Closing the Gap Between Cache-oblivious and Cache-adaptive Analysis

Michael A. Bender
Rezaul A. Chowdhury
Rathish Das
Stony Brook University
Stony Brook, NY, USA
{bender,rezaul,radas}@cs.stonybrook.edu

Rob Johnson
VMware Research
Palo Alto, CA, USA
robj@vmware.com

William Kuszmaul
Andrea Lincoln
Quanquan C. Liu
Jayson Lynch
Helen Xu
MIT CSAIL
Cambridge, MA, USA
{kuszmaul,andreali,quanquan}@mit.edu
{jaysonl,hjxu}@mit.edu

ABSTRACT

Cache-adaptive analysis was introduced to analyze the performance of an algorithm when the cache (or internal memory) available to the algorithm dynamically changes size. These memory-size fluctuations are, in fact, the common case in multi-core machines, where threads share cache and RAM. An algorithm is said to be efficiently cache-adaptive if it achieves optimal utilization of the dynamically changing cache.

Cache-adaptive analysis was inspired by cache-oblivious analysis. Many (or even most) optimal cache-oblivious algorithms have an (a, b, c)-regular recursive structure. Such (a, b, c)-regular algorithms include Longest Common Subsequence, All Pairs Shortest Paths, Matrix Multiplication, Edit Distance, Gaussian Elimination Paradigm, etc. Bender et al. (2016) showed that some of these optimal cache-oblivious algorithms remain optimal even when cache changes size dynamically, but that in general they can be as much as logarithmic factor away from optimal. However, their analysis depends on constructing a highly structured, worst-case memory profile, or sequences of fluctuations in cache size. These worst-case profiles seem fragile, suggesting that the logarithmic gap may be an artifact of an unrealistically powerful adversary.

We close the gap between cache-oblivious and cache-adaptive analysis by showing how to make a smoothed analysis of cache-adaptive algorithms via random reshuffling of memory fluctuations. Remarkably, we also show the limits of several natural forms of smoothing, including random perturbations of the cache size and randomizing the algorithm's starting time. Nonetheless, we show that if one takes an arbitrary profile and performs a random shuffle on when "significant events" occur within the profile, then the shuffled profile becomes optimally cache-adaptive in expectation, even when the initial profile is adversarially constructed.

These results suggest that cache-obliviousness is a solid foundation for achieving cache-adaptivity when the memory profile is not overly tailored to the algorithm structure.

CCS CONCEPTS

• **Theory of computation → Caching and paging algorithms**; **Parallel algorithms**.

KEYWORDS

Cache-adaptive algorithms; smoothed analysis; cache-oblivious algorithms

ACM Reference Format:
Michael A. Bender, Rezaul A. Chowdhury, Rathish Das, Rob Johnson, William Kuszmaul, Andrea Lincoln, Quanquan C. Liu, Jayson Lynch, and Helen Xu. 2020. Closing the Gap Between Cache-oblivious and Cache-adaptive Analysis. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '20), July 15–17, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3350755.3400274

1 INTRODUCTION

On multi-threaded and multi-core systems, the amount of cache available to any single process constantly varies over time as other processes start, stop, and change their demands for cache. On most multi-core systems, each core has a private cache and the entire system has a cache shared between cores. A program's fraction of the private cache of a core can change because of time-sharing, and its fraction of the shared cache can change because multiple cores use it simultaneously [14, 23, 24].

Cache-size changes can be substantial. For example, there is frequently a winner-take-all phenomenon, in which one process grows to monopolize the available cache [25]; researchers have suggested periodically flushing the cache to counteract this effect [57]. With this policy, individual processes would experience cache allocations that slowly grow to the maximum possible size, then abruptly crash down to 0.

Furthermore, even small cache-size changes can have catastrophic effects on the performance of algorithms that are not designed to handle them. When the size of cache shrinks unexpectedly, an algorithm tuned for a fixed-size cache can thrash, causing its performance to drop by orders of magnitude. (And if the cache grows, an algorithm that assumes a fixed cache size can leave performance on the table.)

This is such an important problem that many systems provide mechanisms to manually control the allocation of cache to different processes. For example, Intel's Cache Allocation Technology [46] allows the OS to limit each process's share of the shared processor cache. Linux's cgroups mechanism [43] provides control over each application's use of RAM (which serves as a cache for disk). Although these mechanisms can help avoid thrashing, they require manual tuning and can leave cache underutilized. Furthermore, systems may be forced to always leave some cache unused in order to be able to schedule new jobs as they arrive.

A more flexible approach is to solve this problem in the algorithms themselves. If algorithms could gracefully handle changes in their cache allocation, then the system could always fully utilize the cache. Whenever a new task arrives, the system could reclaim some cache from the running tasks and give it to the new task, without causing catastrophic slowdowns of the older tasks. When a task ends, its memory could be distributed among the other tasks on the system. The OS could also freely redistribute cache among tasks to prioritize important tasks.

Practitioners have proposed many algorithms that heuristically adapt to cache fluctuations [13, 31, 44, 45, 47, 48, 63–65]. However, designing algorithms with guarantees under cache fluctuations is challenging and most of these algorithms have poor worst-case performance.

Theoretical approaches to adaptivity. Barve and Vitter [2, 3] initiated the theoretical analysis of algorithms under cache fluctuations over twenty years ago by generalizing the external-memory/disk-access machine (DAM) model [1] to allow the cache size to change. They gave optimal algorithms under memory fluctuations for sorting, FFT, matrix multiplication, LU decomposition, permutation, and buffer trees. In their model, the cache can change size only periodically and algorithms know the future size of the cache and adapt explicitly to these forecasts.

Writing programs and analyzing algorithms that explicitly adapt to changing memory is complicated because the algorithm needs to pay attention to the changing parameter of cache sizes. Moreover, it's hard to have performance guarantees that apply to all possible ways that memory can change size. Thus, most prior work by practitioners is empirical without guarantees, and even the Barve and Vitter work only has guarantees for a restricted class of memory profiles, which limits its generality.

More recently, Bender et al. [5, 6] proposed using techniques from cache-oblivious algorithms to solve the adaptivity problem. Since cache-oblivious algorithms are oblivious to the size of the cache, it is compelling that the algorithms should work well when the cache size changes dynamically. They defined the cache-adaptive model, which is akin to the ideal-cache model [28, 29] from cache-oblivious analysis, except that the size of memory can change arbitrarily over time. They showed that many cache-oblivious algorithms remain optimal even when the size of cache changes dynamically. However, they also showed that some important cache-oblivious algorithms are *not* optimal in the cache-adaptive model.

Concretely, they define optimality in terms of how much progress an algorithm makes under a given ***memory profile*** and they also show that only a restricted class of memory profiles need to be considered. A memory profile $m(t)$ is a function specifying the size of memory at each time t. Prior results show that, for cache-oblivious algorithms, and up to a constant factor of resource augmentation, we need only consider ***square profiles***, i.e., memory profiles which can be decomposed into a sequence of boxes $(\square_1, \square_2, \ldots)$, where a box of size x means that memory remains at size x blocks for x time steps. In its strongest form, cache adaptivity requires that for an algorithm $\mathcal{A}$, the total amount of progress that $\mathcal{A}$ makes on a series of boxes $(\square_1.\square_2, \ldots)$ should be within a constant factor of the total potential $\sum_i \rho(|\square_i|)$ of those boxes, where the potential $\rho(|\square_i|)$ of a box $\square_i$ is defined to be the maximum number of operations that $\mathcal{A}$ could possibly perform in $\square_i$, where the max is taken over all possible places that $\square_i$ could occur in the execution of $\mathcal{A}$.[1]

Thus, Bender et al.'s results show that cache-obliviousness is a powerful technique for achieving adaptivity without the burden of having to explicitly react to cache-size changes. They define optimality in terms of worst-case, adversarial memory profiles, which makes their optimality criteria very strong, but also tough to meet. It's natural to ask how algorithms perform under less adversarial profiles. This is important because for a large class of cache-oblivious algorithms, there exists highly structured and pernicious worst-case profiles on which the algorithms do *not* run optimally.

