Analysis of Recursive Cache-Adaptive Algorithms

by

Andrea Lincoln

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2014

© Andrea Lincoln 2014. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Author . . . .
Department of Electrical Engineering and Computer Science
May 22, 2014

Certified by . . . .
Erik Demaine
Professor
Thesis Supervisor

Accepted by . . . .
Anantha Chandrakasan
EECS Department Head

Abstract

The performance and behavior of caches is becoming increasingly important to the overall performance of systems. As a result, there has been extensive study of caching in theoretical computer science. The traditionally studied model was the ***external-memory model*** [AV88]. In this model cache misses cost $O(1)$ and operations on the CPU are free [AV88]. In 1999 Frigo, Leiserson, Prokop and Ramachandran proposed the ***cache-oblivious model*** [FLPR99]. In this model algorithms don't have access to cache information, like the size of the cache. However, neither model captures the fact that an algorithm's available cache can change over time, which can effect its efficiency. In 2014, the ***cache-adaptive model*** was proposed [BEF$^+$14]. The cache-adaptive model is a model where the cache can change in size when a cache miss occurs [BEF$^+$14]. In more recent work, to be published, methods for analysis in the cache-adaptive context are proposed [MABM]. In this thesis we analyze the efficiency of recursive algorithms in the cache-adaptive model. Specifically, we present lower bounds on progress per cache miss and upper bounds on the total number of cache misses in an execution. The algorithms we analyze are divide and conquer algorithms that follow the recurrence $T(N) = aT(N/b) + N^c$. For divide and conquer algorithms of this form, there is a method for calculating within a constant factor the number of cache misses incurred. This provides a theorem analogues to the Master Theorem, but applicable to the cache-adaptive model.

Contents

1 Introduction **3**

2 Background **7**

2.1 External-Memory Model 7

2.2 Cache-Oblivious Model 8

2.3 Cache-Adaptive Model 9

2.4 Example: Matrix Multiplication algorithms 11

2.4.1 Traditional Computational Model 11

2.4.2 External Memory Model 11

2.4.3 Cache Oblivious Model 12

2.4.4 Cache Adaptive Model 13

3 Algorithm Performance **15**

3.1 Assumptions 15

3.2 Square Profiles 16

3.3 Progress 18

3.4 Main Theorems 20

3.5 Proofs About Average Progress 24

3.5.1 Bounding Progress 24

3.5.2 Comparing the two square profiles 26

3.6 Proving Theorems 17 and 16 27

4 Conclusion **31**

Acknowledgements

Many thanks to Erik Demaine for advising my thesis. Many thanks to Jayson Lynch with whom I did much of this work. Many thanks to Michael Bender, Erik Demaine, Roozbeh Ebrahimi, Rob Johnson, Jayson Lynch and Samuel McCauley with whom I collaborated on the related paper [MABM]. I have learned a lot from my interactions with all of you. I greatly appreciate it.

Thanks to Erik Demaine, Jayson Lynch, Adam Yedidia, Mika Braginsky and Alex Arkhipov for reading, proof reading and providing feedback on this thesis.

Chapter 1

Introduction

The standard word RAM model of computation does not account for the nonuniform cost of accessing memory. In the word RAM model only CPU opprations and pointer follows on registers of size $\log n$ bits incur a cost. However, in many practical scenarios, cache misses are the bottleneck for the program. A cache miss takes more time than an instruction. In some architectures a cache miss on a low level cache takes up to 100 times as long as a computation [Luu15]. This has led to the development of theoretical models that capture the importance of the cache.

Caching Models. Cache behavior has been extensively studied. Sometimes, this was done in an ad-hoc way. Some systems papers analyze cache misses for a given program and a given piece of hardware, an example and explanation of such analysis is in section 10.2 of the book "Quantitative System Performance" [LZGS84]. These papers approximate the number of cycles needed for a cache miss and then the approximate number of cache misses needed for a run of the algorithm being analyzed.

To formalize this idea, in 1988 Aggarwal and Vitter developed the *external-memory model* [AV88]. The external-memory model is also known as the I/O model [Dem02]. This model spawned a large amount of research in caching. The external-memory model offers a theoretical scaffolding on which one can analyze the cache performance of algorithms. Furthermore, the external-memory model is the foundation for two other models: the cache-oblivious model and the cache-adaptive model.

In the external-memory model, there is a cache of size M with cache lines of size B [Dem02]. Every time a memory address is pulled into cache, B contiguous bytes will also be pulled into mem-

ory. The cache can fit $m = M/B$ cache lines, each of which fits B words, for a total of $M = mB$ words in cache. Capital M is the number of words that fit in cache. Lower case m is the number of cache lines in cache. We can say that an algorithm takes a certain amount of time: for example it takes $O(N/B)$ to read off the N elements in order from a list and $O(\frac{N^3}{\sqrt{MB}})$ to do matrix multiplication on $N \times N$ matrices. We will describe the external-memory model in more detail in Section 2.1.

Knowing the specific cache line size and and cache size was key to designing efficient external-memory algorithms. In particular, if the cache size were different, an efficient algorithm might become very inefficient. To deal with this, Frigo, Leiserson, Prokop and Ramachandran proposed the *cache-oblivious model* [FLPR99]. In the cache-oblivious model, we assign one unit of time to every cache miss. However, the algorithm is not given access to the cache size, nor is it given access to the size of the cache line. The cache-oblivious model is basically a specialization of the external-memory model in which the algorithm isn't given access to the numbers M and B. The algorithms must be *oblivious* to the size of the cache and the size of a cache line. An algorithm's obliviousness doesn't prevent making statements about its efficiency of with respect to M and B. There have been many successful results that produce excellent efficiency even when we don't know the size of cache or the cache line size. For example, matrix multiplication can be solved in $O\left(\frac{N^3}{B\sqrt{M}}\right)$ time [FLPR99].

In the *cache-adaptive model*, the size of cache is no longer static through the run of the algorithm, instead the size of cache at a given point in time, t, is $M(t)$. There has been one paper on cache adaptivity published at this point, Cache-adaptive Algorithms [BEF+14]. The paper by Bender, Ebrahimi, Fineman, Ghasemieseh, Johnson and McCauley proposes a model where the cache can change size whenever there is a cache miss [BEF+14]. This paper builds a lot of machinery used in this thesis. In the cache-adaptive model, making statements about algorithm complexity becomes more difficult. How can one characterize how long an algorithm will take when, if an algorithm takes longer, the size of cache will change? The results of this thesis are entirely in the cache-adaptive model, working toward giving time bounds for recursive algorithms.

Previous work. In 2013 Peserico published a paper considering the case of a changing cache size [Pes13]. In Peserico's model the size of cache is a function of the number of accesses that have occurred, so the size of cache can change over the course of a single cache miss. The paper by Peserico explores cache replacement algorithms and shows that Least Recently Used (LRU) cache replacement (along with other replacement algorithms) is within a constant factor of optimal (OPT)

cache replacement in his model [Pes13].

