Cache-Adaptive Exploration: Experimental Results and Scan-Hiding for Adaptivity

Andrea Lincoln* Quauquan C. Liu* Jayson Lynch* Helen Xu*

August 10, 2022

Abstract

Systems that require programs to share a cache such as shared-memory machines, multicore architectures, and time-sharing systems are ubiquitous in modern computing. Programs that share a cache experience different levels of cache space throughout execution. Moreover, practitioners have observed that the cache efficiency of an algorithm is often critical to its overall performance. If an algorithm is optimally cache efficient in a fixed-size cache, its performance improves when it has more cache space. Similarly, cache-adaptive algorithms use a dynamic cache optimally.

Despite the increasing popularity of shared-cache systems, the theoretical behavior of most algorithms in the face of memory fluctuations is not yet well understood. There is a gap between our knowledge about how algorithms perform in a fixed-size (static) cache versus a dynamic cache where the amount of memory available to a program fluctuates.

Cache-adaptive analysis is a method of analyzing how well algorithms use a dynamic cache. Bender *et al.* showed that optimal cache-adaptivity does not follow from cache-optimality in a static cache. Specifically, they proved that some cache-optimal algorithms in a static cache are suboptimal when subject to certain memory profiles (patterns of memory fluctuations). For example, the canonical cache-oblivious divide-and-conquer formulation of Strassen's algorithm for matrix multiplication is suboptimal in the cache-adaptive model because it does a linear scan to add submatrices together.

In this paper, we introduce "scan hiding", the first technique for converting a class of non-cache-adaptive algorithms with linear scans to optimally cache-adaptive variants. We work through a concrete example of scan hiding on Strassen's algorithm, a subcubic algorithm for matrix multiplication that involves linear scans at each level of its recursive structure. All currently known subcubic algorithms for matrix multiplication include linear scans, however, so our technique applies to a large class of algorithms.

We also experimentally evaluated different algorithms in the face of memory fluctuations to explore how theoretical analysis of cache-adaptivity manifests in practice. We experimentally compared two cubic algorithms for matrix multiplication which are both cache-optimal when the memory size stays the same, but diverge under cache-adaptive analysis. We also subjected three standard sorting algorithms to memory fluctuations. Our findings suggest that memory fluctuations affect algorithms with the same theoretical cache performance in a static cache differently. For example, the optimally cache-adaptive naive matrix multiplication algorithm achieved fewer relative faults than the non-adaptive variant in the face of changing memory size. Our experiments also suggest that the performance advantage of cache-adaptive algorithms extends to more practical situations beyond the carefully-crafted memory specifications in proofs of worst-case behavior.

*Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139, USA. Emails: {andreali, quanquan, jaysonl, hjxu}@mit.edu. This research was supported in part by NSF grants 1314547, 1740519, 1651838 and 1533644, a National Physical Sciences Consortium Fellowship, and National Science Foundation Graduate Research Fellowship grant 1122374.

1 Introduction

Since multiple processors may compete for space in a shared cache on modern machines, the amount of memory available to a single process may vary dynamically. Programs running on multicore architectures, shared-memory machines, and time-shared machines often experience memory fluctuations. For example, Dice, Marathe and Shavit [17] demonstrated that in practice, multiple threads running the same program will each get a different amount of access to a shared cache.

Experimentalists have recognized the problem of changing memory size in large-scale scientific computing and have developed heuristics [24, 25, 26] for allocation with empirical guarantees. Furthermore, researchers have developed adaptive sorting and join algorithms [12, 22, 29, 30, 37, 38, 39] that perform well empirically in environments with memory fluctuations. While most of these algorithms work well on average, they may still suffer from poor worst-case performance [6, 7].

In this paper, we continue the study of algorithmic design of external-memory algorithms which perform well in spite of memory fluctuations. Barve and Vitter [6, 7] initiated the theoretical analysis of such algorithms. Bender *et al.* [10] later formulated the "cache-adaptive model" to formally study the effect of such memory changes and analyzed specific "cache-adaptive" algorithms using this model. ***Cache-adaptive*** algorithms use the cache optimally in the face of memory fluctuations. They then introduced techniques for analyzing some classes of divide-and-conquer algorithms in the cache-adaptive model. We use "adaptive" and "cache-adaptive" interchangeably throughout the rest of this paper.