Cache-oblivious algorithms and (a, b, c)-regularity. One of the fundamental insights in the design of cache-oblivious algorithms is that, by using certain basic recursive structures in the design of an algorithm, one can get optimal cache-obliviousness for free. The algorithms with this recursive structure are known as ***(a, b, c)-regular algorithms***. If an algorithm is (a, b, c)-regular, its I/O-complexity satisfies a recurrence of the form $T(N) = aT(N/b) + \Theta(1 + N^c/B)$,

[1] Several variations on this definition have also been used [5, 6] when considering particular problems (e.g., matrix multiplication). For (a, b, c)-regular algorithms, which are the focus of this paper, the used definitions are equivalent (and thus, as a convention, we use the most general definition).

where B is the block size of the cache and $\Theta(1 + N^c/B)$ represents the cost of scanning an array of size N^c.

For the purposes of cache-adaptivity, the only interesting cases are when $a > b$ and $c \leq 1$.[2] When $a > b$, an algorithm's performance is sensitive to the size of the cache, and so adaptivity becomes important.

If cache-oblivious algorithms were always cache-adaptive, then we could view adaptivity as a solved problem. Unfortunately, this is not the case. Bender et al. showed that, for $a > b$, (a, b, c)-regular algorithms are adaptive if and only if $c < 1$.

The worst-case gap between obliviousness and adaptivity. When $c = 1$, there can be a logarithmic gap between an algorithm's performance in the ideal cache and cache-adaptive models.[3]

Although many classical cache-oblivious algorithms are (a, b, c)-regular [17, 18, 28, 30, 49, 51, 56], many notable algorithms, including cache-oblivious dynamic programming algorithms [17], naive matrix multiplication [28], sub-cubic matrix multiplications (e.g., Strassen's algorithm [55]), and Gaussian elimination [17], have $a > b$ an $c = 1$ and hence fall into this gap. These algorithms are kernels of many other algorithms, including algorithms for solving linear equations in undirected and directed Laplacian matrices (see e.g., [26, 37, 52]), APSP [53, 54, 66], triangle counting [9], min-weight cycle [59], negative triangle detection and counting [59], and the replacement paths problem [59]. Lincoln, et al. [40] show that some algorithms can be rewritten to reduce c, making them adaptive, but the transformation is complex, introduces overhead, and doesn't work for all algorithms.

The goal of this paper is to show that this gap closes under less stringent notions of optimality.

Beyond the worst-case gap. Previous analysis shows that in the worst case there is a gap between the cache-adaptive and ideal-cache/cache-oblivious models. However, the logarithmic gap may just be an artifact of an unrealistically powerful adversary. The proof depends on exhibiting worst-case memory profiles that force the algorithm to perform poorly. The worst-case profiles mimic the recursive structure of the algorithm and maintain a tight synchronization between the algorithm's execution and the fluctuations in memory size. A concrete example of the worst-case profile for matrix multiplication can be found in Section 3.

The natural question to ask is: what happens to these bad examples when they get tweaked in some way, so that they no longer track the program execution so precisely? Is this gap robust to the inherent randomness that occurs in real systems?

Results

Our main result shows that, given any probability distribution Σ over box-sizes, if each box has size chosen i.i.d. from the distribution Σ, (a, b, c)-regular algorithms achieve optimal performance in the cache-adaptive model, matching their performance in the ideal cache model.

Theorem 1. *Consider an (a, b, c)-regular algorithm $\mathcal{A}$, where $a > b$ are constants in $\mathbb{N}$ and $c = 1$. Let Σ be a probability distribution over box sizes, and consider a sequence of boxes $(\square_1, \square_2, \square_3, \ldots)$ drawn independently from the distribution Σ. If all boxes in Σ are sufficiently large in $\Omega(1)$, then $\mathcal{A}$ is cache-adaptive in expectation on the random sequences $(\square_1, \square_2, \ldots)$.*

Proving this requires a number of new combinatorial ideas, an overview of which appear in Section 4. The full version of the paper formally proves this positive result.

The proof begins by reinterpreting cache-adaptivity in expectation in terms of the expected stopping time of a certain natural martingale. We then show a relationship between the expected stopping time for a problem and the expected stopping times for the child subproblems. A key obstacle, however, is that the linear scans performed by the algorithm can cause the natural recurrence on stopping times to break. In particular, the recurrence is able to relate the time to complete subproblems (including scans) and the time to complete their parent problems (excluding scans); but is unable to consider the parent problems in their entirety (including scans). We fill in this gap by showing that the total effect of the scans at all levels of the recursion on the expected stopping time is at most a constant factor. By analyzing the aggregate effect of scans across all levels of the recursion, we get around the fact that certain scans at specific levels can have far more impact on the expected stopping time than others.

Robustness to weaker smoothings. We further show that drawing box-sizes indepenently from one-another is necessary in the sense that several weaker forms of smoothing fail to close the logarithmic gap between the ideal-cache and cache-adaptive models. We show that worst-case profiles are robust to all of the following perturbations: randomly tweaking the size of each box by a constant factor, randomly shifting the start time of the algorithm, and randomly (or even adversarialy) altering the recursive structure of the profile.

These smoothings substantially alter the overall structure of the profile, and eliminate any initial alignment between the program and the structure of the memory profile. Nonetheless, the smoothed profiles remain worst-case in expectation. That is, as long as some recursive structure is maintained within the profile, the algorithm is very likely to gradually synchronize its execution to the profile in deleterious ways. In this sense, even a small amount of global structure between the sizes of consecutive boxes is enough to cause the logarithmic gap.

Map. This paper is organized as follows. Section 2 gives the definitions and conventions used in the rest of the paper. Section 3 provides intuition for the fragility of worst-case memory profiles. Section 4 explains the intuition for the proofs of the main theorems and sketch the combinatorial ideas behind the proofs. The full proofs can be found in the full version of this paper. Section 5 gives an in-depth examination of previous work, and Section 6 gives concluding remarks.

[2]When $a < b$ and $c = 1$, the algorithm runs in linear time independent of the cache size, and hence is trivially cache-adaptive. We are not aware of any (a, b, c)-regular cache-oblivious algorithms with $c > 1$.

[3](a, b, c)-regular algorithms are cache-adaptive when $a < b$ or $c < 1$. When $a = b$ and $c = 1$, no algorithm can be optimally cache-adaptive because such algorithms are already a $\Theta(\log \frac{M}{B})$ factor away from optimal in the DAM model [22]. This is why two-way merge sort, classic (i.e., not cache-oblivious) FFT, etc. cannot be optimal DAM algorithms.

2 PRELIMINARIES

The cache-adaptive model. The ***cache-adaptive*** (CA) model [5, 6] extends the classical disk access model (DAM) [1] to allow for the cache to change in size in each time step. In the DAM, the machine has a two-level memory hierarchy consisting of a fast ***cache*** (sometimes also referred to as ***memory*** or ***RAM***) of size M words and a slow ***disk***. Data is transferred between cache and disk in blocks of size B. An algorithm's running time is precisely the number of block transfers that it makes. Similarly, each I/O is a unit of time in the CA model.

In the cache-adaptive model, the size of fast memory is a (non-constant) function $m(t)$ giving the size of memory (in blocks) after the tth I/O. We 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. Although the cache-adaptive model allows the size of cache to change arbitrarily from one time-step to the next, prior work showed that we need only consider ***square profiles*** [5, 6]. Throughout this paper, we use the terms ***box*** and ***square*** interchangeably.

Definition 1 (Square Profile [5]). *A memory profile M or m is a **square profile** if there exist boundaries $0 = t_0 < t_1 < \dots$ such that for all $t \in [t_i, t_{i+1})$, $m(t) = t_{i+1} - t_i$. In other words, a square memory profile is a step function where each step is exactly as long as it is tall. We will use the notation $(\square_1, \square_2, \dots)$ to refer to a profile in which the i-th square is size $|\square_i|$.*

For convenience, we assume that cache is cleared at the start of each square. The paging results underlying cache-adaptivity [6] explain that this assumption is w.l.o.g. With this assumption, an algorithm gets to reference exactly X distinct blocks in a square of size X. Since any memory profile can be approximated with a square profile up to constant factors [5], any random distribution of generically produced profiles has a corresponding random distribution over square profiles that approximates it.

