.** Since OPT and LRU can never place more than one page in cache per time step, we may assume without loss of generality that $m(t+1) \leq m(t) + 1$ for all $t$, that is the memory profile is *usable*. Let $0 = t_0 < t_1 < \ldots$ be the inner square boundaries of $m$.

We prove by induction on $i$ that for all page sequences $\sigma$, if $C_{OPT}(m, \sigma) \in [t_i+1, t_{i+2})$, then $C_{LRU}(m'_4, \sigma) \leq 4t_{i+1}$.

**Base case** $i = 0$ We argue that if OPT can process $\sigma$ in the first two intervals of $m$, then LRU can process it in the first interval of $m'_4$. Suppose $C_{OPT}(m, \sigma) < t_2$. From lemma 3.3, $t_2 = t_2 - t_0 = (t_2 - t_1) + (t_1 - t_0) \leq 2(t_1 - t_0) + (t_1 - t_0) = 3(t_1 - t_0) = 3t_1$. Since the cache is empty at time $t_0 = 0$, this implies that $\sigma$ can refer to at most $t_2 \leq 3t_1$ distinct pages. Since, during interval $[0, 4t_1)$, profile $m'_4$ always has size at least $4t_1$ pages, LRU can load all the pages referenced by $\sigma$ into memory, and this will require at most $3t_1$ I/Os. Thereafter, no further I/O will be required. Thus $C_{LRU}(m'_4, \sigma) < 4t_1$.

**Inductive step** The inductive assumption is that for all $\sigma'$, if $C_{OPT}(m, \sigma') \in [t_i+1, t_{i+2})$, then $C_{LRU}(m'_4, \sigma') \leq 4t_{i+1}$. Suppose that $C_{OPT}(m, \sigma) \in [t_{i+2}, t_{i+3})$. Let $\sigma'$ be the prefix of $\sigma$ that OPT services in $[t_0, t_{i+2})$. Consequently, $C_{OPT}(m, \sigma') < t_{i+2}$. By the inductive hypothesis, $C_{LRU}(m'_4, \sigma') \leq 4t_{i+1}$.

Let $\sigma''$ be the remainder of $\sigma$, i.e. $\sigma = \sigma'||\sigma''$. Observe that, since OPT can process $\sigma''$ in $t_{i+3} - t_{i+2} \leq 2(t_{i+2} - t_{i+1})$ time steps and with a cache whose initial size is at most $2(t_{i+2} - t_{i+1})$, $\sigma''$ can contain references to at most $4(t_{i+2} - t_{i+1})$ distinct pages.

82
We now only need to show that LRU can process any suffix of $\sigma''$ in interval $[4t_{i+1}, 4t_{i+2})$. The memory available during this interval in $m'_{4,4}$ is $4(t_{i+2} - t_{i+1})$. Thus LRU can load all the distinct pages referenced by $\sigma''$ into memory, which will require at most $4(t_{i+2} - t_{i+1})$ I/Os. Thereafter, it can serve all the page requests in $\sigma''$ with no further I/O. \qed

**Theorem 9.3.** LRU with 4-memory and 4-speed augmentation always completes sooner than OPT.

*Proof.* Take $\sigma, m$, and $m'$ as in lemma 9.2. Since $m'_{4,4}$ always has less memory than $m_{4,4}$, $C_{\text{LRU}}(m_{4,4}, \sigma) \leq C_{\text{LRU}}(m'_{4,4}, \sigma)$. Thus, by lemma 9.2, $C_{\text{LRU}}(m_{4,4}, \sigma) \leq 4C_{\text{OPT}}(m, \sigma)$. \qed

The following lemma shows that LRU requires speed augmentation to be competitive. Sleator and Tarjan have previously proven that LRU requires memory augmentation to be constant competitive [49]. Therefore theorem 9.3 is, up to constants, the best possible.

**Lemma 9.4.** There exists a memory profile $m(t)$ and a page request sequence $\sigma$, such that a non-speed augmented LRU performs arbitrarily worse than OPT.