Cache Analysis Without Changing Memory

Despite the reality of dynamic memory fluctuations, the majority of theoretical work on external-memory computation [35, 36] assumes a fixed cache size M. In the external-memory model, algorithm performance is determined by the number of I/Os, or fetches from external memory, that an algorithm takes to finish its computation. Many I/O-efficient algorithms in the fixed-memory model suffer from poor performance when M changes due to thrashing if the available memory drops.

Cache-oblivious algorithms [14, 18, 19] provide insight about how to design optimal algorithms in the face of changing memory. Notably, cache-oblivious algorithms are not parameterized by the cache or cache line size. Instead, they leverage their recursive algorithm structure to achieve cache-optimality under static memory sizes. They are often more portable because they are not tuned for a specific architecture. Although Bender *et al.* [9] showed that not all cache-oblivious algorithms are adaptive, many cache-oblivious algorithms are in fact also cache-adaptive. Many cache-oblivious algorithms are also adaptive. The connections between adaptivity obliviousness suggest that modifying the recursive structure of the non-adaptive cache-oblivious algorithms may be the key to designing optimally cache-adaptive algorithms.

Algorithms designed for external memory efficiency may be especially affected by memory level changes as they depend on memory locality. Such programs include external-memory algorithms, shared-memory parallel programs, database joins and sorts, scientific computing applications, and large computations running on shared hardware in cloud computing.

Practical approaches to alleviating I/O-latency constraints include techniques such as latency hiding. Latency hiding [21, 33] is a technique that leverages computation time to hide I/O latency to improve overall program performance. Since our model counts computation as free (i.e. it takes no time), we cannot use latency hiding because it requires nonzero computation time.

Analysis of Cache-Adaptive Algorithms

Prior work took important steps toward closing the separation between the reality of machines with shared caches and the large body of theoretical work on external-memory algorithms in a fixed cache. Existing tools for design and analysis of external-memory algorithms that assume a fixed memory size often cannot cope with the reality of changing memory.

Barve and Vitter [6, 7] initiated the theoretical study of algorithms with memory fluctuations by extending the disk-access machine (DAM) model to accommodate changes in cache size.

Specifically, they allow the memory size M to change and present complex but optimal algorithms for sorting, FFT, matrix multiplication, LU decomposition, permutation, and buffer trees. Their work proves the existence of optimal algorithms in spite of memory changes but lacks a framework for finding such algorithms.

Bender *et al.* [10] continued the theoretical study of external-memory algorithms in the face of fluctuating cache sizes. They formalized the "cache-adaptive model", which allows memory to change more frequently and dramatically than Barve and Vitter's model, and introduced "memory profiles", which define sequences of memory fluctuations.

They showed that some (but not all) optimal cache-oblivious algorithms are optimal in the cache-adaptive model. At a high level, an algorithm is "optimal" in the cache-adaptive model if it performs well under all memory profiles. Specifically, if a recursive cache-oblivious algorithm fits in the form $T(n) = aT(n/b) + O(1)$, it is optimally cache-adaptive. They also showed that lazy funnel sort [11] is optimally cache-adaptive and does not fit into the given form. Despite the close connection between cache-obliviousness and cache-adaptivity, they show that neither external-memory optimality nor cache-obliviousness is necessary or sufficient for cache-adaptivity. The connections and differences between cache-oblivious and cache-adaptive algorithms suggest that cache-adaptive algorithms may one day be as widely used and well-studied as cache-oblivious algorithms.

In follow-up work, Bender *et al.* [9] gave a more complete framework for designing and analyzing cache-adaptive algorithms. Specifically, they completely characterize when a linear-space-complexity Master-method-style or mutually recursive linear-space-complexity Akra-Bazzi-style algorithm is optimal in the cache-adaptive model. For example, the in-place recursive naive[1] cache-oblivious matrix multiplication algorithm is optimally cache-adaptive, while the naive cache-oblivious matrix multiplication that does the additions upfront (and not in-place) is not optimally cache-adaptive. They provide a toolkit for the analysis and design of cache-oblivious algorithms in certain recursive forms and show how to determine if an algorithm in a certain recursive form is optimally cache-adaptive and if not, to determine how far it is from optimal.