The paper "Cache-adaptive Algorithms" proposes the cache-adaptive model [BEF$^+$14]. They show that in the cache-adaptive model the LRU with **4-*speed augmentation*** runs as fast or faster than OPT [BEF$^+$14]. Speed augmentation will be explained in Section 2.3. Additionally, they propose a notion of an algorithm being optimally cache-adaptive. In their paper they give optimally cache-adaptive algorithms for matrix multiplication, matrix transpose, Gausian elimination, all-pairs shortest paths and sorting [BEF$^+$14]. They also show that cache-obliviousness and cache-adaptivity are not equivalent. Specifically, there exist optimal cache-oblivious algorithms which are not optimal cache-adaptive algorithms and vica-versa.

However, this paper doesn't offer a simple framework for evaluating the optimality of a given algorithm. In the upcoming paper "Cache-adaptive Analysis" a simple method for proving cache-adaptive optimality is presented [MABM]. But, proving optimality doesn't tell us how many cache misses an algorithm will take. "Cache-adaptive Analysis" includes a section on the evaluation of non-optimal algorithms. This section gives lower bounds on the progress made by even non-optimal algorithms [MABM]. In this thesis, we do that analysis with a progress function that allows for smoother analysis. Additionally, we offer the time bound for optimal or non-optimal cache-adaptive algorithms. Our analysis allows an algorithm designer to predict when their computation will finish if given the function of the size of cache over time.

Results. Chapter 3 is related to the work in progress with Bender, Demaine, Ebrahimi, Fineman, Johnson, Lincoln, Lynch and McCauley. This Chapter is the work done by Lincoln and Lynch advised by Demaine. This Thesis characterizes the time for any recursive algorithm in the external memory model. This Thesis provides a formula for recursive algorithms of the form $T(N) = aT(N/b) + N^c$ to compute progress. In the cache-adaptive model, this Thesis, offers a method to calculate finishing time up to constant factors. This Thesis bounds the progress made in squares of memory which last for T cache misses and have $M(t) = TB$ for the T cache misses. This result shows that progress is guaranteeably made throughout execution.

Chapter 2

Background

In this chapter we will cover the previous research in the field to give context for the results in this thesis. The importance of caching in the practical efficiency of computation is huge. This has sparked an interest among theorists in modeling the efficiency of algorithms with regards to caching. There are three models that relate to this work primarily: the external-memory model, the cache oblivious model and the cache-adaptive model.

This chapter will offer a relatively brief overview of these topics. An excellent survey paper covering the external-memory model and the cache oblivious model is “Cache-Oblivious Algorithms and Data Structures” by Erik D. Demaine [Dem02].

2.1 External-Memory Model

The external-memory model is known by many names, *the external-memory model*, *the I/O model* and *disk access model* [Dem02]. The external memory model attempts to capture the time constraints of caching. Specifically, there are three components to the model: disk, cache and CPU. The disk is treated as having infinite size. Cache accesses and CPU operations are low-cost. However, loading B contiguous words, a cache line, from disk into cache takes $O(1)$ time. The model is expressed in Figure 1.

The cache stores M words in $m = M/B$ cache lines. If a location in memory is requested then not just that location but a whole cache line, B words in length, is pulled from memory into cache. Moving one line from main memory to cache takes one unit of time, writing a line back to main memory takes 1 unit of time. Writing, reading and doing computations on elements in the cache takes

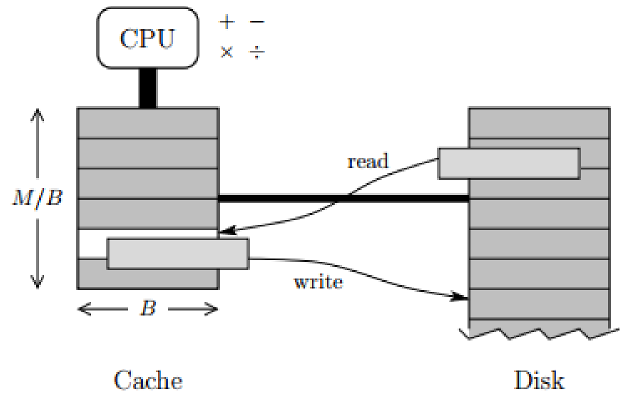

Figure 2-1: An image depicting the external-memory model from Demaine's paper [Dem02].

no time.

The external-memory model motivates the construction of structures like B trees. B-trees are used in practice, for example, they are used in mySQL [Ora15] and OSX [osx94]. The structure can speed up search speeds from $\log_2 n$ to $\log_B n$ which is a factor of $\log_2 B$. Practitioners care about factors of $\log_2 B$ because B can often be of size 2^9 or even 2^{20} for disk.

A common assumption in this model is that B, the cache line size, is smaller than the whole cache M by some margin. This is called the tall cache assumption. The standard tall cache assumption is $M = \Omega(B^2)$. There is a weaker tall cache assumption sometimes made $M = \Omega(B^c)$ where $c > 1$ [Dem02].

2.2 Cache-Oblivious Model

The cache oblivious model was proposed by Frigo, Leiserson, Prokop and Ramachandran in "Cache Oblivious Algorithms" [FLPR99]. In this model an algorithm is subject to the constraints of the external-memory model and the algorithms are not given access to the size of cache nor the size of a cache line. Many cache oblivious algorithms have been designed to run within a constant factor of the best algorithms in the external-memory model. These algorithms run efficiently on multiple machines without the need of alteration, access or measurement of the low level information about the machine. Additionally, on real machines, there are multiple levels of caches [Dem02]. The higher-level caches are faster and smaller; the lower the level of cache the more expensive a read is. A cache-oblivious algorithm that performs within a constant factor of the optimal algorithm for the external-memory model will make within a constant factor of the ideal number of cache misses on

every layer of cache [Dem02]. Furthermore, if we take any weighting on the cache misses on a cache with multiple levels, an efficient cache oblivious algorithms will be within a constant factor of the optimal algorithm [Dem02].

2.3 Cache-Adaptive Model

The cache-adaptive model was proposed in 2014 in the paper "Cache-adaptive Algorithms" [BEF$^+$14]. The cache-adaptive model generalizes the external-memory model by considering what happens when the size of available cache changes over time. The effective access to cache can be affected by how many other processes are running on the same machine. For example, even when running the same program on several different CPUs with shared cache there can be persistent unfairness in the speed of the algorithms [DMS14]. This persistent unfairness results from the fact that over time one process will randomly start to move faster, moving faster means it controls a greater portion of the cache, which causes it to continue to run faster. The suggestion of the paper is to occasionally clear the cache. If done in practice some algorithms will gain access to more of the cache slowly over time, only to have quick drops in their access to cache. This motivates studying the performance of algorithms with changing cache size.