*Proof.* Consider the following memory profile for LRU:

$$m_{\text{LRU}}(t) = \begin{cases} 
cM & t \leq cM + 1 \\
2cM + 1 - t & cM + 1 < t < 2cM - 4 \\
5 & 2cM - 4 \leq t
\end{cases}$$

Assume that memory augmentation of LRU is $c$ where $c \geq 2$, i.e. LRU starts with $cM$ memory and OPT starts with $M$ memory. Thus $m_{\text{OPT}}(t) = \lfloor \frac{m_{\text{LRU}}(t)}{c} \rfloor$. Let $H = M - \lfloor \frac{(c-1)M - 4}{c} \rfloor$. Consider a sequence of page requests $\sigma = \{\sigma_i\}_{i \geq 1}$ as follows:

$$\sigma = (a_1, \ldots, a_{cM}, a_{cM+1}, a_1, \ldots, a_{M-1}, a_M, \ldots, a_{cM-5}, a_1, \ldots, a_{H-1}, a_1, \ldots, a_{H-1}, a_1, \ldots).$$

LRU and OPT both fault on all the first $cM + 1$ page requests because all of them are new pages. Furthermore, LRU will also fault on page requests $\sigma_{cM+2} = a_1, \ldots, a_{2cM-4} = a_{cM-5}$. This happens because at page request $\sigma_{cM+1} = a_{cM+1}$, LRU will evict $a_1$ and therefore must bring it again at $\sigma_{cM+2}$. The same thing happens for all requests in $\sigma_{cM+2}, \ldots, a_{2cM-4}$. When accounted in the memory profile function, LRU will have a memory size of 5 at time $2cM - 4$. Since OPT has a full insight into future, it will keep pages $a_1, \ldots, a_{M-1}$ in its memory and thus will not fault on page requests $\sigma_{cM+2} = a_1, \ldots, \sigma_{(c+1)M+1} = a_{M-1}$. However, on requests $\sigma_{(c+1)M+2} = a_M, \ldots, \sigma_{2cM-4} = a_{cM-5}$, OPT will fault on $cM - 5 - M$ pages. At time step $2cM - 4$, the memory of OPT has decreased to $H = M - \lfloor \frac{(c-1)M - 4}{c} \rfloor$ pages. Again, because of the insight into future, OPT could keep pages $a_1, \ldots, a_{H-1}$ in memory. So OPT will not fault on any pages after $\sigma_{2cM-4}$. Since $H \geq \frac{M+4}{c} + 1 > 5$, LRU, will fault on all page requests after $\sigma_{2cM-4}$. Therefore, without speed augmentation, LRU is not cache-adaptive. \qed
9.2 Optimal (Offline) Page Replacement in the CA Model

Now we examine a replacement policy known to be optimal in the DAM model. Belady’s algorithm [8] is an offline policy that evicts the page with Longest Forward Distance (LFD). In other words, LFD evicts the page whose next request comes latest. We prove that LFD is an optimal page replacement policy in the CA model.

Even though we require that all policies are feasible, but with changing cache size it may be that there isn’t an optimal on-demand policy. As such, the proof needs to show that LFD is optimal among all feasible policies in the CA model.

We begin with a method to transform a non on-demand policy into an on-demand one, thus showing that there exists an optimal adaptive on-demand policy. This method simplifies the proof that Belady’s algorithm is optimal.

Lemma 9.5 (The Swap Method). Suppose a page-replacement policy $P$ is on-demand up until serving page request $\sigma_i$. There exists a policy that agrees with $P$ up to page request $\sigma_i$, has the same number of I/Os as $P$ and is on-demand up until serving page request $\sigma_{i+1}$.

Proof. If $P$ is on-demand for $\sigma_{i+1}$, then we are done. Otherwise, there is at least one I/O performed by $P$ between $\sigma_i$ and $\sigma_{i+1}$. If $\sigma_{i+1}$ is already in cache when $\sigma_i$ is accessed, all of these I/Os can be moved after $\sigma_{i+1}$ and we are done—the resulting policy is on-demand until $\sigma_{i+1}$, is feasible, and has the same number of I/Os.

Otherwise, let $(D', \sigma_{i+1})$ be the first I/O after $\sigma_i$ to load $\sigma_{i+1}$. We want $(D', \sigma_{i+1})$ to be the first and only I/O after $\sigma_i$ is accessed. Let $(F, p)$ and $(E, q)$ be consecutive I/Os, and
let $E[q/p]$ denote replacing any occurrence of $p$ with $q$ in $E$. We show that we can swap $(D', \sigma_{i+1})$ with the I/O before it using the **swap method**:

$$(F, p), (E, q) \rightarrow \begin{cases} (F, q), (E[q/p], p) & \text{if } p \notin E \\ (F, q), (E[q/p], q) & \text{otherwise} \end{cases}$$

It can be shown via case-by-case analysis (see fig. 9.1), that the swap method does not change the state of memory after $(E, q)$. Thus, the policy is still feasible. We continue to swap $(D', \sigma_{i+1})$ with the previous I/O until it occurs immediately after $\sigma_i$ is accessed. We can then move all other page accesses after $\sigma_{i+1}$, again retaining feasibility.