Although these results take important steps in cache-adaptive analysis, open questions remain. The main contribution of Bender *et al.* [9] was an algorithmic toolkit for recursive algorithms in specific forms. At a high level, cache-oblivious algorithms that have long ($\omega(1)$ block transfers) scans[2] (such as the not-in-place n^3 matrix multiplication algorithm) in addition to their recursive calls are not immediately cache-adaptive. However, there exists an in-place, optimally cache-adaptive version of naive matrix multiplication. Is there a way to transform other algorithms that do $\omega(n/B)$ block transfers at each recursive call (where B is the cache line size in words), such as Strassen's algorithm [32], into optimally cache-adaptive algorithms? Our ***scan-hiding*** technique answers this question for Strassen and other similar algorithms. Furthermore, Bender *et al.* [10] gave a worst-case analysis in which the non-adaptive naive matrix multiplication is a $\Theta(\lg N)$ factor off from optimal. Does the predicted slow down manifest in reality? We begin to answer this question via experimental results in this paper.

Contributions

The contributions of this paper are twofold:

First, we develop a new technique called ***scan-hiding*** for making a certain class of non-cache-adaptive algorithms adaptive and use it to construct a cache-adaptive version of Strassen's algorithm for matrix multiplication in Section 4. Strassen's algorithm involves linear scans in its recurrence, which makes the algorithm as described non-adaptive via a theorem from Bender *et al.* [9]. We generalize this technique to

[1] We use "naive" matrix multiplication to refer to the $O(n^3)$ work algorithm for matrix multiplication.

[2] That is, the recurrence for their cache complexity has the form $T(n) = aT(n/b) + \Omega(n/B)$ where B is the cache line size in words.

many algorithms which have recurrences of the form $T(n) = aT(n/b) + O(n^c)$ in Section 3. This is the first framework in the cache-adaptive setting to transform non-adaptive algorithms into adaptive algorithms.

Next, we empirically evaluate the performance of various algorithms when subject to memory fluctuations in Section 5. Specifically, we compared cache-adaptive and non-adaptive naive matrix multiplication.

We also tested cache-oblivious funnel sort [28], `stxxl::sort` from `STXXL` [15] and `std::sort` [**?**] from the `C++` standard library and report our results in Appendix B.

Our results suggest that algorithms that are "more adaptive" (i.e. closer to optimal cache-adaptivity) are more robust under memory changes. Moreover, we observe performance differences even when memory sizes do not change adversarially.

2 Preliminaries

In this section we introduce the disk-access model (DAM) [4] for analyzing algorithms in a static cache. We review how to extend the disk-access model to the cache-adaptive model [10] with changing memory. Finally, this section formalizes mathematical preliminaries from [9] required to analyze scan-hiding techniques.

Disk-Access Model

Aggarwal and Vitter [4] first introduced the disk-access model for analyzing algorithms in a fixed-size cache. This model describes I/O limited algorithms on single processor machines. Memory can reside in an arbitrarily large, but slow disk, or in a fast cache of size M. The disk and cache are divided into cache lines of size B (in bytes). Access to memory in the cache and operations on the CPU are free. If the desired memory is not in cache, however, a cache miss occurs at a cost of one unit of time. A line is evicted from the cache and the desired line is moved into cache in its place. We measure algorithm performance in this model by counting up the number of cache-line transfers. That is, algorithms are judged based on their performance with respect to B and M. Furthermore, we differentiate between algorithms which are parameterized by B or M, called ***cache-aware***, and those which do not make use of the values of the cache or line size, called ***cache-oblivious*** [18].

The ***cache-adaptive model*** [9, 10] extends the disk-access model to accommodate for changes in cache size over time. Since we use the cache-adaptive model and the same definitions for cache-adaptivity, we repeat the relevant definitions in this section.

Cache-Adaptive Analysis

First, we will give a casual overview of how to do cache-adaptive analysis. This is meant to help guide the reader through the array of technical definitions that follow in this section, or perhaps to give enough sense of the definitions that one may follow the ideas, if not the details, in the rest of the paper. For a more thorough treatment of this topic, please see the paper *Cache-Adaptive Analysis* [9].

In general, we want to examine how well an algorithm is able to cope with a memory size which changes with time. We consider an algorithm A good, or "optimally cache-adaptive" if it manages to be constant-competitive with the same algorithm A^* whose scheduler was given knowledge of the memory profile ahead of time .

