Cache-Adaptive Analysis

Michael A. Bender [†] Erik D. Demaine[‡] Roozbeh Ebrahimi[§] Jeremy T. Fineman[¶]
Rob Johnson[†] Andrea Lincoln[‖] Jayson Lynch[‡] Samuel McCauley[†]

ABSTRACT

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 (RAM) is shared across multiple processes, the effective memory-size allocated to each process fluctuates over time.

This paper presents techniques for designing and analyzing algorithms in a cache-adaptive setting, where the RAM available to the algorithm changes over time. These techniques make analyzing algorithms in the cache-adaptive model almost as easy as in the external memory, or DAM model. Our techniques enable us to analyze a wide variety of algorithms — Master-Method-style algorithms, Akra-Bazzi-style algorithms, collections of mutually recursive algorithms, and algorithms, such as FFT, that break problems of size N into subproblems of size $\Theta(N^c)$.

We demonstrate the effectiveness of these techniques by deriving several results:

- We give a simple recipe for determining whether common divide-and-conquer cache-oblivious algorithms are optimally cache adaptive.
- We show how to bound an algorithm's non-optimality. We give a tight analysis showing that a class of cache-oblivious algorithms is a logarithmic factor worse than optimal.
- We show the generality of our techniques by analyzing the cache-oblivious FFT algorithm, which is not covered by the above theorems. Nonetheless, the same general techniques can show that it is at most $O(\log \log N)$ away from optimal in the cache adaptive setting, and that this bound is tight.

These general theorems give concrete results about several algorithms that could not be analyzed using earlier techniques. For example, our results apply to Fast Fourier Transform, matrix multiplication, Jacobi Multipass Filter, and cache-oblivious dynamic-programming algorithms, such as Longest Common Subsequence and Edit Distance.

Our results also give algorithm designers clear guidelines for creating optimally cache-adaptive algorithms.

This research was supported in part by NSF grants CCF 1114809, CCF 1217708, CCF 1218188, CCF 1314633, IIS 1247726, IIS 1251137, CNS 1408695, and CCF 1439084; MADALGO - Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation; and a Stanford Graduate Fellowship.

[†]Department of Computer Science, Stony Brook University, Stony Brook, NY 11794-4400, USA. Email: {bender, rob, smccauley}@cs.stonybrook.edu.

[‡]MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge, MA 02139, USA. Email: {edemaine,jaysonl}@mit.edu.

[§]Google Inc., 1600 Amphitheatre Parkway, Mountain View, CA 94043 USA. Email: rebrahimi@google.com.

[¶]Department of Computer Science, Georgetown University, 37th and O Streets, N.W., Washington D.C. 20057, USA. Email: jfineman@cs.georgetown.edu.

[‖]Department of Computer Science, Stanford University, Palo Alto, CA 94305 USA. Email: andreali@cs.stanford.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPAA '16, July 11 - 13, 2016, Pacific Grove, CA, USA

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4210-0/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2935764.2935798

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 multi-core computers.

External-memory computations 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 [9, 18, 19], and have developed many sorting and join algorithms [15, 20, 21, 25–27] 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 perform poorly in the worst case [3, 4].

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., [24]) 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 other processes, and 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 [3, 4] took the first major step towards closing this gap by showing that worst-case, external-memory bounds are possible in an environment where the cache[1] changes size. Barve and Vitter generalized the DAM (disk-access machine) model [1] to allow the memory size M to change periodically. They give optimal algorithms for sorting, FFT, matrix multiplication, LU decomposition, permutation, and buffer trees. Their work shows that it is possible to specially design intricate algorithms to handle adaptivity, but stops short of giving a general framework.

Bender et al. [7] took the next major step towards closing this gap between theoretical performance models and real systems. They formally define the cache-adaptive model,[2] and prove that some—but not all—optimal cache-oblivious[3] algorithms [13, 14, 22] remain optimal when the cache changes size dynamically. Specifically, if a recursive cache-oblivious algorithm performs $O(1)$ block transfers in addition to its recursive calls, then it is optimally cache adaptive. So is lazy funnel sort [8], despite not fitting this recursive pattern. Bender et al.'s results are encouraging. Because cache-oblivious algorithms are well understood, frequently easy to design, and widely deployed, there is hope that provably good cache-adaptive algorithms can also be deployed in the field.

On the other hand, open questions remain. In particular, the primary contribution of Bender et al. [7] was proposing a computational model rather than an analysis technique. Is there an algorithmic toolkit for cache-adaptive analysis so that future engineers could write their own cache-oblivious algorithms and then quantify their adaptivity? How can we analyze more general forms of recursive algorithms, e.g., in the common case where the additive term is $\omega(1)$? (Examples include cache-oblivious FFT, some versions of cache-oblivious matrix multiply and mutually recursive cache-oblivious dynamic programming algorithms such as LCS [10], Edit Distance [10], and Jacobi Multipass Filter [22].) How can we prove that a recursive algorithm is *not* optimally cache adaptive? For such algorithms, how far are they from optimality? The point of the present paper is to help answer these and other questions.

[1]We use "cache" to refer to the smaller level in any two-level hierarchy. Because this paper emphasizes RAM and disk, we use the terms "internal memory," "RAM," and "cache" interchangeably.

[2]The cache-adaptive model allows memory to change size more rapidly and unpredictably than the model of [3, 4], meaning that cache-adaptive results are more likely to hold up in the real world.

[3]***Cache-oblivious*** algorithms are not parameterized by the memory hierarchy, yet they often achieve provably optimal performance for any *static* hierarchy; see Section 2.

Results

The contribution of this paper is a set of tools that make analyzing the performance of recursive algorithms in the cache-adaptive setting almost as easy as analyzing their performance in the DAM [1] or cache-oblivious/ideal-cache models [13, 14, 22]. Analyzing the performance of many algorithms in the cache-adaptive model boils down to mechanically transforming the recurrence relation for the algorithm's I/O complexity, solving this new recurrence, and comparing the result to a problem-specific lower bound: if these bounds are asymptotically equal, then the algorithm is optimal; if they are not, then their ratio bounds how far the algorithm is from optimal.

Our techniques are general. They can analyze a wide variety of algorithms: Master-method-style algorithms [11], Akra-Bazzi-style algorithms [2], algorithms made of several mutually recursive functions, and algorithms, such as cache-oblivious FFT, that break a problem of size N into subproblems of size $\Theta(N^c)$.