A paper by Peserico in 2013 proposes a model to capture the idea of changing cache [Pes13]. In his model every operation (cache access) can cause the size of cache to change. Peserico showed that least recently used (LRU) cache replacement performs within a constant factor of optimal cache replacement (OPT) in this model. Peserico also analyzes a wider set of replacement strategies. The downside of this model is that cache sizes are changing even when time isn't moving forward in our model.

In 2014 Bender, Ebrahimi, Fineman, Ghasemiesfeh, Johnson and McCauley [BEF$^+$14] developed another model, the *cache-adaptive model*, for changing cache size. In this model the size of cache can only change when a cache miss occurs. This model allows cache sizes to change less frequently than Peserico, capturing the fact that cache ought to only change when time moves forward. The increased realism of this model makes it feasible to compare the running times of two different algorithms on the same function of changing cache size.

In the cache-adaptive model the algorithm has a cache of size $M(t)$ and a cache line of size B. The ***memory profile*** is a function that specifies the size of the cache at the tth cache miss [BEF$^+$14].

Specifically, the memory profile is a function $m(t)$ from natural numbers to natural numbers, where the cache has $m(t)$ cache lines, or $m(t)B$ words, at the tth cache miss. The ***memory profile in words*** is the function for $M(t) = Bm(t)$, for block size B. The algorithm starts running at $t = 0$ and, every cache miss, time increments by one step. If $M(t)$ increases then there is more room in cache and no elements need be replaced. If $M(t)$ decreases by $\delta \cdot B = M(t) - M(t-1)$ then the cache replacement strategy removes $\delta + 1$ cache lines. The cache miss itself results in at least one cache line being removed from cache, and the decrease in size of the cache results in another δ lines being removed.

One example of how tools have to change given the non-constant cache size is the tall cache assumption. Because $M(t)$ isn't constant, this bound basically serves as a lower bound for the function. This is reasonable because we still don't expect memories that are very small. We say that $M(t)$ is *H-respecting* if $M(t) \geq H(B)$ for all $t \geq 0$ [MABM].

Two important tools from the cache-adaptive paper were the concept of ***speed augmentation*** and ***square profiles*** [BEF$^+$14].

Definition 1. *An algorithm with **c-speed augmentation** performs c cache misses in each time step in the memory profile, $m(t)$.*

This gives a useful way to consider how algorithms perform on a given memory profile up to constant factors. In the cache-adaptive model LRU with 4-speed augmentation takes at most as many cache misses as OPT [BEF$^+$14].

Square profiles are a tool for evaluating algorithms [BEF$^+$14]. In Chapter 3 they will be used to get a lower bound on the amount of progress made in each cache miss. The square profiles are useful because it is easier to evaluate caching algorithms over periods where the cache's size stays constant.

Definition 2. *A memory profile m is **square** if it is a step function such that each step in the function has the same hight and width. Specifically, there must exist a set of times $\{t_0, t_1, \ldots\}$ where $0 = t_0 < t_1 < \ldots$ such that, for all $t \in [t_i, t_{i+1})$, $m(t) = t_{i+1} - t_i$.*

Additionally, an algorithm A run on a memory profile $m(t)$ will have more cache misses than the algorithm A run a memory profile $m'(t)$ where $m'(t) \geq m(t)$. So we will define the square profiles above and below the memory profile. These will be used to get an upper bound on the progress done in a given region of memory. It will give a lower bound on the size of memory during that same

stretch. The cache-adaptivity paper further defined square boundaries that exist above and bellow a memory profile [BEF$^+$14].

Definition 3. *The* ***upper square boundaries*** $t_0 < t_1 < t_2 < \ldots$ *of a memory profile* $m(t)$ *are defined by their set of times* $\{t_0, t_1, \ldots\}$*. We will have* $t_0 = m(0)$*. Recursively define* t_{i+1} *as the smallest integer such that* $t_{i+1} - t_i \geq m(t)$ *for all* $t \in [t_i, t_{i+1})$*. The* ***upper square profile*** *of* m *is the profile* m' *defined by* $m'(t) = t_{i+1} - t_i$ *for all* $t \in [t_i, t_{i+1})$*.*

Definition 4. *The* ***inner square boundaries*** $t_0 < t_1 < t_2 < \ldots$ *of a memory profile* $m(t)$ *are defined by their set of times* $\{t_0, t_1, \ldots\}$*. We will have* $t_0 = m(0)$*. 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* $t \in [t_i, t_{i+1})$*.*

2.4 Example: Matrix Multiplication algorithms

Here we will use matrix multiplication as an example of a case where the different models result in different efficiencies for algorithms. matrix multiplication is a good example of how the model changes algorithm analysis. The problem of matrix multiplication is: to compute $C = A \cdot B$ where A and B are both $n \times n$ matrices. Note that here the problem size is not n, the input are two matrices each have n^2 entries.

2.4.1 Traditional Computational Model

Naive matrix multiplication method computes the summation $c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}$ for each $c_{i,j}$ resulting in an $O(n^3)$-time algorithm.

There has been a long history of improving the constants in the traditional computation model. In 1969 Strassen improved the algorithm to $O(n^{2.807})$ [Str69]. In 1987 Copersmith-Winograd improved the exponent to 2.376 [CW87]. Then in 2012 Williams brought the exponent down to 2.3727 [Wil12].

2.4.2 External Memory Model

Let us consider briefly analyze the naive method in the external-memory model. Consider the case where $n > M$ If each row is laid out in order in main memory then we incur $n/B + n$ cache misses

for each calculation of $c_{i,j}$. We still need to calculate n^2 entries. So in total we will still incur $\Theta(n^3)$ cache misses with this naive approach!

We have M words of memory. Intuitively, we want to compute multiplication that fully fit in cache. Consider the algorithm which splits A and B into $\frac{2n^2}{M}$ sub-matrices of size $\sqrt{M/2}$ by $\sqrt{M/2}$ and does all the multiplications needed on these sub-matrices [GVL12]. Each sub-matrix multiplication takes $O(M/B)$ time (to read in the two sub-matrices to multiply). There are a total of $(n/\sqrt{M/2})^2$ sub-matrices in C. Computing each of these takes $n/\sqrt{M/2}$ sub-matrix multiplications.

Matrix multiplication with this external-memory model method takes $O(n^3/(B\sqrt{M}) + n^2/B)$ cache misses (the n^2/B term appears because regardless of the size of M we must read in the two input matrices). This is an improvement over the naive approach by a factor of $B\sqrt{M}$.

2.4.3 Cache Oblivious Model

Despite the improvement of the external-memory model approach, it requires a specific knowledge of the size of memory, M. Imagine we did the approach explained above, with a bad guess. Say we split the matrix into sub-matrices of size $M'/2$ instead of M. If M' were, say, 5 times larger than M, each sub-matrix multiplication will take much longer. Just like in the naive approach we will have $(M')^{3/2}$ cache misses per sub-matrix, resulting in a multiplication time of n^3!