To give our non-omniscient algorithm a fighting chance we also allow ***speed-augmentation*** where A gets to perform a constant number of I/Os in a given time step, whereas A^* still runs at one I/O per time step.

To prove an algorithm is cache-adaptive, we instead show it has a stronger condition called optimally progressing. An optimally progressing algorithm has some measure of "progress" it has made, and it is constantly accomplishing a sufficient amount of progress. The "progress bound" does not have to resemble any sort of real work being done by the algorithm, but has more general constraints and can be thought of more as an abstraction that amortizes what A is accomplishing. We pick our progress bound to be an upper

bound on our optimally scheduled algorithm and then show that our (speed-augmented) algorithm always accomplishes at least as much progress as A^*.

Small slices of time or strangely-shaped cache sizes often make analyzing algorithms difficult in the cache-adaptive model. For simplicity, we instead consider ***square profiles***, which are memory profiles that stay at the same size for a number of time steps equal to their size. Thus, when we look at a plot of these profiles, they look like a sequence of squares. There are two important square profiles for each given memory profile M: one that upper bounds and another that lower bounds the progress A can accomplish in M. Bender *et al.* [9] showed that these square profiles closely approximate the exact memory profile.

In short, proving an algorithm is cache-adaptive involves:

1. Picking a "progress function" (Definition 9) to represent work done by our algorithms.
2. Upper bound the progress A^* can make in a square of memory by a progress bound (Definition 13).
3. Show that a speed-augmented version of A can make at least as much progress as A^* given the same square of memory.

Achieving Cache-Optimality on any Memory Profile

Since the running time of an algorithm is dependent on the pattern of memory size changes during its execution, we turn to competitive analysis to find "good" algorithms that are close to an optimal algorithm that knows the future set of memory changes. We will now formalize what makes an algorithm "good" and how to analyze algorithms in spite of cache fluctuations.

Definition 1 (Memory Profile) *A **memory profile** M is a sequence (or concatenation) of memory size changes. We represent M as an ordered list of of natural numbers ($M \in \mathbb{N}^*$) where $M(t)$ is the value of the cache size (in words) after the t-th cache miss during the algorithm's execution. We use $m(t)$ to denote the number of cache lines at time t of memory profile M (i.e. $m(t) = M(t)/B$).*

The size of the cache is required to stay in integral multiples of the cache line size.

In general, an optimal algorithm A^* takes at most a constant factor of time longer than any other algorithm A for a given problem on any given memory profile. Since a memory profile might be carefully constructed to drop almost all of its memory after some specific algorithm finishes, we allow the further relaxation that an optimal algorithm may be speed augmented. This relaxation allows an algorithm to complete multiple I/Os during one time step, and can be thought of in a similar manner to the memory augmentation allowed during the analysis of Least Recently Used. Speed augmentation relates to running lower latency memory access. Since memory augmentation (giving an online algorithm more space as described in Definition 3) is a key technique in the analysis of cache-oblivious algorithms, speed augmentation is an important tool for proving algorithms optimal in the cache-adaptive model.

We now formally define these notions of augmentation.

Definition 2 (Speed Augmentation [10]) *Let A be an algorithm. We use A' to denote the **c-speed augmented** version of A that performs c I/Os in each step of the memory profile.*

Bender *et al.* [9] extended the notions of memory augmentation and the tall-cache assumption, common tools in the analysis of cache-oblivious algorithms, to the cache-adaptive model. The tall-cache assumption specifies that M must be a certain size in terms of B, ensuring there are enough distinct cache lines in the cache for certain algorithms.

We assume page replacement is done according to a least-recently-used policy, which incurs at most a constant factor more page faults (and therefore I/Os) more than an optimal algorithm [8] under constant space-augmentation [31].

Definition 3 (Memory Augmentation [10]) *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.*

Definition 4 (H-tall [9]) *In the cache-adaptive model, we say that a memory profile M is **H-tall** if for all $t \geq 0$, $M(t) \geq H \cdot B$ (where H is measured in lines).*

Definition 5 (Memory Monotone [9]) *A **memory monotone** algorithm runs at most a constant factor slower when given more memory.*

Intuitively, an optimally cache-adaptive algorithm for a problem $\mathcal{P}$ does as well as any other algorithm for $\mathcal{P}$ given constant speed augmentation.