Our results provide easy guidelines for cache-adaptive-algorithm designers. For example, in the case of linear-space-complexity Master-method-style algorithms, our results give an easy rule: if you want your algorithm to be optimal in the cache-adaptive model, then it can perform linear scans of size up to $O(N^c)$, where $c < 1$, in addition to its recursive calls. See Theorem 7.3 for the detailed criteria.

Our results suggest that the problem of designing and analyzing algorithms that adapt to memory fluctuations is tractable. The cache-adaptive model places almost no restrictions on how much and when memory can change size, so results in the model can carry over to most real-world systems. Despite this generality, our method enables algorithm designers to easily evaluate performance in this model.

We demonstrate our techniques by deriving several concrete results. (Below N refers to the input size.)

- We establish that cache-obliviousness does not always lead to cache-adaptivity using a variation of the cache-oblivious naïve matrix multiplication algorithm of Frigo et al. [13]. While this variation is optimal in the DAM model, it is a $\Theta(\log N)$ factor away from optimal in the cache-adaptive model. This result serves as a concrete example of our more general techniques.
- We completely characterize when a Master-method-style linear-space-complexity algorithm is optimal in the cache-adaptive model; see Theorem 7.3. We show that when such an algorithm is DAM-optimal, it is at most an $O(\log N)$-factor slower than optimal in the cache-adaptive model.
- More generally, we completely characterize when a set of mutually recursive linear-space-complexity Akra-Bazzi-style algorithms are optimal in the cache-adaptive model (Theorem 6.10 and Theorem 6.11).
  We apply these theorems to show the cache-adaptive optimality of mutually recursive algorithms such as the cache-oblivious dynamic programming algorithms of Chowdhury and Ramachandran [10] for Longest Common Subsequence and Edit Distance, and Prokop's cache-oblivious Jacobi Multipass Filter Algorithm [22].
- We show that the same techniques can be used to analyze non-Akra-Bazzi-style algorithms. As an example, we show that the cache-oblivious FFT algorithm [13] is $O(\log \log N)$ away from optimal.

Paper overview. The rest of the paper is organized as follows. Section 2 reviews the cache-adaptive model. Section 3 axiomatizes the notion of a progress bound, which are used in many DAM optimality proofs.

Section 4 works through the analysis of a version of the cache-oblivious matrix multiply algorithm of Frigo et al. [13]. This example shows how our results apply to a classic cache-oblivious algorithm, and demonstrates the key ideas behind our techniques. Section 5 generalizes these ideas to an easy-to-use recipe for performing cache-adaptive analysis.

Section 6 gives a high-level overview of the core technical idea of the paper. We show how to bound the progress an algorithm can make on any memory profile. Section 7 describes our main cache-adaptive optimality characterization theorems. We give an optimal cache-adaptivity characterization theorem for collections of mutually recursive Akra-Bazzi-style algorithms. The proofs for the theorems given in Section 7 appear in the full version of this paper.

The paper ends with a summary of concrete results about specific algorithms that can be obtained using our methods.

2. CACHE-ADAPTIVE MODEL, DEFINITIONS, AND ANALYTICAL TOOLS

The ***cache-adaptive*** model [7] is the same as the DAM model [1] except that memory may change size after each I/O (i.e., after each cache miss).[4] Thus, the size of memory is not a constant, but rather a function $m(t)$ giving the size of memory (in blocks) after the tth I/O. We also use $M(t) = B \cdot m(t)$ to represent the size, in words, of memory at time t. We call $m(t)$ and $M(t)$ ***memory profiles*** in blocks and words, respectively.

Optimality in the cache-adaptive model mirrors optimality in the DAM model. On every memory profile, an optimal algorithm has worst-case I/O complexity within a constant factor of any other algorithm's worst-case I/O complexity.

However, in the cache-adaptive model it does not make sense to compare two algorithms' running times directly. That is, given a profile $m(t)$ and two algorithms A and B, we cannot compare A and B's performance by simply running them on $m(t)$ and comparing their running times. To see why, suppose B performs one dummy I/O and then runs A. By any reasonable definition, B's running time should be asymptotically no worse than A's. But consider an input I and profile $m(t)$ that drops to very little memory as soon as $A(I)$ finishes. Now B may finish arbitrarily later than A.

Thus, we formalize optimality by comparing algorithms using ***speed augmentation***. Rather than granting an algorithm extra time, we allow it to perform multiple I/Os per time step in our analysis. Speed augmentation is a theoretical tool to ensure that algorithms are compared on profiles that provide the same overall resources up to a constant factor. Thus, rather than saying one algorithm is (say) twice the speed of another, we say that it performs equally well on hardware with half the latency. This gives the same intuition as classic asymptotic analysis while being meaningful in the cache-adaptive model.

Definition 2.1. *Giving an algorithm A, **c-speed augmentation** means that A may perform c I/Os in each step of the memory profile.*

[4]We use I/Os as a proxy for time because we are studying I/O-bound algorithms—the algorithm spends most of its time performing these I/Os (see [7]).

In order to make the definition of optimality in the CA model as strong as possible, we allow algorithms to query $m(t)$, even into the future. Thus, an optimal algorithm in the CA model is asymptotically as fast as any other algorithm, even one that can see the future size of memory and plan accordingly.[5]

However, allowing algorithms to query the memory profile creates a problem: a P/poly algorithm may be able to use the memory profile as a "reference string" to speed up its computations on some memory profiles. We rule out this behavior by only permitting ***memory-monotone*** algorithms:

Definition 2.2. *A **memory monotone** algorithm runs at most a constant factor slower when given more memory.*

Memory monotonicity is a weak restriction: all cache-oblivious algorithms are memory monotone, as are LRU, the optimal offline paging algorithm, and many other paging algorithms. Memory monotone also includes almost all "reasonable" DAM-model algorithms. One notable exception is FIFO paging [6], which was recently shown not to be memory monotone [12].

Definition 2.3. *An algorithm A that solves problem P is **optimal** in the cache-adaptive model if there exists a constant c such that on all memory profiles and all sufficiently large input sizes N, the worst-case running time of a c-speed-augmented A is no worse than the worst-case running time of any other (non-augmented) memory-monotone algorithm.*

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 2.4. *For any memory profile m, we define a **c-memory augmented** version of m as the profile $m'(t) = cm(t)$. Running an algorithm A with **c-memory augmentation** on the profile m means running A on the c-memory augmented profile of m.*

If an algorithm is not parameterized by M or B, then we say it is ***cache-oblivious***. These algorithms are designed to perform well without knowing the size of memory.