Luckily, Frigo, Leiserson, Prokop and Ramachandran developed an approach to combat this [FLPR99]. By recursively dividing the matrices we can avoid the penalty of poorly sized matrices.

```
def mm(A,B):
    if A.sidelength() ≤ 2:
        return naive_mm(A,B)
    # let A.subpart({0,1},{0,1}) return the corresponding sub-matrices of size n^2/4
    for i ∈ {0,1}:
        for j ∈ {0,1}:
            C.subpart(i,j) = mm(A.subpart(i,0),B.subpart(0,j))+ mm(A.subpart(i,1),B.subpart(1,j))
    return C
```

The matrices are laid out in memory in a clever zig-zag pattern described in [FLPR99]. This pattern allows us to save a factor of the cache line size B. As soon as we recurse down to a size where the size of the two matrices is less than M, we can finish the whole multiplication of those sub-matrices with M/B cache misses. This corresponds to a recurrence of

$$T(n) = 8T(n/2) + \Theta(n^2),$$

$$T(\sqrt{M}) = M/B.$$

To calculate $T(n)$ we can consider the height and the branching factor of the recurrence tree. The height will be $\lg(n/\sqrt{M}) = 1/2 \lg(n^2/M)$ and the branching factor is 8. So in total we get

$$T(n) = M/B 8^{1/2 \lg(n^2/M)} + n^2/B = M/B(n^2/M)^{3/2} + n^2/B = n^3/(B\sqrt{M}) + n^2/B.$$

Note that our recursion has brought us back to the bound from the non-oblivious external-memory model! But now we obtain this bound regardless of the size of the cache.

2.4.4 Cache Adaptive Model

Once again we will see that the previous matrix multiplication approach fails to achieve the bounds we want. The process of writing back the whole matrix every time, the $\Theta(n^2)$ task from the recurrence, can make this algorithm slow. Consider the case where a large amount of memory is given during the easy n^2 task (this doesn't speed up the task at all), and the memory is decreased when we get to the multiplications. This can result in an inefficiency of $\log(n/B^2)$ [BEF$^+$14]. The proof of inefficiency is in the paper "Cache-adaptive Algorithms" [BEF$^+$14].

To fix this we can simply write back to the disk at all the smallest sized subproblems. Consider the problem where we input A,B and C where C is empty (filled with 0s). If you want to consider the case where C is not empty, then a n^2/B pass can be made to write 0s into all of C's entries.

```
def mm(A,B,C):
    if A.sidelength() ≤ 2:
        C += naive_mm(A, B)
    # let A.subpart({0, 1}, {0, 1}) return the corresponding sub-matrices of size n^2/4
```

```
for i ∈ {0,1}:
    for j ∈ {0,1}:
        mm(A.subpart(i,0),B.subpart(0,j),C.subpart(i,j))
        mm(A.subpart(i,1),B.subpart(1,j),C.subpart(i,j))
```

In this case we get a recurrence of $T(n) = 8T(n/2) + \Theta(1)$. Once again we have $T(\sqrt{M}) = M/B$. Now the matrix multiplication algorithm is cache-adaptive [BEF+14].

Chapter 3

Algorithm Performance

This chapter is work that was done by Andrea Lincoln and Jayson Lynch advised by Erik Demaine. The work is related to the work done in Cache-Adaptive Algorithms [MABM].

In this chapter, we give explicit time bounds for a class of recursive algorithms. First, we give an analysis of this class of algorithms in the static memory case, which will be used in subsequent proofs but is interesting in its own right. Next, we define a notion of progress, give a formula for calculating it in Theorem 16, and show the total amount of progress needed to complete a problem in Corollary 15. These two results allow one to calculate the running time of an algorithm given a known memory profile up to constant factors. Theorem 17 also gives a strong statement about progress within any square of memory compared to running the algorithm with a constant memory profile. We then give simplified versions of these results in corollaries 21 and 22 given some common assumptions about the algorithm's parameters.

3.1 Assumptions

In this chapter we make the following assumptions about our algorithms and memory profile, unless specified otherwise:

First , $m(t+1) \leq m(t)+1$ for all $t > 0$. This requirement is in place because an algorithm can not pull in more than one cache line per time step. If $m(t)$ grew faster the extra memory would be unusable. If one wants to analyze a memory profile that grows faster than this, one can alter it, making the profile smaller at times where it grows too quickly.

Second, the algorithms we analyze will be (a, b, c)-*regular*. The definition of (a, b, c)-regular

algorithms is from the work "Cache-adaptive Analysis" [MABM]. An (a, b, c)-regular algorithm is essentially a recursive algorithm with the recurrence $T(N) = aT(N/b) + N^c$.

Definition 5. *We require $a \geq 1$, $b > 1$, and $0 \leq c \leq 1$ be constants. An **(a, b, c)-regular** algorithm on an input of size N, makes*

(i) a recursive calls on subproblems of size N/b and;

(ii) performs $O(1)$ linear scan *of size $\Theta(N^c)$.*

Now we will define a ***linear scan***. Basically, the linear scan is a merge for a problem that is laid out well in memory [MABM].

Definition 6. *A **linear scan of size ℓ** accesses ℓ distinct locations, performs $O(\ell)$ memory references, and incurs $O(1 + \ell/B)$ cache misses.*

Third, all algorithms are *memory monotone* or cache-oblivious. If our algorithm is not cache-oblivious it uses a memory-monotone, conservative page replacement algorithm. The memory-montone requrement is defined in "Cache-adaptive Analysis" [MABM].

Definition 7. *An algorithm is **memory monotone** if it runs no slower given more memory.*

Using LRU or Optimal Cache Replacement more memory will always result in the same number or fewer cache misses [BEF+14]. It is the case that the FIFO cache replacement can anomalously take longer with more memory [BNS69]. But, algorithms can use LRU cache replacement. Our goal with the memory monotone requirement is not to rule out using FIFO cache replacement, but to prevent algorithms from gaining an advantage from reading in the size of cache as an advice string.

3.2 Square Profiles

We will be using square profiles to aid in our analysis. We will do part of the analysis looking at the individual squares making up a square profile. So we will define a square of size M. The square will have hight M and width M/B, this is because every cache miss can bring in a whole cache line. So, in a given square we read in enough memory to fully replace the contents of the cache.

Definition 8. *Let $\square M$ be a section of M/B cache misses where with cache size M. So*

$$M(t) = \begin{cases} M, & \text{if } 0 \leq t \leq M/B \\ 0, & \text{else} \end{cases}.$$

Definition 9. *Let $T(N, B, M)$ be the number of cache misses an (a, b, c)-regular algorithm makes when run on a memory profile of $M(t) = M$ with problem size N.*

The total number of cache misses for an (a, b, c)-regular algorithm when there is a cache of size M will be the sum of:

- the time taken for all the recursive calls that completely fit in cache,
- the time to merge the output of the problems that completely fit in cache and,
- the time to read the entire problem.

In the end we find that if $c < 1$ then the time taken for recursive calls that fit in cache dominates the time taken by merging. If $c = 1$ the two terms have equal weight.