The resulting policy is on-demand until $\sigma_{i+1}$. We never changed any I/O before $\sigma_i$, nor did we change the number of I/Os.

**Theorem 9.6.** LFD is an optimal page replacement policy for a variable-sized cache.

**Proof.** Among all on-demand optimal algorithms, consider the algorithm, OPT, that matches LFD’s behavior for the longest time. We will create another on-demand optimal algorithm, $G$, that matches LFD’s behavior for one more step and arrive at a contradiction.

Let $\sigma_i$ be the first page request for which OPT and LFD diverge in their behavior. If memory increases, an on-demand algorithm does not need to evict any page. Thus, the divergence happens only when memory decreases or stays the same. Before processing $\sigma_i$ the two algorithms have the same pages in memory, so $\sigma_i$ must be a page miss for both of them. Since they diverge at $\sigma_i$, they replace $\sigma_i$ with different pages. Let $q$ be the page that LFD discards and $p$ be the page that OPT discards. Let $t$ be the first time after $i$ that OPT discards $q$. We will alter OPT’s behavior between $i$ and $t$ to create a new optimal algorithm $B$. $B$ will service $\sigma_i$ the same way that LFD does (by replacing $q$) and will satisfy $\text{Cost}_B(\sigma) = \text{Cost}_{OPT}(\sigma)$. $B$ is not necessary on-demand, so we use lemma 9.5 to convert $B$ into an on-demand policy $G$ with same number of I/Os as OPT.

By the definition of LFD, the next request to $p$, $\sigma_a$, comes before the next request to $q$, say $\sigma_b$. Therefore, the sequence of page requests, $\sigma$, looks like this:

$$\sigma = (\sigma_1, \ldots, \sigma_i, \ldots, \sigma_a = p, \ldots, \sigma_b = q, \ldots).$$

Policy $B$, unlike OPT, services $\sigma_i$ by discarding $q$ (like LFD). There are two general cases: Either $i < t < a$ or $t \geq a$. In both of the cases we should be careful of the effect of memory changes in the behavior of the algorithms.

1. If $i < t < a$, we need to describe how $B$ services all the requests $\sigma_x$ for $x \in (i, t)$ and for $x = t$. We will argue that $B$ and OPT converge after $x = t$. In these cases, $\sigma_x \neq p$. 85
1.a. $x \in (i, t)$: In this case $B$ misses exactly when OPT misses. Whether or not the memory changes, $B$ would be matching the behavior of OPT on every fault. OPT and $B$ would share all but one page of their memory.

1.b. $x = t$: Since $\sigma_t$ OPT discards $q$, $\sigma_t$ must be page fault for OPT and since $t < a$ and OPT and $B$ share all but one page of their memory, $\sigma_t$ must be a fault for $B$ as well. At time $t$ memory profile can only stay the same or decrease. Hence OPT would load $\sigma_t$ and discard $q \cup D$ where $D$ is the set of other possible discarded pages. At this point, $B$ stops mimicking OPT and acts in a way to converge to OPT. So $B$ discards $p \cup D$ and loads $\sigma_t$. The two algorithms would have the same memory state servicing $\sigma_t$. For the rest of sequence, $B$ will continue mimicking OPT. Since they did the same number of I/Os upto and including $\sigma_t$, their total cost would be equal.

2. If $t \geq a$. We will describe the behavior of $B$ on $x \in (i, a)$ and on $x = a$. We will argue that $B$ and OPT converge after $x = a$.

2.a. $x \in (i, a)$: The same as in case 1.a happens here too.

2.b. $x = a < t$: Since $b > a$ and OPT discarded $p$ at time step $i$, OPT will fault on $\sigma_a = p$. OPT would load $p$ and discard $D$. Since at time $a$ memory might increase, $D$ might be an empty set. Unlike OPT, $B$ would not fault on $\sigma_a$. However, $B$ would perform a non on-demand I/O ($D, q$) (delete $D$ and load $q$). This I/O causes the state of memory of $B$ and OPT to become equal and they would converge afterwards. Note that their cost is also equal.