Recursive algorithms with (a, b, c)-regular structure. This paper focuses on algorithms with a common recursive structure.

Definition 2. *Let $a, b \in \mathbb{N}$ be constants, $b > 1$ and $c \in [0, 1]$. An **(a, b, c)-regular algorithm** is a recursive algorithm that, when run on a problem of size n blocks (equivalently $N = nB$ words), satisfies the following:*

- *On a problem of size n blocks, the algorithm accesses $\Theta(n)$ distinct blocks.*
- *Until the base case (when $n \in \Theta(1)$), each problem of size b blocks recurses on exactly a subproblems of size n/b.*
- *Within any non-base-case subproblem, the only computation besides the recursion is a **linear scan of size N^c/B**. This linear scan is any sequence of N^c contiguous memory accesses satisfying the property that a cache of a sufficiently large constant size can complete the sequence of accesses in time $O(N^c/B)$. Parts of the scan may be performed before, between, and after recursive calls.*

Remark 1. *When referring to the size of a subproblem, box, scan, etc., we use blocks, rather than machine words, as the default unit. We define (a, b, c)-regular algorithms to have a base case of size $O(1)$ blocks. This differs slightly from previous definitions [5, 6] which recurse down to $O(1)$ words.*

Remark 2. *The definition of linear scans ensures the following useful property. Consider a linear scan of size N^c/B. Consider any sequence of squares $(\square_1, \square_2, \dots, \square_j)$, where each $|\square_i|$ is a sufficiently large constant, and where $\sum_{i=1}^{j} |\square_i| = \Omega(N^c/B)$, for a sufficiently large constant in the Ω. Then the sequence of squares can be used to complete the scan in its entirety.*

The following theorem gives a particularly simple rule for when an (a, b, c)-regular algorithm is optimal.

Theorem 2 ((a, b, c)-regular optimality [5], informal). *Suppose $\mathcal{A}$ is an (a, b, c)-regular algorithm that is optimal in the DAM model. Then $\mathcal{A}$ is optimal in the cache-adaptive model if $c < 1$ or if $b > a$ and $a \geq 1$. If $c = 1$ and $a \geq b$, then $\mathcal{A}$ is $O(\log_b N)$ away from optimal on an input of size N in the cache-adaptive model. Optimality is defined as in [6], or equivalently as given by the notion of* efficiently cache adaptive, *defined below.*

Paper goal: closing the logarithmic gap. The above theorem uses a very structured memory profile in the case of $c = 1$ and $a \geq b$ to tease out the worst possible performance of $(a, b, 1)$-regular algorithms. We explore the smoothing of these profiles when $a > b$ in this paper. We leave the case of $a = b$ for future work because we prioritize the broader class of algorithms described by $a > b$.

Progress bounds in the cache-adaptive model. When an (a, b, c)-regular algorithm is run on a square profile, the ***progress*** of a box is the number of base-case subproblems performed (at least partly) within the box. Define the ***potential $\rho(|\square|)$*** of a box of size $|\square|$ to be the maximum possible progress that a size $|\square|$ box could ever make starting at any memory access of any execution of $\mathcal{A}$ on a problem of arbitrary size.

Lemma 1. *The potential of a box $\square$ for an (a, b, c)-regular algorithm $\mathcal{A}$ with $a > b$ and $c = 1$ is $\rho(|\square|) = \Theta(|\square|^{\log_b a})$.*

Proof. A square $\square$ can complete any subproblem A whose size in blocks is sufficiently small in $\Theta(|\square|)$. This allows $\square$ to complete $\Omega(a^{\log_b |\square|}) = \Omega(|\square|^{\log_b a})$ base-case subproblems, which proves $\rho(|\square|) \geq \Omega(|\square|^{\log_b a})$.

On the other hand, a square $\square$ is unable to complete in full any subproblem A whose size in blocks is sufficiently large in $\Theta(|\square|)$. It follows that $\square$ can complete base cases from at most two such subproblems A (the one that $\mathcal{A}$ begins $\square$ in and the one that $\mathcal{A}$ ends $\square$ in). This limits the number of base cases completed to $\rho(|\square|) \leq O(|\square|^{\log_b a})$. $\square$

Intuitively, the potential of a box $\square$ is essentially the same as the number of base-case recursive leaves in a problem of size $|\square|$.

Optimality in the cache-adaptive model. The progress of each square is upper bounded by its potential. An execution of the algorithm $\mathcal{A}$ on a problem of size n blocks and on a square profile M is ***efficiently cache-adaptive*** if the sequence of squares $(\square_1, \square_2, \dots, \square_j)$ given to the algorithm (with the final square rounded down in size to be only the portion of the square actually used) satisfies

$$\sum_{i=1}^{j} \rho(|\square_i|) \leq O(n^{\log_b a}). \tag{1}$$

The right-hand side of the expression represents the total amount of progress that must be made by any (a, b, c)-regular algorithm on a problem of size n in order to complete. In summary, an execution is efficiently cache-adaptive on the profile if every square in the profile makes progress asymptotically equal to its maximum possible potential.

An algorithm $\mathcal{A}$ (rather than just a single execution) is ***efficiently cache-adaptive*** (or ***cache-adaptive*** for short) if every execution of $\mathcal{A}$ is efficiently cache-adaptive on every infinite square-profile consisting of squares that are all of sizes sufficiently large in $O(1)$.

By Lemma 1, Inequality 1 can be written as $\sum_{i=1}^{j} |\square_i|^{\log_b a} \leq O(n^{\log_b a})$. Since all squares $\square_i$ completed by the algorithm are of size $O(n)$, an equivalent requirement is

$$\sum_{i=1}^{j} \min(n, |\square_i|)^{\log_b a} \leq O(n^{\log_b a}). \tag{2}$$

This requirement has the advantage that the size of the final square $\square_j$ cannot affect the veracity of the condition. Consequently, when using this version of the condition, we may feel free to *not* round down the size of the final square $\square_j$ in the profile M.

The definition of efficiently cache-adaptive is easily adapted to use an arbitrary progress function and an arbitrary algorithm $\mathcal{A}$ that need not be (a, b, c)-regular.[4]

Cache-adaptivity over distributions of profiles. We now define what it means for an algorithm to be cache-adaptive in expectation over a distribution of memory profiles. This allows us to perform smoothed and average-case analyses in subsequent sections.

Definition 3 (Efficiently cache-adaptive in expectation). *Let $\mathcal{M}$ be a distribution over (infinite) square memory profiles. Let M be a square-profile $(\square_1, \square_2, \ldots)$ drawn from the distribution $\mathcal{M}$, and define $\mathcal{S}_n$ to be the number of squares in the profile required by an (a, b, c)-regular $\mathcal{A}$ to complete on any problem of size n. We say that $\mathcal{A}$ is **(efficiently) cache-adaptive in expectation on $\mathcal{M}$** if for all problem sizes n,*

$$\mathbb{E}\left[\sum_{i=1}^{\mathcal{S}_n} \min(n, |\square_i|)^{\log_b a}\right] = O(n^{\log_b a}).$$

The bulk of this paper is devoted to investigating which memory-profile distributions cause $(a, b, 1)$-regular algorithms to be cache-adaptive in expectation.

[4]There is an alternative progress function based on operations. Consider the progress function in which each square makes progress equal to the number of memory accesses it completes. This generalizes our definition to non(a, b, c)-regular algorithms and, for many natural (a, b, c)-regular algorithms, this yields the same definition of cache-adaptivity as the above progress definition. However, the memory-access-based definition of progress can differ from our definition if some large scans are very non-homogeneous, i.e. if they contain portions in which a single small box can complete a large number of memory accesses. We use the sub-problem-based definition so that our results can apply to all (a, b, c)-regular algorithms.

A useful lemma. We conclude the section by presenting a useful lemma, known as the ***No-Catch-up Lemma***, that is implicitly present in [6]. The lemma will be used as a primitive throughout the paper, and for completeness, a full proof is given in the full version of the paper. Intuitively, the No-Catch-up Lemma states that delaying the start time of an algorithm can never help it finish earlier than it would have without the start time delay.