Lemma 10.

$$f(N, B, M) = \Theta\left(\frac{M}{B}\left(\frac{N}{M}\right)^{\log_b a} + \frac{M^c}{B}\left(\frac{N}{M}\right)^{\log_b a}\right) + \frac{N}{B}$$

Proof. We split the cost into the cost to solve problems of size M and the cost to solve merges of problems of size M to size N.

When N has been divided enough times that subproblems are of size M the subproblems will take $O(M/B)$ cache misses to solve. We need to subdivide N $\log_b\left(\frac{N}{M}\right)$ times to have the problems be of size M. Because the branching factor is a this means we must solve $a^{\log_b\left(\frac{N}{M}\right)} = \left(\frac{N}{M}\right)^{\log_b a}$ subproblems of size M. Thus the total cost for solving problems of size M is

$$\frac{M}{B}\left(\frac{N}{M}\right)^{\log_b a}.$$

For the merges we will incur extra cache misses for all merges of size greater than M. Each merge of a problem of size X takes $O(X^c/B)$ cache misses. The total merge cost is thus equal to the sum

$$\sum_{i=0}^{\log_b(N/M)} a^i \left(Nb^{-i}\right)^c.$$

This sum is equal to

$$\frac{M^c}{B}\left(\frac{N}{M}\right)^{\log_b a}.$$

Finally, regardless of the size of cache we must read the entirety of the problem into memory. This takes $\frac{N}{B}$ time.

Thus the total sum is

$$f(N, B, M) = \Theta\left(\frac{M}{B}\left(\frac{N}{M}\right)^{\log_b a} + \frac{M^c}{B}\left(\frac{N}{M}\right)^{\log_b a}\right) + \frac{N}{B}$$

□

3.3 Progress

We now define *progress*, which is a potential function for an amortization argument. Intuitively, it is measuring what percentage of the work has been completed by adding up how many subproblems which fit in B would have to be solved to by the algorithm. The potential at any given point is simply the sum of the progress weight up to that point. The potential of a given operation is simply the progress weight of that operation.

Definition 11. *Progress, $p(t)$, has the following properties:*

1. *On an access, s_i, where a smallest size subproblem is accessed (there are $N^{\log_b a}$ smallest subproblems total) then*

$$p(s_i) = 1.$$

2. *On an access, s_i, where a problem of size X is merged with d other problems of size X (a merge has X^c accesses) then*

$$p(s_i) = \left\lceil X^{\log_b a - 1} \right\rceil.$$

Definition 12. *Let the potential $\Phi(t)$ be the progress made up to time t (i.e. cache miss t). Let s_i be the access that causes the $t + 1^{th}$ cache miss.*

$$\Phi(t) = \sum_{j=0}^{i} p(s_j).$$

Lemma 13. *The total progress encompassed in a completely solved problem of size X is*

$$X^{\log_b a} \frac{1}{a}\left(\frac{1 - X^{c-1}}{1 - b^{c-1}}\right)$$

Proof. The progress involved in solving the smallest subproblems will be $\Theta(X^{\log_b a})$, the number of smallest sized subproblems.

Let the progress involved in all of the merges used in solving a problem of size X (which includes the merging of the subproblems of size X/b) be $P(X)$. Note that $P(X)$ is the sum of the progress of all of the merges needed to merge the smaller problems into the larger problem of size X.

The merge of a problems of size Y is the progress per merge access multiplied by the number of merge accesses. So the progress for merging problems of size Y is

$$Y^{\log_b a-1}Y^c.$$

There are a^i merges to be done for problems of size X/b^i in order to solve one problem of size X, because a is the branching factor.

So using these two facts we have that

$$P(X) = \sum_{i=0}^{\log_s X-1} a^i(b^{\lg_s(X)-1-i})^{\log_b a-1+c}$$
$$P(X) = \sum_{i=0}^{\log_s X-1} a^i((X/b)b^{-i})^{\log_b a+c-1}$$
$$P(X) = (X/b)^{\log_b a+c-1} \sum_{i=0}^{\log_s X-1} a^i(b^{-i})^{\log_b a+c-1}$$
$$P(X) = (X/b)^{\log_b a+c-1} \sum_{i=0}^{\log_s X-1} (a/b^{\log_b a+c-1})^i.$$
$$P(X) = (X/b)^{\log_b a+c-1} \sum_{i=0}^{\log_s X-1} (1/b^{c-1})^i$$
$$P(X) = (X/b)^{\log_b a+c-1} \sum_{i=0}^{\log_s X-1} (b^{1-c})^i$$
$$P(X) = (X/b)^{\log_b a+c-1} \left(\frac{(b^{1-c})^{\lg_s(X)} - 1}{b^{1-c}-1}\right)$$

$$P(X) = (X/b)^{\log_b a+c-1}\left(\frac{(X^{1-c})-1}{b^{1-c}-1}\right)$$
$$P(X) = (X/b)^{\log_b a}(X/b)^{c-1}\left(\frac{(X^{1-c})-1}{b^{1-c}-1}\right)$$
$$P(X) = (X/b)^{\log_b a}b^{1-c}\left(\frac{1-X^{c-1}}{b^{1-c}-1}\right)$$
$$P(X) = (X/b)^{\log_b a}b^{1-c}b^{c-1}\left(\frac{1-X^{c-1}}{1-b^{c-1}}\right)$$
$$P(X) = X^{\log_b a}\left(\frac{1}{a}\right)\left(\frac{1-X^{c-1}}{1-b^{c-1}}\right)$$

□

Definition 14. *Let $\Phi(\infty)$ be the total progress assigned to the execution of an (a, b, c)-regular algorithm run on a problem of size N.*

Corollary 15. *The total progress on the problem is*

$$\Phi(\infty) = \Theta\left(\frac{1}{a}N^{\log_b a}\left(\frac{1-N^{c-1}}{1-b^{c-1}}\right)\right).$$

Proof. $\Phi(\infty)$ is equivalent to solving a problem of size N. Thus $\Phi(\infty) = \Theta\left(\frac{1}{a}N^{\log_b a}\left(\frac{1-N^{c-1}}{1-b^{c-1}}\right)\right)$. □

3.4 Main Theorems

Our first theorem gives a time bound based on the value of M at various accesses to memory. The proofs of Theorems 16 and 17 can be found in Section 3.5.

Theorem 16. *For a memory monotone (a, b, c)-regular algorithm where $a \geq b$ run over an H-respecting $M(t)$ the progress done by cache miss T , is:*

$$\Phi(T) = \Omega\left(\left[\frac{Bb}{a^{\log_b(12)+3}}\right]\sum_{t=0}^{T} M(t)^{\log_b a-1}\right).$$

The time, T, it takes for the algorithm to complete is the first time T such that $\Phi(T)$ equals $\Phi(\infty)$.