2.c. $x = a = t$: Since $b > a$ and OPT discarded $p$ at time step $i$, OPT will fault on $\sigma_a = p$. OPT would load $p$ and discard $D \cup q$. Unlike OPT, $B$ would not fault on $\sigma_a$. However, $B$ would perform a non on-demand I/O ($D \cup p, p$) (delete $D \cup p$ and load $p$). This I/O causes the state of memory of $B$ and OPT to converge as in case 2.b.

Using lemma 9.5, we will convert the non on-demand policy $B$ into an on-demand policy $G$. $G$ is an algorithm with the same cost (number of operations) as OPT and also matches LFD at $\sigma_i$ which is a contradiction with the fact that OPT is the longest on-demand optimal algorithm which matches LFD.
Chapter 10

Foresight and Simulation of Square Profiles

In this chapter we show that, if an algorithm is optimal on square profiles then, with some foresight, we can use simulation to obtain an algorithm that is optimal on arbitrary profiles. In particular, this means that, with some additional foresight and resource augmentation, algorithms optimal in Barve and Vitter’s model [6] are optimal on arbitrary profiles.

An algorithm in the CA model takes an instance of a problem to solve, the block size, and oracular access to the memory profile in which it will execute, and generates a sequence of page requests. Thus, algorithms may have arbitrary foresight about the memory available in the future, but we also consider “$k$-prescient” algorithms that only look a limited distance into the future. The performance of an algorithm on inputs of size $N$ in profile $m$ is its worst case performance on any problem of size at most $N$.

**Definition 10.1.** In this paper, an algorithm $A$ for solving a problem $Q$ takes as input an instance, $I \in Q$, an oracle for memory profile $m$, and the block size $B$ and generates a sequence of page accesses, $\sigma = A^{m(I)}(I, B)$ (we will omit $B$ unless necessary). An algorithm is **$k$-prescient** if, during every time step $t$, it never queries oracle $m$ on a value larger than $t + km(t)$. Define

$$C(m, A, N) = \max_{I \in Q, |I| \leq N} C(m, A^{m(I)}(I)).$$

**Theorem 10.2.** If there exists a $k$-prescient algorithm that is optimal among all memory monotone algorithms on square profiles, then there exists a $2(k + 1)$-prescient algorithm that, with 4-memory and 4-speed augmentation, is optimal among all memory-monotone algorithms on all profiles.

**Proof.** Let $A$ be any such algorithm. To construct an optimal algorithm on all memory profiles, we will use prescience to simulate a square profile when answering $A$’s oracle queries.
For any input $I$ and memory profile $m$, define $S^m(I) = A^m(I)$. When, at time $t$, $A$ queries its oracle for the memory that will be available at time $t'$, $S$ must compute the inner square profile of $m$ and return $m'(t')$ to $A$. Since $A$ is $k$-prescient, we know that $t' \leq t + km(t)$. Therefore, since $m$ can only increase by one in each time step, $m(t') \leq m(t) + km(t) = (k + 1)m(t)$. Thus the inner square interval containing $t'$ is at most $(k + 1)m(t)$ long, and hence $S$ can compute $m'(t')$ by querying $m$ on inputs no larger than $2(k + 1)m(t)$. Hence $S$ is $2(k + 1)$-prescient.

We prove by contradiction that $S$ is optimal. Suppose that $S$ is not optimally cache adaptive. We will derive a contradiction that $A$ is optimal by constructing, for any augmentation factors $c_1$ and $c_2$, an algorithm, $R'$ that outperforms $A$ on inputs of a certain size on a square profile.

Let $c_1$ be any memory augmentation factor and $c_2$ be any speed augmentation factor. Since $S$ is non-optimal, there exists a memory profile $m$, integer $N$, and algorithm $R$ such that $4c_2C(m, R, N) < C(m_{4c_1, 4c_2}, S, N)$. Let $z$ be the largest query to $m$ that $R$ makes while processing any input of size $N$.

We now simulate $R$ running on $m$ as follows. Define $u(t) = m(t)$ when $t \leq z$ and $u(t) = 1$ for $t > z$. We can imagine that the first $z$ values of $m$ are effectively hard-coded into $u$. In memory profile $p$, the simulation $R'$ performs $R'^p(I) = R'^u(I)$.