When analyzing cache-oblivious algorithms in the CA model, we assume that the system performs automatic page replacement. Belady's algorithm [5] turns out to be an optimal offline paging algorithm in the CA model and the least-recently-used (LRU) policy is $O(1)$-competitive with resource augmentation [7].

The performance bounds for cache-oblivious algorithms commonly rely on a so-called ***tall-cache*** assumption, which means that there is a value $H(B)$, polynomial in B, such that $M \geq H(B)$. For example, for cache-oblivious sorting or matrix transpose, $H(B) = \Theta(B^2)$ [13]. We support these kinds of analyses in the CA model as follows.

Definition 2.5. *In the CA model, we say that a memory profile M is **H-tall** if for all $t \geq 0$, $M(t) \geq H(B)$.*

Definition 2.6. *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 I is $\Theta(f(N))$.*

Space complexity is often defined as the maximum memory location indexed. This definition is slightly more general.

[5]Note that while our model allows the algorithms to see the future memory, the cache-oblivious algorithms presented here do not take advantage of this—in fact, they are not even aware of the present size of memory.

3. PROGRESS BOUNDS: HOW MUCH AN ALGORITHM CAN DO ON A PROFILE

This section axiomatizes the notion of ***progress bounds***, which are used in many lower-bound proofs in the DAM model, and develops tools to easily port progress bounds from the DAM model to the CA model.

In the DAM model, a ***progress bound*** $\rho(M, T)$ for a problem P gives an upper bound on the amount of progress that any algorithm can make towards solving an instance of P given M words of memory and T I/Os. A ***progress requirement function*** $R(N)$ gives a lower bound on the amount of progress any algorithm must make in order to solve all problem instances of size N. In the DAM model, the I/O complexity of any algorithm must be at least $\Omega(T \cdot R(N)/\rho(M, T))$.

Example 3.1. *The DAM sorting lower bound [1] says that, after sorting each block, a comparison-based sorting algorithm learns at most $O(B \log(M/B))$ bits of information per I/O, given M memory, and must learn $\Omega(N \log N)$ bits to sort. Thus, $R(N) = \Omega(N \log N)$ and $\rho(M, T) = O(TB \log(M/B))$. Hence sorting in the DAM model takes at least $R(N)/\rho(M, 1) = \Omega(\frac{N}{B} \log_{M/B} N)$ I/Os.*

Example 3.2. *The DAM matrix multiplication lower bound [16, 17, 23] states that, given M memory and M/B I/Os, no naive matrix multiplication algorithm can perform more than $O(M^{3/2})$ elementary multiplications, and multiplying two $\sqrt{N} \times \sqrt{N}$ matrices requires performing $\Theta(N^{3/2})$ elementary multiplications. Thus $\rho(M, M/B) = M^{3/2}$ and $R(N) = \Theta(N^{3/2})$. Consequently, multiplying two $\sqrt{N} \times \sqrt{N}$ matrices requires $\frac{M}{B} R(N)/\rho(M, \frac{M}{B}) = \Omega(MN^{3/2}/(BM^{3/2})) = \Omega(\frac{N^{3/2}}{B\sqrt{M}})$ I/Os.*

We first generalize the notion of progress bounds to arbitrary profiles. Given an arbitrary memory profile $M(t)$, we write $\rho(M)$ as the upper bound on progress that any algorithm can make on an instance of problem P when given memory profile $M(t)$.

Most[6] progress bounds developed in the DAM model also apply to the CA model because they are "memory-less", i.e., the bound $\rho(M, t)$ applies no matter what the size or content of memory was before the t I/Os in question. In fact, in the full version of this paper, we give a general method for deriving progress bounds from DAM lower bounds based on the red pebble game [23].

The only challenge to using DAM progress bounds on arbitrary memory profiles in the CA model is that some bounds apply only to time-spans during which memory does not change size. For example, the matrix multiply progress bound is defined only for M/B I/Os during which memory is always of size M.

We solve this problem using ***square profiles***. A profile $m(t)$ is square if time can be decomposed into a sequence of regions $t_0 < t_1 < \ldots$ such that $m(t) = t_{i+1} - t_i$ for all $t \in [t_i, t_{i+1})$. Square profiles are useful because we can apply progress bounds, like the matrix multiply progress bound, to each square independently and sum to get a bound on the progress that any algorithm can make on the entire profile. We use the notation $\square_M$ to denote a square of size M words by M/B I/Os, and we denote a square profile with squares of size $M_1, \ldots, M_k$ as $\square_{M_1} \| \cdots \| \square_{M_k}$. We generalize ρ from the DAM model to squares as $\rho(\square_M) = \rho(M, M/B)$.

[6] In fact, all that we have found.

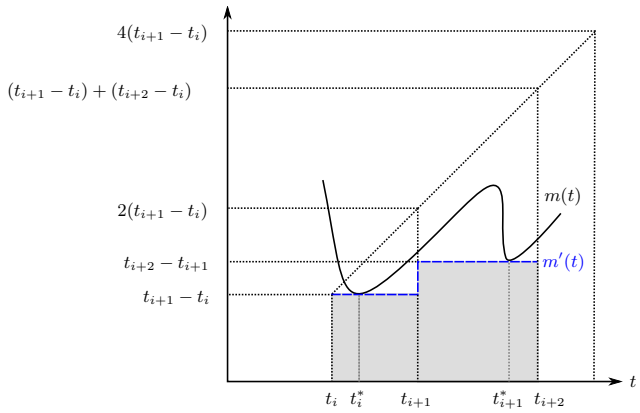

Figure 1: The inner square profile of memory profile $m(t)$.

We can then generalize these bounds to arbitrary profiles by using ***inner square profiles*** [7]. The inner square profile of a profile M is constructed by greedily packing the largest-possible squares under M from left to right, as shown in Figure 1. Square profiles enable us to compute bounds on the amount of progress that any algorithm can make on an arbitrary memory profile.

Definition 3.3. *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 $t_{i+1} - t_i \le m(t)$ 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 $t \in [t_i, t_{i+1})$ (see Figure 1).*

Bender et al. [7] proved the following useful lemma about inner square profiles.

Lemma 3.4. *Let m be a memory profile where $m(t+1) \le m(t) + 1$ for all t. Let $t_0 < t_1 < \ldots$ be the inner square boundaries of m, and m' be the inner square profile of m.*

1. *For all t, $m'(t) \le m(t)$.*
2. *For all i, $t_{i+2} - t_{i+1} \le 2(t_{i+1} - t_i)$.*
3. *For all i and $t \in [t_{i+1}, t_{i+2})$, $m(t) \le 4(t_{i+1} - t_i)$.*

The following definitions axiomatize two technical but obvious properties of progress functions: (1) that profiles with more time and memory can support more progress, and (2) that the progress possible on a square profile is just the sum of the progress possible on each of its squares.