So, in order to produce a time bound one can simply find a T such that

$$\Phi(T) = \Phi(\infty) = \Theta\left(N^{\log_b a}\left(\frac{1-N^{c-1}}{1-b^{c-1}}\right)\right).$$

Due to the asymptotic bounds this will only guarantee that we have finished up to a constant factor of the progress. Information about the hidden constant factors in the algorithm would be needed to make this calculation exact. If for an algorithm A we have that $\Omega\left(\left[\frac{Bb}{a^{\log_b(12)+3}}\right]\sum_{t=0}^{T} M(t)^{\log_b a-1}\right) = \Theta(\Phi(\infty))$. Then there exists a constant c such that the algorithm A with c-speed augmentation run on $M(t)$ will finish by time T. One may also be concerned that we need to know about the execution of the algorithm to be able to calculate when it will terminate; however, the sum is associative and only cares about about the memory profile, not the memory profile in relation to the algorithm. If one finds this calculation to be cumbersome, there are a number of approximations that can be made, such as only considering memory above a certain fixed level.

Theorem 17. *Let $\square M$ be a M square profile with M/B cache misses and M cache available. Let $\Phi(\square M)$ be the progress an (a, b, c)-regular algorithm where $a > b$ and $c < 1$ makes during the square $\square M$. Then, for $a = O(1)$ and $b = O(1)$*

$$\frac{\Phi(\square M)}{M} = \Omega\left(\frac{\Phi(\infty)}{f(N, B, M)}\right).$$

In other words the average progress in each square is greater than or equal to of the average progress made running the algorithm on a constant $M(t) = M$ for its entire execution, up to constant factors. This shows that the good relative behavior of worst case alignment holds not only globally but also over every local square. Furthermore, this shows that for any valid lower bound square profile the order in which we analyze them does not matter. Additionally, if an algorithm A would finish running on $M(t)$ after time T and $M'(t)$ up to time T has values which are a re-arrangement of $M(t)$up to time T then there exists a constant c such that A with c-speed augmentation will complete by time T.

We now examine $\Phi(\infty)$ in more detail.

Lemma 18.

$$\left(\frac{1-N^{c-1}}{1-b^{c-1}}\right) = \begin{cases} \Theta\left(1/\left(1-b^{c-1}\right)\right), & \textit{if } c < 1 \\ \Theta\left(\log_b N\right), & \textit{if } c = 1 \\ \Theta\left((N/b)^{c-1}\right), & \textit{if } c > 1 \end{cases}$$

Proof. For the case $c < 1$, N^{c-1} is very small for large N. Thus, the value asymptotically approaches $\Theta(1/(1-b^{c-1}))$.

For the case $c = 1$, we simply take the limit as $c - 1$ approaches 0 and the division approaches $\Theta(\log_b N)$.

For the case $c > 1$, because $N \geq b > 1$ we have that $1 - N^{c-1} < 0$ and $1 - b^{c-1} < 0$. Thus, we have that, $\left(\left(1 - N^{c-1}\right) / \left(1 - b^{c-1}\right)\right) = \Theta\left(N^{c-1}/b^{c-1}\right)$. □

In (a, b, c)-regular algorithm $0 \leq c \leq 1$ so either this term is some constant, or if $c = 1$ then there is a log factor introduced.

Lemma 19.

$$\Phi(\infty) = \begin{cases} \Theta\left(\frac{1}{a}N^{\log_b a} / \left(1 - b^{c-1}\right)\right), & \text{if } c < 1 \\ \Theta\left(\frac{1}{a}N^{\log_b a}\log_b N\right), & \text{if } c = 1 \\ \Theta\left(\frac{1}{a}N^{\log_b a}\left(\frac{N}{b}\right)^{c-1}\right), & \text{if } c > 1 \end{cases}$$

Proof. Plug in the values from Lemma 18 into the result about total progress from Lemma 15. □

One may notice that in the case $c < 1$ and $a \geq b$ the total number of smallest size sub-problems is within a constant factor of $\Phi(\infty)$, suggesting its use as a natural measure of progress. So for (a, b, c)-regular algorithms, if $c < 1$ then we are in the case of $\Theta(N^{\log_b a})$ and, if $c = 1$ then $\Theta(N^{\log_b a}\lg(N))$. So we are adding a log factor when $c = 1$. We will now compare our bound on performance to the average performance achieved with a constant $M(t) = M$.

Lemma 20. *The average progress to cache miss ratio when $M(t) = M$ and a and b are constants is*

$$\Phi(\infty)/f(N, B, M) = \begin{cases} \Omega\left(BM^{\log_b a - 1}\right), & \text{if } c < 1, \\ \Omega\left(BM^{\log_b a - 1}\left(\frac{\log_b N}{\log_b(M)}\right)\right), & \text{if } c = 1, \\ \Omega\left(BM^{\log_b a - 1}\left(\frac{N}{M}\right)^{c-1}\right), & \text{if } c > 1. \end{cases}$$

Proof. With a and b constant

$$\Phi(\infty)/f(N, B, M) = \begin{cases} \Theta\left(N^{\log_b a}\right), & \text{if } c < 1, \\ \Theta\left(N^{\log_b a}\log_b N\right), & \text{if } c = 1, \\ \Theta\left(N^{\log_b a}(N)^{c-1}\right), & \text{if } c > 1. \end{cases}$$

Furthermore with a and b constant,

$$f(N,B,M) = (M/B)\,(N/M)^{\log_b a}(1+O(M^{c-1}))$$

. Thus, when we divide one by the other we get

$$\Phi(\infty)/f(N,B,M) = \begin{cases} \Omega\left(BM^{\log_b a-1}\right), & \text{if } c<1, \\ \Omega\left(BM^{\log_b a-1}\log_b N/\log_b M\right), & \text{if } c=1, \\ \Omega\left(BM^{\log_b a-1}(N/M)^{c-1}\right), & \text{if } c>1. \end{cases}$$

□

When we compare this to Theorem 16 we can note that the average progress when a and b are constants is $BM(t)^{\log_b a-1}$, which exactly matches the average progress for the case where $c<1$. Intuitively, when $c<1$, the (a,b,c)-regular algorithm makes progress at a rate within a constant factor of the average performance if the memory were constant.

Corollary 21. *When $a=O(1)$, $b=O(1)$:*

$$\Phi(T) = \Omega\left(B\sum_{t=0}^{T} M(t)^{\log_b a-1}\right).$$

Proof. When $bB>2$ this result follows from Theorem 16. When $bB\le 2$, we have that the average progress will be $ba^{-\log_b(12)-1}M(t)^{\log_b a-1}$. However, B, a and b are all $O(1)$. Thus,

$$ba^{-\log_b(12)-1}M(t)^{\log_b a-1} = \Theta\left(BM(t)^{\log_b a-1}\right).$$

□

In most algorithm analysis your subproblem size decrease ratio, b, and your branching factor, a, are constants and not picked based on the input size. Thus, this simplified statement holds for most cases we care about.