Lemma 2. *Let $\sigma = (r_1, r_2, r_3, \ldots)$ be a sequence of memory references, and let $S = (\square_1, \square_2, \ldots \square_k)$ be a sequence of squares. Suppose that if $\square_1$ starts at r_i, then $\square_k$ finishes at r_j. Then, for all $i' < i$, if $\square_1$ starts at $r_{i'}$, then for some $j' \leq j$, $\square_k$ finishes at $r_{j'}$.*

3 WHAT BAD MEMORY PROFILES LOOK LIKE

We begin by explaining how an (a, b, c)-regular algorithm can fail to be adaptive in the worst case, and why there is reason to hope that the worst cases are brittle.

*MM-Scan: a canonical non-adaptive algorithm. Consider a divide-and-conquer matrix-multiplication algorithm MM-Scan that computes eight subresults and then merges them together using a linear scan.MM-Scan is an $(8, 4, 1)$-regular cache-oblivious algorithm and its recurrence relation is $T(N) = 8T(N/4) + \Theta(N/B)$. Its I/O complexity is $O(N^{3/2}/\sqrt{M}B)$, which is optimal for an algorithm that performs all the elementary multiplications of a naïve nested-loop matrix multiply [28, 29].

However, since MM-Scan has $c = 1$ in its recurrence, it is not adaptive: there are bad memory profiles that cause it to run slowly despite giving MM-Scan ample aggregate resources[5]. This section gives intuition for what these bad profiles look like.

A worst-case profile for MM-Scan. Here's how to make a bad profile for MM-Scan [5]. The intuition is to give the algorithm lots of memory when it cannot benefit from it, i.e., when it is doing scans, and give it a paucity of memory when it could most use it, i.e., during subproblems.

Concretely, during a scan of size N, which takes N/B I/Os, set the memory to the fixed size N/B. Repeat recursively. Thus, a bad profile for MM-Scan on a problem of size N consists of eight recursive bad profiles for $N/4$ followed by a "square" of size N/B I/Os by N/B blocks of cache; see Figure 1.[6] This recursion continues down to squares of size $\Theta(B)$ blocks[7]. MM-Scan's I/O cost with this worst-case profile is exactly the same as if the memory stayed constant at its smallest possible value. MM-Scan can perform exactly one multiply of $\Theta(\sqrt{N} \times \sqrt{N})$ matrices on this profile. MM-InPlace, on the other hand, can perform $\Omega(\log \frac{N}{B})$ multiplies on this profile [5]. This proves that MM-Scan is not optimal in the cache-adaptive model.

[5]There is an alternate form of the algorithm, MM-InPlace, that immediately adds the results of elementary multiplications into the output matrix as they are computed. Since it needs no linear scan to merge results from sub-problems, it is an (8, 4, 0)-regular algorithm. Consequently, its I/O complexity in the DAM model is also $O(N^{3/2}/\sqrt{M}B)$, but it *is* optimally cache-adaptive.

[6]In the cache-adaptive model, it's enough to analyze cache-oblivious algorithms only on ***square profiles***, defined as follows [5]. Whenever the RAM size changes to have the capacity for x blocks, it stays constant x I/Os before it can change again. This paper focuses exclusively on cache-oblivious algorithms, so we use square profiles throughout.

[7]We stop at $\Theta(B)$ blocks due to the ***tall-cache requirement*** of MM-Scan [28]

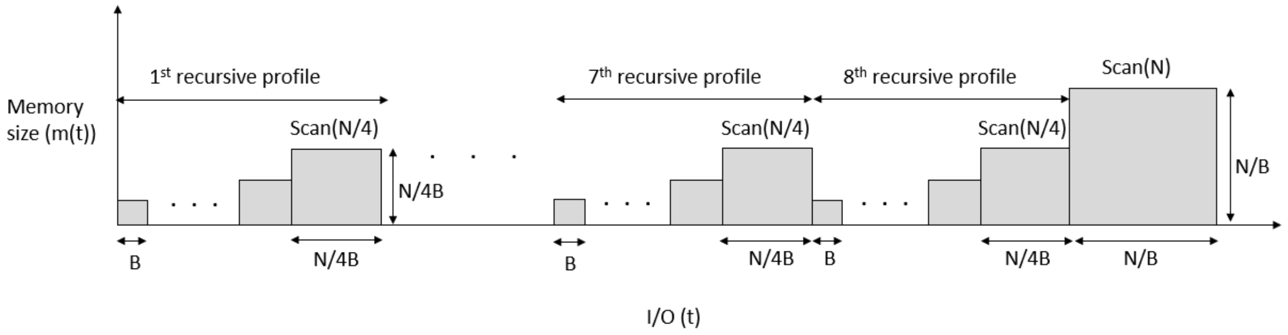

Figure 1: A bad profile for MM-SCAN as defined recursively.

This worst-case profile exactly tracks the execution of MM-Scan. From the perspective of the algorithm, the memory profile *does the wrong thing at every time step*; whenever MM-Scan cannot use more memory, it gets the maximum amount possible, and whenever it can use more memory, that memory gets taken away. This bad example for matrix multiplication generalizes to any $(a, b, 1)$-regular algorithm. When $c < 1$, this construction is ineffective—the scans are simply too small to waste a significant amount of resources.

The MM-Scan example reveals a fascinating combinatorial aspect of divide-and-conquer algorithms. At some points of the execution, the I/O performance is sensitive to the size of memory and sometimes it is almost entirely insensitive. These changes in memory sensitivity make cache-adaptive analysis nontrivial.

4 TECHNICAL OVERVIEW

Cache-Adaptivity on Randomly Shuffled Profiles

The main technical result of the paper is that random shuffling of adversarially constructed box-profiles makes (a, b, c)-regular algorithms where $a > b$ and $c = 1$ cache-adaptive in expectation. In the full version of the paper we prove the following:

Theorem 3 (Informal)**.** *Consider an (a, b, c)-regular algorithm $\mathcal{A}$, where $a > b$ $(b > 1)$ are constants in $\mathbb{N}$ and $c = 1$. Let Σ be a probability distribution over box sizes, and consider a sequence of boxes $(\square_1, \square_2, \square_3, \ldots)$ with sizes drawn independently from the distribution. Then $\mathcal{A}$ is cache-adaptive (in expectation) on the random sequence of boxes $(\square_1, \square_2, \square_3, \ldots)$.*

For simplicity, we discuss here the case where the block size $B = 1$, and $\mathcal{A}$ has the same values of a, b, c as the not-in-place naïve matrix-multiplication algorithm,with $a = 8$ and $b = 4$ and $c = 1$. Moreover, we assume that all box sizes and problem sizes are powers of 4. Doing so ensures $|a - b| \geq \Omega(1)$, allows us to simplify many of the expressions in intermediate calculations, and frees us from tracking factors of B and its added complexity[8]. Additionally, we consider a simplified model of caching: any box of size s that begins in a subproblem of size s or smaller completes to the end of the problem of size s containing it (and goes no further); and any box of size s that begins in the scan of a problem of size greater than s continues either for the rest of the scan or for s accesses in the scan, whichever is shorter. As a final simplification, we assume that each scan in each problem of some size s consists of exactly s memory accesses. In fact, we show in the full version of the paper that these simplifications may be made without loss of generality for arbitrary (a, b, c)-regular algorithms.

[8]At a high level, the cache-line analysis works exactly as one would expect for a nice, recursive algorithm. However, actually getting the probabilistic analysis correct adds some complication and is resolved through several of the simplification steps in the full version of the paper. As intuition (a, b, c)-regular algorithms will recurse down to sizes small enough to fit inside cache lines getting the requisite cache locality.