Intuitively, we say that one profile is ***smaller*** than another, i.e., offers less memory and/or time, if it can be cut into pieces, each of which fits underneath a corresponding piece of the other.

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 exist profiles $L_1, L_2 \ldots L_k$ and $U_0, U_1, U_2 \ldots U_k$, such that $M = L_1 \| L_2 \ldots \| L_k$ and $U = U_0 \| U_1 \| U_2 \ldots \| U_k$, and for each $1 \le i \le k$,*
(i) *if d_i is the duration of L_i, U_i has duration $\ge d_i$, and*
(ii) *as standalone profiles, L_i is always below U_i.*

Definition 3.6. *A function $\rho : \mathbb{N}^* \to \mathbb{N}$ is **monotonically increasing** if for any profiles M and U, $M \prec U$ implies $\rho(M) \le \rho(U)$.*

Second, we assume that the progress on a square profile should be, essentially, the sum of the progress possible on each square.

Definition 3.7. *Let $M_1 \| M_2$ indicate concatenation of profiles M_1 and M_2. A monotonically increasing function $\rho : \mathbb{N}^* \to \mathbb{N}$ is* ***square-additive*** *if* ***(i)*** *$\rho(\square_M)$ is bounded by a polynomial in M, and* ***(ii)*** *$\rho(\square_{M_1} \| \cdots \| \square_{M_k}) = \Theta(\sum_{i=1}^{k} \rho(\square_{M_i}))$.*

With these requirements in mind, we can axiomatize the notion of progress bound.

Definition 3.8. *A problem has a* ***progress bound*** *if there exists a monotonically increasing polynomial-bounded* ***progress-requirement function*** $\boldsymbol{R : \mathbb{N} \to \mathbb{N}}$ *and a square-additive* ***progress limit function*** $\boldsymbol{\rho : \mathbb{N}^* \to \mathbb{N}}$ *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 ρ as the ***progress function*** or ***progress bound***.

3.1 Optimally-Progressing Algorithms

In this paper, we prove that algorithms are optimal (or non-optimal) by analyzing whether they always make within a constant factor of the maximum possible progress on a profile. An algorithm is ***optimally progressing*** if, for every usable profile m that is just long enough for the algorithm to solve all problems of size N, $\rho(m) = O(R(N))$. We first define what it means for a profile to be "usable" and "just long enough" and then define optimally progressing formally.

The CA model allows memory to increase or decrease arbitrarily from one time step to the next. However, since an algorithm can only load one block into memory per time step, its memory usage can only increase by one block per time step.

Definition 3.9. *An h-tall memory profile m is* ***usable*** *if $m(0) = h(B)$ and if m increases by at most 1 block per time step, i.e. $m(t+1) \leq m(t) + 1$ for all t.*

Definition 3.10. *For an algorithm A and problem instance I we say a profile M of length ℓ is* ***I-fitting*** *if A requires exactly ℓ time steps to process input I on profile M. A profile M is* ***N-feasible for A*** *if A, given profile M, can complete its execution on all instances of size N. We further say that M is* ***N-fitting for A*** *if it is N-feasible and there exists at least one instance I of size N for which M is I-fitting. (When A is understood, we will simply say that M is N-feasible, N-fitting, etc.)*

Definition 3.11. *For an algorithm A, integer N, and N-feasible profile $M(t)$, let $M_N(t)$ denote the N-fitting prefix of M. We say that algorithm A with tall-cache requirement H is* ***optimally progressing with respect to a progress bound ρ*** *(or simply optimally progressing if ρ is understood) if, for every integer N and N-feasible H-tall usable profile M, $\rho(M_N) = O(R(N))$.*

The following two lemmas show that usable profiles and square profiles support essentially the same amount of progress. This implies that, if a memory-monotone algorithm is optimally progressing on all usable profiles, then it is optimally progressing on all square profiles, and vice versa. This enables us to focus exclusively on square profiles, which are easier to analyze, when proving algorithms optimal (or non-optimal) in the CA model.

Lemma 3.12. *If ρ is square additive and M is a usable profile with inner square profile M', then $\rho(M) = \Theta(\rho(M'))$.*

Proof. Since ρ 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 ρ is square-additive and since $\rho(\square_N)$ is bounded by a polynomial in N, $\rho(M'_{4,4}) = O(\rho(M'))$.

We prove that $M \prec M'_{4,4}$. Thus, by monotonicity of ρ,

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

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 inner 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 $M = L_1 \| L_2 \| \ldots \| L_k$. Also, define U_i to be a 4-speed 4-memory augmented version of square S_i and allow $U'_k = U_k \| U_{k+1}$. Notice that $M'_{4,4} = U_1 \| U_2 \| \ldots U_{k-1} \| U'_k$.

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

We start by considering U_1 and L_1. If M is $H(B)$-tall, by Definition 3.9 we have that $M(0) = H(B)$. By Definition 3.3, 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 3.4, 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.

Similarly, for each U_i, $2 \leq i \leq k$, by Lemma 3.4, 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 $U'_k = U_k \| U_{k+1}$ also satisfies both of the above conditions. Therefore, we have shown that $M \prec M'_{4,4}$ and the lemma is complete. □

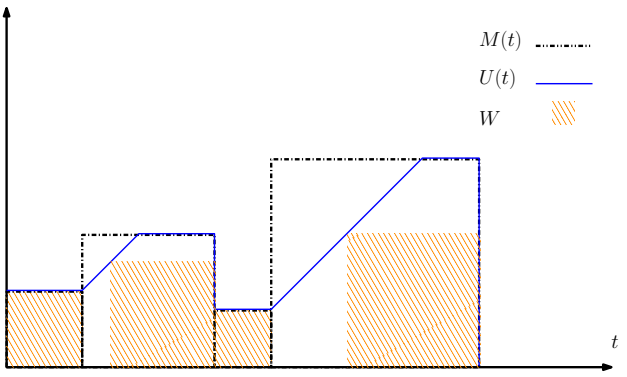

Figure 2: The usable profile beneath each square profile.