Corollary 22. *When $b=O(1)$, $a=O(1)$, $c<1$ and $\frac{1}{1-c}=O(1)$*

$$\Phi(\infty) = \Theta\left(N^{\log_b a}\right).$$

So, the total amount of progress needed for any fixed c greater than 1 is asymptotically equivalent to the total number of smallest sized subproblems. This shows that the progress measure is within a constant factor of the progress measure of assigning one unit of progress per computation. The progress measure of one per cache miss is used in the paper "Cache-adaptive Analysis" [MABM].

3.5 Proofs About Average Progress

The proof will be presented in two sections, the first giving bounds on how well an algorithm will perform given a square memory profile and the second section giving a bound on how well the performance at a general $M(t)$ compares to a strictly smaller square profile.

3.5.1 Bounding Progress

For the next chunk of the proof we will argue that the progress made in a $\square M'$ square is at least some value. We will credit this progress to the section of upper bound profile associated with the next square in the profile, thus completing the argument.

Lemma 23. *The amount of progress made solving a problems of size $\frac{M'}{b}$ in $M'/(bB)$ cache misses is*

$$\Omega\left(\left(\frac{M'}{b}\right)^{\log_b a} \frac{1}{a}\left(\left(1-\left(\frac{M'}{b}\right)^{c-1}\right) / \left(1-b^{c-1}\right)\right)\right)$$

Proof. This follows from $X = \frac{M'}{b}$ from Lemma 13. ☐

Lemma 24. *The amount of progress that can be made merging problems of size $\frac{M'}{b}$ or larger over the course of $M'/(bB)$ cache misses is*

$$\Omega\left((M'/b)^{\log_b a}\right)$$

Proof. The progress per access in a merge of problems of size M'/b or larger is $\Omega\left((M'/b)^{\log_b a-1}\right)$. Every B access are pulled in in one cache pull. So the number of access pulled in in $M'/(bB)$ cache misses is $BM'/(bB)$. Thus the total progress for $M'/\left(bB\right)$ cache misses is

$$\Omega\left((M'/b)^{\log_b a-1} BM'/\left(bB\right)\right) = \Omega\left((M'/b)^{\log_b a}\right)$$

☐

Now we want to lower bound the number of sections of $\frac{M'}{bB}$ cache misses in the problem because determining exactly how much progress is made by partially completed subproblems is difficult.

Lemma 25. *There are at least $b-2$ uninterrupted sections of $\frac{M'}{b}$ cache misses in $\square M'$.*

Proof. The start of the M' cache misses may be in the middle of a partially completed sub-problem of size M'/b. Solving the rest of this problem will take less than $\frac{M'}{bB}$ cache misses. We will treat this portion as making no progress.

Any completed problems of size M'/b solved in the middle of the computation will be solved uninterrupted with at least M' cache available.

Any merges that happen in the middle of the problem that incur cache misses also give progress.

So the only potentially interrupted sections are the beginning and end of the M' cache misses. We can at most incur $\frac{M'}{bB}$ cache misses at each end. Thus in total accounting for at most $2\frac{M'}{bB}$ lost cache misses. There are b sections of $\frac{M'}{bB}$ cache misses over the course of M' cache misses.

Thus in the time of M' cache misses we must solve at least $b-2$ sections of $\frac{M'}{bB}$ cache misses, either in the form of merges or as entire sub problems. $\square$

Corollary 26. *If $b > 2$ there must be at least $\Omega\left(b/a^2\left(M'\right)^{\log_a(b)}\right)$ progress.*

Proof. There are at least $b-2>0$ sections of $\frac{M'}{bB}$ cache misses. These can either have merges of size $M'/(bB)$ or larger or a problem of size $M'/(bB)$ being solved. The amount of progress solved by either is

$$\Omega\left(\frac{1}{a}\left(\frac{M'}{b}\right)^{\log_b a}\right).$$

This is equal to

$$\Omega\left(\frac{1}{a^2}\left(M'\right)^{\log_b a}\right)$$

Thus, there must be at least $\Omega(\frac{1}{a^2}(b-2)\left(M'\right)^{\log_b a})$ progress over the course of M' cache misses at memory M', which is equivalent to

$$\Omega\left(\frac{b}{a^2}\left(M'\right)^{\log_b a}\right).$$

$\square$

Lemma 27. *If $b \leq 2$ then there must be at least $\Omega(a^{-\log_b(3)-1}\left(M'\right)^{\log_b a})$ progress in a $\square M'$.*

Proof. If $b \leq 2$ then $1 < b \leq 2$. Let $x = \lceil \log_b(3) \rceil$. We consider problems of size $M'/(b^x)$. We will solve at least $(b^x - 2)$ of these problems. The amount of progress solved in a square $\square\, M'$ is

$$\Omega\left(a^{-\log_b(3)-1}\left(M'\right)^{\log_b a}\right).$$

$\square$

3.5.2 Comparing the two square profiles

We have determined a lower bound on the average progress per cache miss given the value of $M'(t)$, where $M'(t)$ is a square profile with heights that are all powers of b. We can use this knowledge to lower bound the progress per cache miss given $M(t)$ which is strictly greater.

Lemma 28. *Let the square after $\square\, M'_i$ be the square $\square\, M'_{i+1}$. Let the maximum value of $M(t)$ over the square M'_{i+1} be M''.*

If $b > 2$ the average progress per cache miss can be bounded as

$$\Omega\left(b^2/a^{3+\log_b(4)}\left(M''\right)^{\log_b a}\right)$$

If $b \leq 2$ the average progress per cache miss can be bounded as

$$\Omega\left(bBa^{-\log_b(12)-2}\left(M''\right)^{\log_b a-1}\right)$$

Proof. Because the maximum slope of $M(t)$ is 1 we have that $M'' \leq 4bM'_i$.

If $b > 2$ the progress is $\Omega\left(b/a^2\left(M'_i\right)^{\log_b a}\right)$ for $O(M'_i/B)$ cache misses for an average progress per cache miss of

$$\Omega\left(bB/a^2\left(M'_i\right)^{\log_b a-1}\right),$$

which is

$$\Omega\left(bB/a^2\left(\frac{M''}{4b}\right)^{\log_b a-1}\right),$$

which is

$$\Omega\left(b^2B/a^{3+\log_b(4)}\left(M''\right)^{\log_b a-1}\right).$$

If $b \leq 2$ the progress is $\Omega\left(a^{-\log_b(3)-1}\left(M'_i\right)^{\log_b a}\right)$ for $O(M'_i/B)$ cache misses for an average

progress per cache miss of

$$\Omega\left(Ba^{-\log_b(3)-1}\left(M_i'\right)^{\log_b a-1}\right),$$

which is

$$\Omega\left(Ba^{-\log_b(3)-1}\left(\frac{M''}{4b}\right)^{\log_b a-1}\right),$$

which is

$$\Omega\left(bBa^{-\log_b(12)-2}\left(M''\right)^{\log_2(d)-1}\right).$$

□

Corollary 29. *Let the square after* $\square M_i'$ *be the square* $\square M_{i+1}'$. *Let the maximum value of* $M(t)$ *over the square* M_{i+1}' *be* M''. *The average progress per cache miss can be bounded as*

$$\Omega\left(bB/a^{\log_b(12)+3}\left(M''\right)^{\log_b a-1}\right)$$

Theorem 30. *The average progress per cache miss can be bounded as*

$$\Omega\left(bB/a^{\log_b(12)+3}\left(M(t)\right)^{\log_b a-1}\right)$$

Proof. Using Lemma 28 and the fact that $M'' \geq M(t)$, we can simply replace M'' with $M(t)$ and get a lower bound on the progress per cache miss in the box M_i'. The total number of cache misses in the box of size M_{i+1}' is M_{i+1}'. Furthermore, $M_{i+1}' \leq M(t)$. Thus, dividing by $M(t)$ will result in a lower bound on the progress per cache miss. □