Let $\square$ denote a single box drawn from the distribution Σ. The proof of Theorem 3 begins by applying the Martingale Optional Stopping Theorem to combinatorially reinterpret what it means for $\mathcal{A}$ to be cache-adaptive in expectation on the random sequence $(\square_1, \square_2, \ldots)$. In particular, if $f(n)$ is the expected number of boxes needed for $\mathcal{A}$ to complete a problem of size n, then cache-adaptivity-in-expectation reduces to:

$$f(n) \leq \frac{O(8^{\log_4 n})}{m_n} = \frac{O(n^{\log_4 8})}{m_n} = \frac{O(n^{3/2})}{m_n}, \tag{3}$$

where $m_n = \mathbb{E}\left[\min(n, |\square|)^{3/2}\right]$ is the ***average n-bounded potential*** of a box.

At the heart of the proof of Theorem 3 is a somewhat unintuitive combinatorial argument for expressing $f(n)$, the expected number of boxes needed to complete a problem of size n, in terms of $f(n/4)$.

Lemma 3 (Stated for the simplified assumptions)**.** *Define $p = \Pr[|\square| \geq n] \cdot f(n/4)$. Then, the expected number of squares to complete the subproblems in a problem of size n is exactly $\sum_{i=1}^{8}(1-p)^{i-1} f(n/4)$, and the expected number of additional squares needed to complete the final scan is $(1 - \Theta(p)) \cdot \frac{\Theta(n)}{\mathbb{E}[\min(|\square|, n)]}$.*

Proof Sketch. When executing a problem of size n, the first subproblem requires $f(n/4)$ boxes to complete, on average. Define q to be the probability the boxes used to complete the first subproblem include a box of size n or larger. Then with probability q, no additional boxes are required[9] to complete the rest of the problem of size n. We will show that $q = p$ later in this proof. Otherwise, an average of $f(n/4)$ additional boxes are needed to complete the next subproblem of size $n/4$. Again, with probability q, one of these boxes completes the rest of the problem in its entirety. Similarly,

[9]This is due to the aforementioned simplified caching model.

the probability that the i-th subproblem is completed by a large box from a previous subproblem is $(1-q)^{i-1}$. Thus the expected number of boxes needed to complete all 8 subproblems is

$$\sum_{i=1}^{8}(1-q)^{i-1}f(n/4). \tag{4}$$

Remarkably, the probability q can also be expressed in terms of $f(n/4)$. Define S to be the random variable for the number of boxes used to complete the first subproblem of size $n/4$; define $\ell \leq O(n^{3/2})$ to be an upper bound for S; and define X to be the random variable for the number of the boxes in the subsequence $\square_1, \ldots, \square_S$ that are of size n or greater. The expectation of X can be expressed as

$$\mathbb{E}[X] = \sum_{i=1}^{\ell}\Pr[S \geq i] \cdot \Pr[|\square_i| \geq n \mid S \geq i].$$

Since $|\square_i|$ is independent of $|\square_1|, \ldots, |\square_{i-1}|$, we have that $\Pr[|\square_i| \geq n \mid S \geq i] = \Pr[|\square_i| \geq n]$. Thus

$$\begin{aligned}\mathbb{E}[X] &= \Pr[|\square| \geq n] \cdot \sum_{i=1}^{\ell}\Pr[S \geq i]\\ &= \Pr[|\square| \geq n] \cdot \mathbb{E}[S] = \Pr[|\square| \geq n] \cdot f(n/4).\end{aligned}$$

Notice, however, that at most one of the boxes $\square_1, \ldots, \square_S$ can have size n or larger (since such a box will immediately complete the subproblem). Thus X is an indicator variable, meaning that $q = \Pr[X \geq 1] = \mathbb{E}[X] = \Pr[|\square| \geq n] \cdot f(n/4) = p$. So $q = p$, as promised. Expanding Equation 4, we get that the expected number of boxes needed to complete the 8 subproblems is, as desired, at most

$$\sum_{i=1}^{8}\Big(1 - \Pr[|\square| \geq n] \cdot f(n/4)\Big)^{i-1} f(n/4) \tag{5}$$

Next we consider the boxes needed to complete the final scan. Suppose the scan were to be run on its own. Let K denote the number of boxes needed to complete it, and let $\square'_1, \ldots, \square'_K$ denote those squares.

Rather than consider $\mathbb{E}[K]$ directly, we instead consider $\mathbb{E}[K] \cdot \mathbb{E}[\min(|\square|, n)]$. Through a combinatorial reinterpretation, we have

$$\begin{aligned}E[K] \cdot \mathbb{E}[\min(|\square|, n)] &= \mathbb{E}[\min(|\square|, n)] \cdot \sum_{i=1}^{\ell}\Pr[K \geq i]\\ &= \sum_{i=1}^{\ell}\Pr[K \geq i] \cdot \mathbb{E}[\min(|\square'_i|, n) \mid K \geq i]\\ &= \mathbb{E}\left[\sum_{i=1}^{K}\min(|\square'_i|, n)\right].\end{aligned}$$

The quantity in the final expectation has the remarkable property that it is *deterministically* between n and $2n-1$. Thus the same can be said for its expectation, implying that $\mathbb{E}[K] \cdot \mathbb{E}[\min(|\square|, n)] = \Theta(n)$.

Recall that K is the expected number of boxes to complete the scan on its own. In our problem, the scan is at the end of a problem, and thus with probability $1-(1-p)^8 = \Theta(p)$, the scan is completed by a large box from one of the preceding subproblems. Hence the expected number of additional boxes to complete the scan is $(1-\Theta(p)) \cdot \frac{\Theta(n)}{\mathbb{E}[\min(|\square|, n)]}$.

□

Lemma 3 suggests a natural inductive approach to proving Theorem 3. Rather than explicitly showing that $f(n) \leq \frac{O(n^{3/2})}{m_n}$, one could instead prove the result by induction, arguing for each n that

$$\frac{f(n)}{f(n/4)} \leq \frac{n^{3/2}/m_n}{(n/4)^{3/2}/m_{n/4}} = 8 \cdot \frac{m_{n/4}}{m_n}. \tag{6}$$

One can construct example box-size distributions Σ showing that the Equation 6 does not always hold, however. In particular, the scan at the end of a subproblem of size n can make $f(n)$ sufficiently larger than $f(n/4)$ that Equation 6 is violated. To get around this problem, one could attempt to instead prove that

$$\frac{f'(n)}{f(n/4)} \leq 8 \cdot \frac{m_{n/4}}{m_n}, \tag{7}$$

where $f'(n)$ is the expected number of boxes needed to complete a problem of size n, without performing the final scan at the end. Unlike Equation 6, Equation 7 does not inductively imply a bound of the form $f(n) \leq \frac{O(n^{3/2})}{m_n}$, which is necessary for cache-adaptivity in expectation. If one additionally proves that

$$\prod_{4^k \leq n}\frac{f(4^k)}{f'(4^k)} \leq O(1), \tag{8}$$

then Equation 8 could be used to "fill in the holes in the induction" in order to complete a proof of cache-adaptivity. Equation 8 is somewhat unintuitive in the sense that individual terms in the product can actually be a positive constant greater than 1. The inequality states that, even though the scans in an individual subproblem size could have a significant impact on $f(n)$, the aggregate effect over all sizes is no more than constant.

To make this semi-inductive proof structure feasible, one additional insight is necessary. Rather than proving Equation 7 for all values n, one can instead restrict oneself only to values n such that

$$f(n) \geq C \cdot \frac{n^{3/2}}{m_n}, \tag{9}$$

where C is an arbitrarily large constant of our choice. In particular, if n_0 is the largest power of 4 less than our problem-size such that $f(n_0) < C \cdot \frac{n^{3/2}}{m_n}$, then we can use cache-adaptivity within problems of size n_0 as a base case, and then prove Equation 7 only for problem-sizes between n_0 and n. Similarly, we can restrict the product in Equation 8 to ignore problem sizes of size smaller than n_0.

When Equation 9 holds, Equation 7 can be interpreted as a negative feedback loop, saying that as we look at problem sizes $n = 1, 4, 16, \ldots$, whenever $f(n)$ becomes large enough to be on the cusp of violating cache-adaptivity, there exists downward pressure (in the form of Equation 7) that prevents it from continuing to grow in an unmitigated fashion.