Lemma 3.13. *Let ρ be square additive. For every H-tall square memory profile M, there exists a* usable *memory profile U below M such that $\rho(U) = \Theta(\rho(M))$.*

Proof. Let $M = \square_{M_1} \| \square_{M_2} \| \dots \square_{M_k}$ be any square profile. We construct a usable profile U as follows. We set $U(x) = H(B) + x$ for $x \in [0, \leq M_1 - H(B)]$ and $U(x) = M_1$ for $x \in (M_1 - H(B), M_1)$. For $i \geq 2$, U does the following on the interval $[t_i, t_i + M_i)$ covered by the ith square of M,

(i) If $M_i \leq M_{i-1}$, we let $U(x) = M_i$ for $x \in [t_i, t_i + M_i)$.
(ii) If $M_i > M_{i-1}$, let $U(x) = M_{i-1} + x$ for $x \in [t_i, t_i + M_i - M_{i-1}]$ and $U(x) = M_i$ for $x \in (t_i + M_i - M_{i-1}, t_i + M_i)$.

See Figure 2 for an illustration. Clearly we have $U \prec M$, so by monotonocity of ρ, 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 $\square_{W_i}$, that all fit below U and for each i, $\square_{W_i}$ is at most 2 times shorter than $\square_{M_i}$. Since each $\square_{W_i}$ fits below U, we have that $W \prec U$ where $W = \square_{W_1} \| \square_{W_2} \| \dots \square_{W_k}$. On the other hand, since ρ is square-additive, ρ is bounded by a polynomial and thus $\rho(\square_{W_i}) = \Theta(\rho(\square_{M_i}))$. Square-additivity of ρ also means that $\rho(W) = \Theta(\rho(M))$. Since $\rho(W) \leq \rho(U)$ the statement follows.

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

Thus, between Lemma 3.12 and Lemma 3.13 we are able to use square profiles for both upper and lower bounds without loss of generality.

Finally, our proofs focus on showing that an algorithm is optimally progressing. This lemma justifies this focus—showing that an algorithm is optimally progressing is sufficient to show that it is cache-adaptive.

Lemma 3.14. *If an algorithm A is optimally progressing, then it is optimally cache adaptive.*

While optimally progressing is sufficient for adaptivity, we do not know if it is necessary—it remains an open question if there are algorithms that are optimally cache-adaptive but not optimally progressing. On the other hand, *progress optimality* and *competitive optimality* are equivalent in the DAM model. (See the full version for the proof.)

4. MATRIX MULTIPLY: A TALE OF TWO ALGORITHMS

This section provides a concrete example of our approach by working through the analysis of two cache-oblivious matrix multiplication algorithms.

We analyze two variants of the cache-oblivious matrix multiplication algorithm of Frigo et al. [13], which we call MM-Inplace and MM-Scan. Both algorithms divide each input matrix into four submatrices and perform eight recursive calls to compute submatrix products. Both run in $O(N^{3/2}/\sqrt{M}B)$ I/Os in the DAM model, which is optimal [16, 17].

Remarkably, only MM-inplace is optimally cache adaptive; MM-scan is a $\Theta(\log(N/B^2))$ factor away from being optimal in the cache-adaptive model. This shows that cache-oblivious algorithms are not always cache adaptive.

The two algorithms differ in how they combine the eight matrix sub-products. 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. MM-Inplace computes the eight matrix sub-products "in place," adding the results of elementary multiplications into the output matrix as soon as it computes them, and yielding a recurrence of $T(N) = 8T(N/4) + O(1)$ in the DAM model. Both algorithms require a tall cache of $\Theta(B^2)$ and both recurrences have the same solution in the DAM model.

However, the linear scans of MM-Scan are wasteful if they execute when memory is plentiful. For each input size N, we can construct a profile W_N^* that causes MM-Scan to run inefficiently by giving the algorithm lots of memory during its scans, when it can't use it, and very little memory at other times. As a result, the profile W_N^* will have a recursive structure that mimics that of MM-Scan: where MM-Scan performs a linear scan of size $\Theta(N)$, W_N^* will contain a square of size $\Theta(N)$. When MM-Scan performs a recursive call on a problem of size $N/4$, W_N^* will have a copy of $W_{N/4}^*$. Hence $\rho(W_N^*)$, the progress possible on W_N^*, will satisfy a recurrence relation very similar to the recurrence relation for MM-Scan. Where MM-Scan's recurrence relation has a term of the form $\Theta(N/B)$, corresponding to a linear scan of size N, $\rho(W_N^*)$'s recurrence will have a term of the form $\rho(\square_N)$. This will enable us to solve explicitly for $\rho(W_N^*)$, yielding a concrete counter-example to the optimality of MM-Scan.

Theorem 4.1. *The cache-oblivious matrix multiplication algorithm* MM-Scan *is a $\Omega(\log(N/B^2))$ factor away from being optimal when solving problem instances of size N.*

Sketch. We show in Theorem 7.3 that MM-Inplace is optimally progressing; intuitively, this is because nearly all of its computation time is spent in recursive calls.

The construction of W_N^* works as follows. Let $\lambda = \Theta(B^2)$ be the tall cache requirment for MM-Scan. Memory starts out at size λ. Whenever MM-Scan starts a linear scan of size $L \geq 2\lambda$ with λ memory, it will necessarily incur $\Theta((L-\lambda)/B) = \Theta(L/B)$ page faults, no matter how memory changes size during the scan. Thus we can increase memory to size $\Theta(L)$ for the next $\Theta(L/B)$ time steps. At the last time step of the linear scan, we drop memory back to λ.

We now compute $\rho(W_N^*)$. From Example 3.2, no naive matrix multiply algorithm can do more than $O(M^{3/2})$ elementary multiplications given M words of memory and M/B I/Os [23]. Thus $\rho(\square_M) = \Theta(M^{3/2})$. Since the top-level recursion performs a linear scan of size $\Theta(N)$, W_N^* contains a square of size $\Theta(N)$, which contributes $\Theta(\rho(\square_N))$ to $\rho(W_N^*)$. Further, W_N^* contains eight copies of $W_{N/4}^*$—one for each recursive call made by MM-Scan. Thus $\rho(W_N^*)$ satisfies the recurrence

$$\rho(W_N^*) = 8\rho(W_{N/4}^*) + \Theta(\rho(\square_N)) = 8\rho(W_{N/4}^*) + \Theta(N^{3/2})$$

with base case $\rho(W_N^*) = \Theta(B^{3/2})$ when $N = O(B^2)$. Thus $\rho(W_N^*) = \Theta(N^{3/2}\log(N/B^2))$. Since MM-Scan finishes at the last time step of W_N, W_N is N-fitting for MM-Scan.