We now have the following inequalities:

\[
\begin{align*}
    c_2C(m'_{4,4}, R', N) &\leq 4c_2C(m, R, N) & \text{(Def. of $R'$ and lemma 9.2)} \\
    &< C(m_{4c_1, 4c_2}, S, N) & \text{(Non-optimality of $S$)} \\
    &\leq C(m'_{4c_1, 4c_2}, A, N) & \text{(Def. of $A$ and since $m'_{4c_1, 4c_2} \leq m_{4c_1, 4c_2}$)}
\end{align*}
\]

So, even if we give $A$ $c_1$-memory augmentation and $c_2$-speed augmentation, $R'$ always finishes faster than $A$ in profile $m'_{4,4}$. This contradicts the optimality of $A$ on square profiles.

The reader may have noticed that, since $R'$ embeds a prefix of $m$ into the body of its algorithm, it may be very large. In fact, its size may depend on $N$. This is not a problem in this proof, however, because we have shown that if $S$ is not optimal, then, roughly speaking, there exists a (possibly very large) program that is faster than $A$ on at least one square profile. Since $A$ is optimal on all square profiles, no program of any size can outperform $A$ on any memory profile. Thus the contradiction is achieved.

We also note that $R'$ is no more prescient than $R$. In fact, $R'$ never queries its oracle. Thus if $A$ is competitive on all square profiles with all algorithms that make no oracle queries, then $A$ is competitive with all algorithms of arbitrary prescience on all memory profiles.

The Barve-Vitter model assumes that, when an algorithm is allocated $s$ blocks of memory, that allocation will last for exactly $s$ I/Os. Thus their model assumes memory profiles are
square. This squareness entails some prescience, though: at the beginning of an allocation of s blocks, the algorithm knows the current size of memory will not change for s I/Os. Thus, algorithms in the Barve-Vitter model are a subset of the class of 1-prescient algorithms in the CA model.

### 10.1 Square Profiles are Adequately Rich for Studying Optimality of Cache-Oblivious Algorithms

Note that a 0-prescient algorithm can still query \( m(t) \) during time \( t \). Cache-oblivious algorithms meet an even stronger condition— they do not query \( m \) at all.

**Observation 10.3.** An algorithm \( A \) is **cache oblivious** if it never queries \( m \) and does not depend on \( B \).

**Definition 10.4.** Let \( T(N, M, B) \) denote the I/O complexity of an algorithm in the DAM model. An algorithm is said to satisfy the **regularity condition** [31], if \( T(N, M, B) = O((T(N, 2M, B))) \).

We get the following theorem immediately:

**Theorem 10.5.** Let \( m \) be a memory profile and let \( m' \) be its inner square profile. Suppose that a memory-monotone cache-oblivious algorithm \( A \) finishes at time \( t \) on profile \( m \). Given a constant factor speed augmentation, \( A \) finishes no later than \( t \) on \( m' \) if either of the following holds.

1. \( A \) is given 4-memory augmentation.
2. If \( A \) satisfies the regularity condition.

**Proof.** 1. In the proof of theorem 10.2, we construct a simulator for an algorithm \( A \) that shows \( A \) is optimal for all profiles. The only purpose of the simulator is to answer \( A \)'s oracle queries. However, if \( A \) is cache-oblivious, then it makes no queries. Hence no simulator is needed: \( A \) itself is optimal on all profiles with 4-memory and 4-speed augmentation.

---

\[1\] Frigo et al. [31] showed that cache-oblivious algorithms that satisfy the regularity condition can be “ported” to systems with LRU page replacement instead of optimal page replacement. We are unaware of any CO algorithm that doesn’t satisfy this condition.
2. Since $A$ satisfies the regularity condition, giving it less memory inside each square of the inner square profile increases its running time by a constant factor $c$. Therefore, the same inductive charging argument in the proof of theorem 10.2 shows that $A$ with $4c$-speed augmentation is optimal on all memory profiles.

This theorem demonstrates that cache-oblivious algorithms are a useful tool for building cache-adaptive algorithms that do not require prescience. The theorem also demonstrates that one needs to consider only square profiles when proving that a cache-oblivious algorithm is optimally cache-adaptive.
Bibliography