The full proof of Theorem 3 takes the structure outlined above. At its core are the combinatorial arguments used in Lemma 3, which allow us to recast $f(n)$ and $f'(n)$ in terms of $f(n/4)$ and $f'(n/4)$. When applied in the correct manner, these arguments can be used to show Equation 7 (assuming Equation 9) with only a few additional ideas. The proof of Equation 8 ends up being somewhat more sophisticated, using combinatorial ideas from Lemma 3 in order to expand each of the terms $f(4^k)/f'(4^k)$, and then relying on a series of subtle cancellation arguments in order to bound the end product by a constant.

Robustness of Worst-Case Profiles

We consider three natural forms of smoothing on worst-case profiles. Remarkably, the worst-case nature of the profiles persists in all three cases. The canonical worst-case profile is highly structured, giving the algorithm a larger cache precisely when the algorithm does not need it. It is tempting to conclude that bad profiles must remain closely synchronized with the progression of the algorithm. By exploiting self-symmetry within worst-case profiles as well as the power of the No-Catchup Lemma, our results establish that this is not the case. The No-Catchup Lemma, in particular, allows us to capture the idea of an algorithm resynchronizing with a profile, even after the profile has been perturbed.

We begin by defining a canonical (a, b, c)-regular algorithm $\mathcal{A}_n$ on problems of size n, and a corresponding worst-case profile $M_{a,b}$. The profile $M_{a,b}$ completes each scan of size k in $\mathcal{A}$ in a single box of size k, thereby ensuring that every box makes its minimum possible progress. The profile $M_{a,b}$ is the ***limit profile*** of the sequence of profiles $M_{a,b}(n)$ for $n = 1, b, b^2, \ldots$, constructed recursively by defining $M_{a,b}(n)$ by concatenating together a copies of $M_{a,b}(n/b)$ and then placing a box of size n at the end. The algorithm $\mathcal{A}_n$ requires the entirety of the profile $M_{a,b}(n)$ to complete. One can check inductively that $M_{a,b}(n)$ has total potential $n^{\log_b a} \cdot \log n$, thereby making $M_{a,b}$ a worst-case profile.

Box-size perturbations. Consider any probability distribution $\mathcal{P}$ over $[0, t]$ for $t \leq \sqrt{n}$ such that for X drawn at random from $\mathcal{P}$, $\mathbb{E}[X] = \Theta(t)$. Let $X_1, X_2, \ldots$ be drawn iid from $\mathcal{P}$ and define $\mathcal{M}$ to be the distribution over square profiles obtained by replacing each box $\square_i$ in $\mathcal{M}$ with a box of size $|\square_i| \cdot X_i$. In the full version of the paper we show that the highly perturbed square profiles in $\mathcal{M}$ still remain worst-case in expectation.

The proof takes two parts. We begin by defining T to be the smallest power of b greater than t, and considering the profile $T \cdot M_{a,b}$ obtained by multiplying each box's size by T. Exploiting self-symmetry in the definition of $M_{a,b}$, we are able to reinterpret $T \cdot M_{a,b}$ as the profile $M_{a,b}$ in which all boxes of size smaller than T have been *removed*. Recall that $M_{a,b}(n)$ denotes the prefix of $M_{a,b}$ on which $\mathcal{A}_n$ completes. Using the fact that $T \leq \sqrt{n}$, we prove that the boxes of size smaller than T contain at most a constant fraction of the potential in the prefix $M_{a,b}(n)$. On the other hand, by iterative applications of the No-Catchup Lemma, the removal of the boxes cannot facilitate $\mathcal{A}$ to finish earlier in the profile. Combining these facts, we establish that $T \cdot M_{a,b}$ remains worst-case.

To obtain an element of $\mathcal{M}$ from $T \cdot M_{a,b}$, one simply multiplies the size of each box $\square_i$ in $T \cdot M_{a,b}$ by X_i/T, where T is drawn from the distribution $\mathcal{P}$. Using that $\mathbb{E}[X_i/T] = \Theta(1)$ and that $n^{\log_b a}$ is a convex function, Jensen's inequality tells us that the expected potential of the new box of size $\frac{|\square_i| \cdot X_i}{T}$ is at least a constant fraction of the original potential. Since the perturbations preserve the expected potentials of the boxes in $T \cdot M_{a,b}$ up to a constant factor, we can prove that the resulting profile is worst-case in expectation by demonstrating that the perturbations do not result in $\mathcal{A}_n$ finishing earlier in $T \cdot M_{a,b}$ then it would have otherwise. Since each perturbation can only reduce the size of a box in $T \cdot M_{a,b}$, this can be shown by iterative applications of the No-Catchup Lemma.

Start time perturbations. We consider what happens if the memory profile $M_{a,b}(n)$ is cyclic-shifted by a random amount. This corresponds to executing $\mathcal{A}_{a,b}(n)$ starting at a random start-time in the cyclic version of $M_{a,b}(n)$. Again, the resulting distribution of profiles remains worst-case in expectation.

The key insight in the proof is that $M_{a,b}(n)$ can be expressed as the concatenation of two profiles $A = (\square_1, \ldots, \square_x)$ and $B = (\square'_1, \ldots, \square'_y)$ such that

$$\sum_{i=1}^{x} |\square_i| \geq \Omega\left(\sum_{i=1}^{y} |\square'_i|\right), \tag{10}$$

$$\sum_{i=1}^{x} |\square_i|^{\log_b a} \leq O\left(\sum_{i=1}^{y} |\square'_i|^{\log_b a}\right). \tag{11}$$

Equation 10 establishes that with at least constant probability, a random selected start-time in the profile $M_{a,b}(n)$ falls in the prefix A. By a slight variant on the No-Catchup Lemma, if $\mathcal{A}$ is executed starting at that random start-time, it is guaranteed to use all of the boxes in the suffix B. By Equation 11, however, these boxes contain a constant fraction of the potential from the original worst-case profile $M_{a,b}(n)$. Thus, with constant probability the algorithm $\mathcal{A}$ runs at a random start-time that results in a profile that is still worst-case. This ensures that the randomly shifted profile will be worst-case in expectation.

Box-order perturbations. The bad profile, $M_{a,b}$, is constructed recursively by making a copies of $M_{a,b}(n/b)$ followed by a box of size n. The box comes at the end, intuitively, because all $(a, b, 1)$-regular algorithms with upfront scans in each subproblem can converted to an equivalent $(a, b, 1)$-regular algorithm, where the scans in all subproblems are at the end, preceded by a single linear scan.

We consider a relaxation of the construction of $M_{a,b}$. When constructing $M_{a,b}(n)$ recursively, rather than always placing a box of size n after the *final* instance of $M_{a,b}(n/b)$, we instead allow ourselves to place the box of size n after *any* of the a recursive instances of $M_{a,b}(n/b)$ (each of which may no longer be identical to the others due to the non-determinedness of the new recursive construction).

Although at first glance moving the largest box in the profile seems to closely resemble the random shuffling considered in the full version of the paper, we prove that the resulting distribution over square profiles again remains worst-case in expectation. In fact, we can prove a stronger statement: for memory profile M drawn from the resulting distribution $\mathcal{M}$ of square profiles, M is a worst-case profile with probability *one*.

5 RELATED WORK

Modeling the performance of algorithms in the real-world is an active area of study and has broad implications both theoretically and for performance engineering. In order to apply our algorithms to real-world systems, it is important to find the right model in which the theoretical efficiency of our algorithms closely matches their practical efficiency.

The ***disk access model (DAM)*** was formulated [1, 33] to account for multi-level memory hierarchies (present in real systems) where the size of memory available for computation and the speed of computation differs in each level. The DAM [1] models a 2-level memory hierarchy with a large (infinite sized) but slow disk, and a small (bounded by M) but fast cache. The drawback of DAM is that efficient algorithms developed in this model require knowledge of cache size. The ***ideal-cache model*** [28, 51] was proposed to counteract this drawback by building an automatic paging algorithm into the model and providing *no knowledge* of the cache size to algorithms. Thus, ***cache-oblivious algorithms*** [21, 36] are independent of the memory parameter and can be applied to complex multi-level architectures where the size of each memory-level is unknown. There exists a plethora of previous work on the performance analysis and implementations of cache-oblivious algorithms (on single-core and multi-core machines) [7, 8, 10, 12, 15, 16, 19, 27, 28, 38, 60, 61]. Among other limits [4, 11], one critical limit of the cache-oblivious model is that it does not account for *changing cache-size*. In fact, preliminary experimental studies have shown that two cache-oblivious algorithms (with the same I/O-complexity) might in fact perform vastly differently under a changing cache [40].