A problem of size N requires $R(N) = \Theta(N^{3/2})$ total multiplications. Since MM-Scan only makes $O(N^{3/2})$ progress

on W_N^* but W_N^* supports $\Theta(N^{3/2}\log(N/B^2))$ progress, MM-Scan is $\Omega(\log(N/B^2))$ less than optimally progressing. □

5. A GENERAL RECIPE FOR ANALYZING ALGORITHMS IN THE CA MODEL

We now explain how to generalize the construction from Section 4 to obtain a simple recipe for testing whether a recursive linear-space cache-oblivious algorithm is optimally progressing.

The method from Section 4 can be used to construct a profile W_N^* for any cache-oblivious algorithm. Whenever $\rho(W_N^*) = \omega(R(N))$, the algorithm is not optimal.

The main technical challenge of this paper is to show that W_N^* is, for many algorithms, the worst possible profile, up to constant factors. Thus, an algorithm is optimally progressing if and only if $\rho(W_N^*) = O(R(N))$.

Since $\rho(W_N^*)$ satisfies, by construction, a recurrence that we can easily derive from the program's structure, this gives us an easy way to analyze Master-method-style cache-oblivious algorithms in the CA model.

1. Write down the recurrence for $T(N)$, the algorithm's I/O complexity in the DAM model:
$$T(N) = aT(N/b) + \Theta(N^c/B).$$
2. Derive the recurrence for $\rho(W_N)$ by replacing terms of the form $T(X)$ with $\rho(W_X)$ and terms corresponding to linear scans of size X with $\rho(\square_X)$:
$$\begin{array}{c} T(N) = aT(N/b) + \Theta(N^c/B) \\ \Downarrow \\ \rho(W_N) = a\rho(W_{N/b}) + \Theta(\rho(\square_{N^c})). \end{array}$$
3. Solve the recurrence for $\rho(W_N)$.
4. Compare the solution to $R(N)$. If $\rho(W_N) = O(R(N))$, then the algorithm is optimal. Otherwise, their ratio bounds how far the algorithm is from optimal.

In fact, we can explicitly solve the recurrence to obtain a simple theorem characterizing when linear-space complexity cache-oblivious Master-method-style algorithms are optimally progressing in the CA model (see Theorem 7.2).

The same basic technique works for Akra-Bazzi-style algorithms and even for collections of mutually-recursive Akra-Bazzi-style algorithms (see Theorem 6.10). Although Theorem 6.10 looks complex, the basic idea is the same.

1. Write down the recurrence relation for the I/O complexity of the algorithm.
2. Derive three new recurrences by performing the transformations specified in Theorem 6.10.
3. Solve these recurrences and compare to $R(N)$.

We explore these generalizations in Sections 6 and 7.

6. BOUNDING THE WORST-CASE PROFILE

This section formalizes the recipe described in Section 5. We first define the structure of algorithms covered by our theorems and then prove bounds on the progress possible on their worst-case profiles.

We first need to formalize the notion of a ***linear scan***. 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 6.1. *We say that algorithm L is a **linear scan of size ℓ** if it accesses ℓ 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 might not access every element of its input (e.g., a search for a specific item in an array), it might not access the pages in sequential order (e.g., cache-oblivious matrix transpose [13]), and the order of accesses can be data-dependent (e.g., the merge operation from merge-sort).

Note that some linear scans and algorithms are I/O efficient only under a tall-cache assumption. For example, for cache-oblivious sorting or matrix transpose, $H(B) = \Theta(B^2)$ [13]. Thus the definition of a scan depends implicitly on the memory profile.

A particular type of linear scans only access a small number of locations. This may occur when, for example, an algorithm does $O(1)$ work at the beginning of a recursive call to set up the computation, or when recursive calls decrease the size of the linear scan below the block size.

Definition 6.2. *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.*

We can now define the class of algorithms covered by our theorems. Briefly, these algorithms are collections of mutually recursive Akra-Bazzi-style algorithms, i.e. they make a constant number of recursive calls on sub-problems a constant factor smaller than their input, and perform linear scans. This class covers everything from simple algorithms, such as the matrix multiplication algorithms described earlier, to advanced cache-oblivious algorithms, such as the cache-oblivious longest-common-subsequence (LCS) and Edit Distance algorithms of Chowdhury and Ramachandran [10], and the cache-oblivious Jacobi Multipass Filter algorithm of Prokop [22].

Definition 6.3. *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 $i = 1, \ldots, f_j$. Algorithms $A_1, \ldots, A_e$ are **generalized compositional regular (GCR) algorithms** if, for all i, A_j on an input of size N*

- **(i)** *makes $\boldsymbol{a_{ji}}$ calls to algorithm $\boldsymbol{A_{ji}}$ on subproblems of size $\boldsymbol{b_{ji}N}$. Algorithm A_{ji} is one of $A_1, \ldots, A_e$.*
- **(ii)** *performs $\Theta(1)$ linear scans before, between, or after its calls, where 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 $\Theta(N^{c_j})$.*

We can now define the worst-case profile for an algorithm.

Definition 6.4. *Algorithm **A's worst-case profile for inputs of size N** among all profiles that are λ-tall is*

$$W_{A,N,\lambda} = \text{argmax}\{\rho(M) \mid M \text{ is } N\text{-fitting, } \lambda\text{-tall, usable}\}.$$

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

$$W_{A,N} = W_{A,N,H} = \operatorname{argmax}\{\rho(M) \mid M \text{ is } N\text{-fitting, } H\text{-tall, usable}\}. \quad (1)$$

The following lemma, which follows directly from the definitions, enables us to analyze algorithms by looking at only their worst-case profiles.

Lemma 6.5. *If $\rho(W_{A,N,\lambda}) = O(R(N))$, then A is optimally progressing on all λ-tall profiles.*

When combined with Lemma 3.12, the above lemma means we can analyze algorithms by looking at only their worst-case square profiles.

The worst-case profile, 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.

The following definition defines three conditions under which the progress from a square can be charged to a linear scan, recursive call, or overhead reference.

Definition 6.6. *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.*

We now define when we can charge every square of a profile to a linear scan, recursive call, or overhead reference.

Definition 6.7. *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 ℓ 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 ℓ steps to process.*

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

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

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

Finally, we prove that, for GCR algorithms with linear space complexity, every profile is N-chargeable.

Lemma 6.8. *Let e be a constant and let $A_1, \ldots, A_e$ 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.*

To see why, consider a square of size S that does not overlap a linear scan of size $\Omega(S)$. Such a square must contain entire executions that access $\Theta(S)$ distinct locations. And since all the algorithms have linear space complexity, any execution that touches $\Theta(S)$ distinct locations must be on a subproblem of size $\Theta(S)$. (See the full version for the complete proof.)