3.6 Proving Theorems 17 and 16

First we will prove Theorem 17.

Theorem (17)**.** *Let* $\square M$ *be a* M *square profile with* M/B *cache misses and* M *cache available. Let* $\Phi(\square M)$ *be the progress an* (a, b, c)*-regular algorithm where* $a > b$ *and* $c < 1$ *makes during the square* $\square M$. *Then, for* $a = O(1)$ *and* $b = O(1)$

$$\frac{\Phi(\square M)}{M} = \Omega\left(\frac{\Phi(\infty)}{f(N,B,M)}\right).$$

Proof. By Corollary 29
$$\Phi(\square\, M) = \Omega\left(b/a^{\log_b(12)+3}\,(M)^{\log_a(b)}\right).$$
Thus, for $a = O(1)$ and $b = O(1)$,
$$\frac{\Phi(\square\, M)}{M/B} = \Omega\left(BM^{\log_b a-1}\right).$$
By Lemma 19 given that a and b are constant:
$$\Phi(\infty) = \Theta\left(N^{\log_b a}\right).$$
By Lemma 10 given that $c < 1$:
$$f(N, B, M) = \Theta\left(\frac{M}{B}\left(\frac{N}{M}\right)^{\log_b a}\right).$$
Thus,
$$\frac{\Phi(\infty)}{f(N, B, M)} = \Theta\left(BM^{\log_b a-1}\right).$$
Thus,
$$\frac{\Phi(\square\, M)}{M} = \Omega\left(BM^{\log_b a-1}\right).$$
$$\frac{\Phi(\square\, M)}{M} = \Omega\left(\frac{\Phi(\infty)}{f(N, B, M)}\right).$$
□

Now we prove Theorem 16.

Theorem (16)**.** *For a memory monotone* (a, b, c)*-regular algorithm where* $a \geq b$ *run over an* H*-respecting* $M(t)$ *the progress done by cache miss* T *, is:*
$$\Phi(T) = \Omega\left(\left[(Bb) \,/\, \left(a^{\log_b(12)+3}\right)\right]\sum_{t=0}^{T} M(t)^{\log_b a-1}\right).$$
Proof. Amortized over each section corresponding to a square in the lower bound square profile a cache miss makes $\Omega\left(\left[(Bb) \,/\, \left(a^{\log_b(12)+3}\right)\right] M(t)^{\log_b a-1}\right)$ progress. The squares of the square profile cover the whole of $M(t)$. Thus, if we sum over the number of cache misses we can get an approxi-

mation of the progress so far. Specifically:

$$\Phi(T) = \Omega\left(\left[(Bb) \,/\, \left(a^{\log_b(12)+3}\right)\right] \sum_{t=0}^{T} M(t)^{\log_b a - 1}\right).$$

□

Chapter 4

Conclusion

This thesis reviewed the theoretical models used in caching, including a relatively new model, cache adaptivity. Additionally, it presented results for analyzing the progress made by recursive functions. This thesis presented a formula to calculate progress made by time T given the cache profile $M(t)$, the cache line size B and the recurrence for the recursive algorithm. We showed that the progress made in a square, $\square M$, is with a constant factor of the average progress had the whole algorithm been run at $M(t) = M$. This showed that (a, b, c)-regular algorithms make consistently high progress through execution. Specifically, every $\frac{\Phi(\square M)}{M} = \Omega\left(\frac{\Phi(\infty)}{f(N,B,M)}\right)$. So, every square made more than or equal to the average progress for that size of cache up to constant factors. Analysis was offered that shows that the permutation of values in $M(t)$ doesn't effect the asymptotic run time as long as the size of memory never grows to fast.

Future work includes developing analysis for a larger set of algorithms in the cache-adaptive setting. Extensions beyond (a, b, c)-regular algorithms are in the upcoming paper "Cache-adaptive Analysis" [MABM]. However, there are not yet results for large classes of algorithms which aren't divide and conquer. Further work includes characterizing the types of memory profiles that exist in practice. If such a characterization is possible, on that set of memory profiles can stronger statements be made about algorithm performance?

Bibliography

[AV88] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[BEF+14] Michael A. Bender, Roozbeh Ebrahimi, Jeremy T. Fineman, Golnaz Ghasemiesfeh, Rob Johnson, and Samuel McCauley. Cache-adaptive algorithms. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 958–971, 2014.

[BNS69] L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Commun. ACM*, 12(6):349–353, June 1969.

[CW87] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6. ACM, 1987.

[Dem02] Erik D. Demaine. Cache-oblivious algorithms and data structures. Lecture Notes from the EEF Summer School on Massive Data Sets, 2002.

[DMS14] Dave Dice, Virendra J. Marathe, and Nir Shavit. Brief announcement: Persistent unfairness arising from cache residency imbalance. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’14, pages 82–83, New York, NY, USA, 2014. ACM.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annual Symposium on the Foundations of Computer Science (FOCS)*, pages 285–298, 1999.

[GVL12] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[Luu15] Dan Luu. What's new in cpus since the 80s and how does it affect programmers?, January 2015.

[LZGS84] Edward D Lazowska, John Zahorjan, G Scott Graham, and Kenneth C Sevcik. *Quantitative system performance: computer system analysis using queueing network models*, volume 84. Prentice-Hall Englewood Cliffs, 1984.

[MABM] Roozbeh Ebrahimi Rob Johnson Andrea Lincoln Jayson Lynch Michael A. Bender, Erik D. Demaine and Samuel McCauley. Cache-adaptive analysis. Submitted.

[Ora15] Oracle. Comparison of b-tree and hash indexes, 2015.

[osx94] Mac os x manual page for btree(3) - apple developer. `https://dev.mysql.com/doc/refman/5.5/en/index-btree-hash.html`, 1994.

[Pes13] Enoch Peserico. Paging with dynamic memory capacity. *CoRR*, abs/1304.6007, 2013.

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[Wil12] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 887–898. ACM, 2012.