Changing cache size can stem from a variety of reasons. For example, shared caches in multi-core environment may allocate different portions of the cache to different processes at any time (and this allocation could be independent of the memory needed by each process). There has been substantial work on paging in shared-cache environments. For example, Peserico formulated alternative models for page replacement [50] provided a fluctuating cache. However, Peserico's page-replacement model differs from the cache-adaptive model because in his model, the cache-size changes at specific locations in the page-request sequence as opposed to being temporally related to each individual I/O. Other page replacement policies have applied to multicore shared-cache environments [35] where several processes share the same cache [2, 32, 34, 41, 42, 62] leading to situations where page sizes can vary [42] and where an application can adjust the cache size itself [34, 62].

Theoretical [2, 3] and empirical studies [47, 64, 65] have been done in the past to study partial aspects of adaptivity to memory fluctuations [13, 31, 44, 45, 48, 63, 65]. Barve and Vitter [2, 3] were the first to generalize the DAM model to account for changing cache size. In their model, they provide optimal algorithms for sorting, matrix multiplication, LU decomposition, FFT, and permutation but stops just short of a generalized technique for finding algorithms that are optimal under memory fluctuations [2, 3]. In their model, the cache is guaranteed to stay at size M for M/B I/Os. In this way, their model is very similar to our notion of square profiles.

The cache-adaptive model [6] introduced the notion of a ***memory profile***. The memory profile provides the cache size at each time step (defined as an I/O-operation), and at each time step the cache can increase by 1 block or decrease by an arbitrary amount. Bender et al. [5] went on to show that *any* optimal (in the DAM) (a, b, c)-regular algorithm where $a > b$ and $c < 1$ is ***cache-adaptive*** or *optimal* under this model. However, disappointingly, they showed that (a, b, c)-regular algorithms where $c = 1$ can be up to a log-factor away from optimal [5]. This leads to the the question of whether non-adaptive (a, b, c)-regular algorithms can be turned into cache-adaptive algorithms via some procedure. Lincoln et al. [40] took the first step in this direction by introducing a ***scan-hiding*** procedure for turning certain non-adaptive (a, b, c)-regular algorithms into cache-adaptive ones. Although scan-hiding takes polynomial time, it introduces too much overhead and also does not apply to all (a, b, c)-regular algorithms where $a > b$ and $c = 1$.

This paper takes another important step in this direction by showing that (a, b, c)-regular algorithms where $a > b$ and $c = 1$ *are cache-adaptive in expectation*. Whereas previous papers analyzed all algorithms in the *worst-case*, we believe that this is, in fact, unnecessary and does not accurately depict real-world architectures. We introduce the notion of average case cache-adaptivity in what we hope to be a more accurate picture of shared-cache multi-core systems.

Acknowledgments

We gratefully acknowledge support from NSF grants: CCF-1439084, CCF-1533644, CCF-1725543, CSR-1763680, CCF-1716252, CCF-1617618, CNS-1938709, CNS-1553510.

This research is partially supported by an NSF GRFP fellowship and Fannie & John Hertz Foundation fellowship. This research is also supported by the United States Air Force Research Laboratory and was accomplished under Cooperative Agreement Number FA8750-19-2-1000.[10]

6 CONCLUSION

This paper presents the first beyond-worst-case analysis of (a, b, c)-regular cache-adaptive algorithms. The main positive result in this paper gives hope for cache-adaptivity: even though the worst-case profile from previous work [5, 6] is robust under random perturbations and shuffling, many (a, b, c)-regular algorithms become cache-adaptive in expectation on profiles generated from any distribution. Notably, to our knowledge, all currently known subcubic matrix multiplication algorithms (such as Strassen's [55], Vassilevska Williams' [58], Coppersmith-Winograd's [20], and Le Gall's [39]) were a logarithmic factor away from adaptive under worst-case analysis, but are adaptive in expectation on random profiles via smoothed analysis. Our results provide guidance for analyzing cache-adaptive algorithms on profiles beyond the adversarially constructed worst-case profile.

Cache fluctuations are a fact of life on modern hardware, but many open questions remain. In this paper, we randomized memory profiles for deterministic (a, b, c)-regular algorithms. Could randomized algorithms also overcome worst-case profiles and result in cache-adaptivity? On the empirical side, which patterns of memory fluctuations occur in the real world? Further exploration of beyond-worst-case analysis may help model practical memory patterns more accurately.

[10]The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

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

[2] Rakesh Barve and Jeffrey S. Vitter. 1998. *External memory algorithms with dynamically changing memory allocations.* Technical Report. Duke University.

[3] Rakesh D. Barve and Jeffrey Scott Vitter. 1999. A Theoretical Framework for Memory-Adaptive Algorithms. In *Proc. 40th Annual Symposium on the Foundations of Computer Science (FOCS).* 273–284.

[4] Michael A. Bender, Gerth Stølting Brodal, Rolf Fagerberg, Dongdong Ge, Simai He, Haodong Hu, John Iacono, and Alejandro López-Ortiz. 2011. The Cost of Cache-Oblivious Searching. *Algorithmica* 61, 2 (2011), 463–505.

[5] Michael A. Bender, Erik D. Demaine, Roozbeh Ebrahimi, Jeremy T. Fineman, Rob Johnson, Andrea Lincoln, Jayson Lynch, and Samuel McCauley. 2016. Cache-Adaptive Analysis. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).* 135–144. https://doi.org/10.1145/2935764.2935798

[6] Michael A. Bender, Roozbeh Ebrahimi, Jeremy T. Fineman, Golnaz Ghasemiesfeh, Rob Johnson, and Samuel McCauley. 2014. Cache-adaptive Algorithms. In *Proc. 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (Portland, Oregon). 958–971.

[7] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. 2007. Cache-oblivious Streaming B-trees. In *Proc. 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).* 81–92.

[8] Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. 2006. Cache-oblivious string B-trees. In *Proc. 25th Annual ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS).* 233–242.

[9] Andreas Björklund, Rasmus Pagh, Virginia Vassilevska Williams, and Uri Zwick. 2014. Listing triangles. In *International Colloquium on Automata, Languages, and Programming.* Springer, 223–234.

[10] Guy E Blelloch, Rezaul A Chowdhury, Phillip B Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. 2008. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proc. 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA).* Society for Industrial and Applied Mathematics, 501–510.

[11] Gerth Stølting Brodal and Rolf Fagerberg. 2003. On the limits of cache-obliviousness. In *Proc. 35th Annual ACM Symposium on Theory of Computing (STOC)* (San Diego, CA, USA). New York, NY, USA, 307–315.

[12] Gerth Stølting Brodal, Rolf Fagerberg, and Kristoffer Vinther. 2007. Engineering a cache-oblivious sorting algorithm. *ACM Journal of Experimental Algorithmics* 12 (2007).

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

[14] Jichuan Chang and Gurindar S Sohi. 2006. *Cooperative caching for chip multiprocessors.* Vol. 34.

[15] Rezaul Chowdhury, Muhibur Rasheed, Donald Keidel, Maysam Moussalem, Arthur Olson, Michel Sanner, and Chandrajit Bajaj. 2013. Protein-Protein Docking with F2Dock 2.0 and GB-Rerank. *PLoS ONE* 8, 3 (2013).

[16] Rezaul Alam Chowdhury, Hai-Son Le, and Vijaya Ramachandran. 2010. Cache-Oblivious Dynamic Programming for Bioinformatics. *IEEE/ACM Trans. Comput. Biology Bioinform.* 7, 3 (2010), 495–510.

[17] Rezaul Alam Chowdhury and Vijaya Ramachandran. 2006. Cache-oblivious dynamic programming. In *Proc. 17th annual ACM-SIAM Symposium on Discrete Algorithms (SODA).* 591–600.