Finally, we bound the size of boxes that contain mostly overhead references, which means that they can be ignored when analyzing many algorithms.

Lemma 6.9. *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 6.7 has size $O\left(B \log_{1/q} X(S)\right)$.*

These two lemmas give us the "recurrence-rewriting" rule outlined in Section 5. The progress on each square of a profile can be charged to either (1) a linear scan of size at least as large as the square, (2) a subproblem of size proportional to the square, or (3) overhead references, although in the last case the square cannot be very large. Theorem 6.10 (at the top of the next page) summarizes this result.

The following theorem, proven in the full version, gives a corresponding lower bound on the progress of the worst-case profile. While Theorem 6.10 can show that an algorithm is optimally progressing, Theorem 6.11 gives a recipe to prove that an algorithm is not optimally progressing.

Theorem 6.11. *Suppose $A_1, \ldots, A_e$ are generalized compositional regular algorithms with linear space complexity, tall-cache requirement $H(B)$, and progress bound ρ. Let $\lambda = \max\{H(B), (B \log_{1/b} B)^{1+\varepsilon}\}$, where ε is any constant larger than 0. When all $c_j = 1$, the bound in Theorem 6.10 is tight, meaning that $\rho(W_{A_j,N,\lambda})$, is $\Theta(\mathcal{T}_j(N) + \mathcal{U}_j(N) + \mathcal{V}_j(N))$.*

7. OPTIMALITY CRITERIA FOR MANY MASTER-METHOD-STYLE ALGORITHMS

Theorem 6.10, Theorem 6.11, and Lemma 6.5 provide an easy way to test whether a linear-space GCR algorithm is optimally progressing: derive the recurrences on $\mathcal{T}_i$, $\mathcal{U}_i$, and $\mathcal{V}_i$ from the structure of the algorithm, solve the recurrences, and check whether $\mathcal{T}_i(N) + \mathcal{U}_i(N) + \mathcal{V}_i(N) = O(R(N))$.

Although we can't give a general solution for all possible $\mathcal{T}_i$, $\mathcal{U}_i$, and $\mathcal{V}_i$, we can use this method to derive a simple test for Master-method-style algorithms.

We first define the class of Master-theorem-style algorithms covered by our optimality criterion. Note that this class is more general than the constant-overhead recursive (COR) form algorithms covered by previous work [7]. Note also that $c \leq 1$ is implied by the linear space restriction.

Definition 7.1. *Let $a \geq 1/b$, $0 < b < 1$, and $0 \leq c \leq 1$ be constants. A linear-space 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) *$\Theta(1)$ linear scans before, between, or after recursive calls, where the size of the biggest scan is $\Theta(N^c)$.*

We will prove that a DAM-optimal linear-space-complexity (a, b, c)-regular algorithm is optimal in the CA model if and only if $c < 1$. We first solve the recurrence from Theorem 6.10 for the special case of (a,b,c)-regular algorithms to get a simple bound on $\rho(W_N)$.

Theorem 6.10. *Let $0 \le c_j \le 1$ and $f_j \ge 1$ 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 ρ.*
Let $b = \max\{b_{ji}\}$ and $\lambda \ge H(B)$ be constants. Then there exist functions $\mathcal{T}_1, \ldots, \mathcal{T}_e$; $\mathcal{U}_1, \ldots, \mathcal{U}_e$; $\mathcal{V}_1, \ldots, \mathcal{V}_e$ such that the progress of the worst-case λ-tall profile for A_j, $\rho(W_{A_j,N,\lambda})$, is $O(\mathcal{T}_j(N) + \mathcal{U}_j(N) + \mathcal{V}_j(N))$ and the $\mathcal{T}_j$, $\mathcal{U}_j$ and $\mathcal{V}_j$ satisfy the recurrences

$$\mathcal{T}_j(N) = \begin{cases} \max\left\{ \rho\left(\square_N\right), \sum_{i=1}^{f_j} a_{ji} \mathcal{T}_{ji}(b_{ji} N) \right\} & \text{if } \lambda < N \\ \Theta(\rho(\square_\lambda)) & \text{if } N \le \lambda; \end{cases}$$

$$\mathcal{U}_j(N) = \begin{cases} \Theta\left(\rho(\square_{N^{c_j}})\right) + \sum_{i=1}^{f_j} a_{ji} \mathcal{U}_{ji}(b_{ji} N) & \text{if } N = \Omega(\lambda) \text{ and } N^{c_j} = \Omega(\lambda) \\ \sum_{i=1}^{f_j} a_{ji} \mathcal{U}_{ji}(b_{ji} N) & \text{if } N = \Omega(\lambda) \text{ and } N^{c_j} \ne \Omega(\lambda) \\ 0 & \text{if } N \ne \Omega(\lambda); \end{cases}$$

$$\mathcal{V}_j(N) = \begin{cases} \sum_{i=1}^{f_j} a_{ji} \mathcal{V}_{ji}(b_{ji} N) & \text{if } B \log_{1/b} N = \Omega(\lambda) \text{ and } N^{c_j} > B \\ \Theta(\rho(\square_{B \log_{1/b} N})) + \sum_{i=1}^{f_j} a_{ji} \mathcal{V}_{ji}(b_{ji} N) & \text{if } B \log_{1/b} N = \Omega(\lambda) \text{ and } N^{c_j} \le B \\ 0 & \text{if } B \log_{1/b} N \ne \Omega(\lambda). \end{cases}$$

where $\mathcal{T}_{ji}$, $\mathcal{U}_{ji}$ and $\mathcal{V}_{ji}$ are one of $\mathcal{T}_1, \ldots, \mathcal{T}_e$; $\mathcal{U}_1, \ldots, \mathcal{U}_e$; $\mathcal{V}_1, \ldots, \mathcal{V}_e$ depending on the structure of A_j.

Theorem 7.2. *Let A be an (a, b, c)-regular algorithm with linear space complexity and tall-cache requirement $H(B)$. Suppose that A is optimal in the DAM model for a problem with progress bound $\rho(\square_X) = \Theta(X^p)$, where p is a constant. Assume that $B \ge 4$. Pick an $\varepsilon > 0$, and let $d = 3(1+\varepsilon)$ and $\lambda = \max\{H(B), (dB \log_{1/b} B)^{1+\varepsilon}\}$.*
Then, $\rho(W_{A,N,\lambda})$ is bounded by $O(\mathcal{X}(N))$, where

$$\mathcal{X}(N) = \begin{cases} \Theta\left(N^{\log_{1/b} a} \log_{1/b} \frac{N}{\lambda}\right) & \text{if } c = 1 \\ \Theta\left(N^{\log_{1/b} a}\right) & \text{otherwise.} \end{cases}$$

The following rule for optimality of (a, b, c)-regular algorithms can be derived by comparing $\mathcal{X}(N)$ with $R(N)$.

Theorem 7.3. *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 $B \ge 4$. Let $\lambda = \max\{H(B), ((1+\varepsilon) B \log_{1/b} B)^{1+\varepsilon)}\}$, where $\varepsilon > 0$.*

1. *If $c < 1$, then A is optimally progressing and optimally cache-adaptive among all λ-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.*

8. APPLYING OUR TECHNIQUES

The techniques presented in this paper provide cookbook methods for designing and analyzing a wide variety of cache-adaptive algorithms. Figure 3 summarizes the result of analyzing several algorithms using this method. In addition to analyzing algorithms that did not fit the requirements of previous analysis techniques, our approach enables us improve on previous results by reducing the tall-cache requirements for some algorithms.

The analysis of FFT is similar to the technique described above, but requires a separate proof because FFT breaks problems of size N into subproblems of size $\sqrt{N}$. The proof uses the same idea: we determine the worst-case profile using a charging argument, and use it to analyze the algorithm's performance. This shows that the technique is useful even when Theorem 6.10 does not apply. See the full version for complete proofs of all these results.

Algorithm	Tall Cache	Ratio to Optimal
Recursive-LCS	$O((B \log_2 B)^{1+\varepsilon})$	$\Theta(1)$
Edit-Distance	$O((B \log_2 B)^{1+\varepsilon})$	$\Theta(1)$
Jacobi	$O((B \log_2 B)^{1+\varepsilon})$	$\Theta(1)$
MM-Inplace	$O(B^2)$	$\Theta(1)$
FW-APSP	$O(B^2)$	$\Theta(1)$
M-Transpose	$O(B^2)$	$\Theta(1)$
FFT	$O(B^2)$	$\Theta(\log\log(N/B^2))$

Figure 3: Adaptivity of several cache-oblivious algorithms. All but FFT can be analyzed using the technique described above. The analysis of FFT is similar, but requires a separate analysis because FFT breaks problems of size N into subproblems of size $\sqrt{N}$.

9. CONCLUSION

This paper revisits one of the most important, but also most difficult, problems in the design, analysis, and deployment of external-memory algorithms. We show that designing and analyzing cache-adaptive algorithms is tractable.

But our results have broader implications for the cache adaptive model itself.

In order for a computational model to be truly useful, (1) it needs to capture an important phenomenon that people care about, (2) its predictions have to be true-to-life,

and (3) it has to be simple enough to work with. There are dozens of theoretical models capturing different aspects of memory-hierarchy performance. The DAM [1] and cache-oblivious [13, 14, 22] models have been so successful because they satisfy these criteria so effectively.

The cache-adaptive model is already known to satisfy two of these criteria. It captures the important phenomenon of changes in cache size, which can strongly affect the performance of algorithms on concurrent systems. It is true to life because it imposes no unrealistic restrictions on changes in the size of memory.

We believe that, by making it easy to analyze many algorithms in the cache-adaptive model, the tools developed in this paper provide the final piece necessary for the cache-adaptive model to join the ranks of truly useful models, such as the DAM and cache-oblivious models.

10. REFERENCES

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

[2] M. Akra and L. Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2):195–210, 1998.

[3] R. Barve and J. S. Vitter. External memory algorithms with dynamically changing memory allocations. Technical report, Duke University, 1998.

[4] R. D. Barve and J. S. Vitter. A theoretical framework for memory-adaptive algorithms. In *Proc. 40th Annual Symposium on the Foundations of Computer Science (FOCS)*, pages 273–284, 1999.

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

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

[7] M. A. Bender, R. Ebrahimi, J. T. Fineman, G. Ghasemiesfeh, R. Johnson, and S. McCauley. Cache-adaptive algorithms. In *Proc. 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 958–971, 2014.

[8] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proc. 29th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 426–438. Springer-Verlag, 2002.

[9] K. P. Brown, M. J. Carey, and M. Livny. Managing memory to meet multiclass workload response time goals. In *Proc. 19th International Conference on Very Large Data Bases (VLDB)*, pages 328–328. Institute of Electrical & Electronics Engineers (IEEE), 1993.

[10] R. A. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *Proc. 17th annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 591–600. ACM, 2006.

[11] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, 2001.

[12] P. Fornai and A. Iványi. FIFO anomaly is unbounded. *CoRR*, abs/1003.1336, 2010.

[13] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annual Symposium on the Foundations of Computer Science (FOCS)*, pages 285–298, 1999.

[14] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4, 2012.

[15] G. Graefe. A new memory-adaptive external merge sort. Private communication, July 2013.

[16] J.-W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proc. 13th Annual ACM Symposium on the Theory of Computation (STOC)*, pages 326–333, 1981.

[17] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, 2004.

[18] R. T. Mills. *Dynamic adaptation to CPU and memory load in scientific applications*. PhD thesis, The College of William and Mary, 2004.

[19] R. T. Mills, A. Stathopoulos, and D. S. Nikolopoulos. Adapting to memory pressure from within scientific applications on multiprogrammed cows. In *Proc. 8th International Parallel and Distributed Processing Symposium (IPDPS)*, page 71, 2004.

[20] H. Pang, M. J. Carey, and M. Livny. Memory-adaptive external sorting. In *Proc. 19th International Conference on Very Large Data Bases (VLDB)*, pages 618–629. Morgan Kaufmann, 1993.

[21] H. Pang, M. J. Carey, and M. Livny. Partially preemptible hash joins. In *Proc. 5th ACM SIGMOD International Conference on Management of Data (COMAD)*, page 59, 1993.

[22] H. Prokop. Cache oblivious algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[23] J. E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1997.

[24] J. S. Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2006.

[25] H. Zeller and J. Gray. An adaptive hash join algorithm for multiuser environments. In *Proc. 16th International Conference on Very Large Data Bases (VLDB)*, pages 186–197, 1990.

[26] W. Zhang and P.-A. Larson. A memory-adaptive sort (MASORT) for database systems. In *Proc. 6th International Conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, pages 41–. IBM Press, 1996.

[27] W. Zhang and P.-A. Larson. Dynamic memory adjustment for external mergesort. In *Proc. 23rd International Conference on Very Large Data Bases (VLDB)*, pages 376–385. Morgan Kaufmann Publishers Inc., 1997.