[18] Rezaul Alam Chowdhury and Vijaya Ramachandran. 2010. The cache-oblivious gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. *Theory of Computing Systems* 47, 4 (2010), 878–919.

[19] R Cole and V Ramachandran. 2010. Efficient resource oblivious scheduling of multicore algorithms. manuscript.

[20] Don Coppersmith and Shmuel Winograd. 1990. Matrix multiplication via arithmetic progressions. *Journal of symbolic computation* 9, 3 (1990), 251–280.

[21] Erik D. Demaine. 2002. Cache-Oblivious Algorithms and Data Structures. (2002). Lecture Notes from the EEF Summer School on Massive Data Sets.

[22] Erik D. Demaine, Andrea Lincoln, Quanquan C. Liu, Jayson Lynch, and Virginia Vassilevska Williams. 2018. Fine-grained I/O Complexity via Reductions: New Lower Bounds, Faster Algorithms, and a Time Hierarchy. In *9th Innovations in Theoretical Computer Science Conference, ITCS 2018, January 11-14, 2018, Cambridge, MA, USA.* 34:1–34:23. https://doi.org/10.4230/LIPIcs.ITCS.2018.34

[23] Peter J Denning. 1968. Thrashing: Its causes and prevention. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I.* 915–922.

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

[25] Dave Dice, Virendra J. Marathe, and Nir Shavit. 2014. Brief Announcement: Persistent Unfairness Arising from Cache Residency Imbalance. In *26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14, Prague, Czech Republic - June 23 - 25, 2014.* 82–83. https://doi.org/10.1145/2612669.2612703

[26] David Durfee, John Peebles, Richard Peng, and Anup B. Rao. 2017. Determinant-Preserving Sparsification of SDDM Matrices with Applications to Counting and Sampling Spanning Trees. In *FOCS.* IEEE Computer Society, 926–937.

[27] Matteo Frigo and Steven G Johnson. 2005. The design and implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231.

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

[29] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 2012. Cache-Oblivious Algorithms. *ACM Transactions on Algorithms* 8, 1 (2012), 4.

[30] Matteo Frigo and Volker Strumpen. 2005. Cache-oblivious stencil computations. Citeseer.

[31] Goetz Graefe. 2013. A New Memory-Adaptive External Merge Sort. Private communication.

[32] Avinatan Hassidim. 2010. Cache Replacement Policies for Multicore Processors. In *Proc. 1st Annual Symposium on Innovations in Computer Science (ICS).* 501–509.

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

[34] Sandy Irani. 1997. Page Replacement with Multi-Size Pages and Applications to Web Caching. In *Proc. 29th Annual ACM Symposium on the Theory of Computing (STOC).* 701–710.

[35] Anil Kumar Katti and Vijaya Ramachandran. 2012. Competitive Cache Replacement Strategies for Shared Cache Environments. In *Proc. 26th International Parallel and Distributed Processing Symposium (IPDPS) (IPDPS '12).* 215–226.

[36] P. Kumar. 2003. Cache Oblivious Algorithms. (2003), 193–212. http://link.springer.de/link/service/series/0558/tocs/t2625.htm

[37] Rasmus Kyng and Sushant Sachdeva. 2016. Approximate Gaussian Elimination for Laplacians - Fast, Sparse, and Simple. In *FOCS.* IEEE Computer Society, 573–582.

[38] R.E. Ladner, R. Fortna, and B.-H. Nguyen. 2002. A Comparison of Cache Aware and Cache Oblivious Static Search Trees Using Program Instrumentation. *Experimental Algorithmics* (2002), 78–92.

[39] François Le Gall. 2014. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation.* 296–303.

[40] Andrea Lincoln, Quanquan C. Liu, Jayson Lynch, and Helen Xu. 2018. Cache-Adaptive Exploration: Experimental Results and Scan-Hiding for Adaptivity. In *Proc. 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA).* 213–222. https://doi.org/10.1145/3210377.3210382

[41] Alejandro López-Ortiz and Alejandro Salinger. 2012. Minimizing Cache Usage in Paging. In *Proc. 10th Workshop on Approximation and Online Algorithms (WAOA).*

[42] Alejandro López-Ortiz and Alejandro Salinger. 2012. Paging for multi-core shared caches. In *Proc. Innovations in Theoretical Computer Science (ITCS).* 113–127.

[43] Paul Menage. [n.d.]. CGROUPS. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

[44] Richard Tran Mills. 2004. *Dynamic adaptation to CPU and memory load in scientific applications.* Ph.D. Dissertation. The College of William and Mary.

[45] Richard T Mills, Andreas Stathopoulos, and Dimitrios S Nikolopoulos. 2004. Adapting to memory pressure from within scientific applications on multiprogrammed COWs. In *Proc. 8th International Parallel and Distributed Processing Symposium (IPDPS).* 71.

[46] Khang T Nguyen. [n.d.]. Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology

[47] HweeHwa Pang, Michael J. Carey, and Miron Livny. 1993. Memory-Adaptive External Sorting. In *Proc. 19th International Conference on Very Large Data Bases (VLDB).* Morgan Kaufmann, 618–629.

[48] HweeHwa Pang, Michael J Carey, and Miron Livny. 1993. Partially Preemptible Hash Joins. In *Proc. 5th ACM SIGMOD International Conference on Management of Data (COMAD).* 59.

[49] J-S Park, Michael Penner, and Viktor K Prasanna. 2004. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on Parallel and Distributed Systems* 15, 9 (2004), 769–782.

[50] Enoch Peserico. 2013. Paging with dynamic memory capacity. *CoRR* abs/1304.6007 (2013).

[51] H. Prokop. 1999. *Cache Oblivious Algorithms.* Master's thesis. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

[52] Aaron Schild, Satish Rao, and Nikhil Srivastava. 2018. Localization of Electrical Flows. In *SODA.* SIAM, 1577–1584.

[53] Raimund Seidel. 1995. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of computer and system sciences* 51, 3 (1995), 400–403.

[54] Avi Shoshan and Uri Zwick. 1999. All pairs shortest paths in undirected graphs with integer weights. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039).* 605–614.

[55] Volker Strassen. 1969. Gaussian elimination is not optimal. *Numerische mathematik* 13, 4 (1969), 354–356.
[56] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The pochoir stencil compiler. In *SPAA*. 117–128.
[57] Jue Wang, Xiangyu Dong, Yuan Xie, and Norman P Jouppi. 2014. Endurance-aware cache line management for non-volatile caches. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 1 (2014), 4.
[58] Virginia Vassilevska Williams. 2012. Multiplying matrices faster than Coppersmith-Winograd. In *In Proc. 44th ACM Symposium on Theory of Computation*. Citeseer.
[59] Virginia Vassilevska Williams and Ryan Williams. 2010. Subcubic equivalences between path, matrix and triangle problems. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*. 645–654.
[60] Sung-Eui Yoon, Peter Lindstrom, Valerio Pascucci, and Dinesh Manocha. 2005. Cache-oblivious mesh layouts. 24, 3 (2005), 886–893.
[61] Kamen Yotov, Tom Roeder, Keshav Pingali, John Gunnels, and Fred Gustavson. 2007. An experimental comparison of cache-oblivious and cache-conscious programs. In *Proc. 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 93–104.
[62] Neal E. Young. 2002. On-Line File Caching. *Algorithmica* 33, 3 (2002), 371–383.
[63] Hansjörg Zeller and Jim Gray. 1990. An adaptive hash join algorithm for multiuser environments. In *Proc. 16th International Conference on Very Large Data Bases (VLDB)*. 186–197.
[64] Weiye Zhang and Per-Äke Larson. 1996. A memory-adaptive sort (MASORT) for database systems. In *Proc. 6th International Conference of the Centre for Advanced Studies on Collaborative research (CASCON)* (Toronto, Ontario, Canada). IBM Press, 41–.
[65] Weiye Zhang and Per-Äke Larson. 1997. Dynamic Memory Adjustment for External Mergesort. In *Proc. 23rd International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann Publishers Inc., 376–385.
[66] Uri Zwick. 1998. All pairs shortest paths in weighted directed graphs-exact and almost exact algorithms. In *Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No. 98CB36280)*. 310–319.