Definition 6 (Optimally Cache-adaptive [10]) *An algorithm A that solves problem $\mathcal{P}$ 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.*

Notably, this definition of optimal allows algorithms to query the memory profile at any point in time but not to query future memory sizes. Optimally cache-adaptive algorithms are robust under any memory profile in that they perform asymptotically (within constant factors) as well as algorithms that know the entire memory profile.

Approximating Arbitrary Memory Profiles with Square Profiles

Since memory size may change at any time in arbitrary memory profiles, Bender *et al.* [10] introduced ***square profiles*** to approximate the memory during any memory profile and simplify algorithm analysis in the cache-adaptive model. Square profiles are profiles where the memory size stays constant for a time range proportional to the size of the memory.

Definition 7 (Square Profile [10]) *A memory profile M or m is a **square profile** if there exist boundaries $0 = t_0 < t_1 < \ldots$ such that for all $t \in [t_i, t_{i+1})$, $m(t) = t_{i+1} - t_i$. In other words, a square memory profile is a step function where each step is exactly as long as it is tall. Let $\square_{M(t)}$ and $\square_{m(t)}$ denote a square of M and m, respectively, of size $M(t)$ by $M(t)$ words and $m(t)$ and $m(t)$ lines, respectively.*

Definition 8 (Inner Square Profile [10]) *For a memory profile m, the **inner square boundaries** $t_0 < t_1 < t_2 < \ldots$ of m are defined as follows: Let $t_0 = 0$. Recursively define t_{i+1} as the largest integer such that $t_{i+1} - t_i \leq m(t)$ for all $t \in [t_i, t_{i+1})$.*

Bender *et al.* [10] showed that for all timesteps t, the size of inner square profile m' is at most a constant factor smaller than the size of the original memory profile m. If an algorithm is optimal on all square profiles, it is therefore optimal on all arbitrary profiles. Cache-adaptive analysis uses inner square profiles because they are easier to analyze than arbitrary profiles and closely approximate any memory profile. Figure 1 shows an example of a memory profile and its inner and outer square profiles.

Lemma 1 (Square Profile Usability [10]) *Let m be a memory profile where $m(t+1) \leq m(t) + 1$ 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, $m'(t) \leq m(t)$.*
2. *For all i, $m'(t_{i+1}) \leq 2m'(t_i)$.*
3. *For all i and $t \in [t_{i+1}, t_{i+2})$, $m(t) \leq 4(t_{i+1} - t_i)$.*

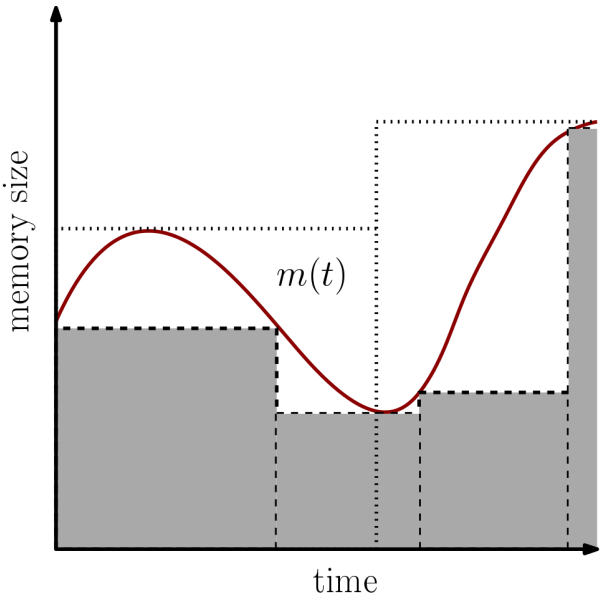


Figure 1: A memory profile m and its inner and outer square profiles. The red curving line represents the memory profile m, the grey dashed boxes represent the associated inner square profile, the dotted lines represent the outer square profile (as defined in [10]).

Optimally Progressing Algorithms are Optimally Cache-Adaptive

We show an algorithm is optimally cache-adaptive by showing a stronger property: that it is "optimally progressing". The ***progress*** of an algorithm is the number of cache accesses that occurred during a time interval. (In other words, we assume the algorithm makes one unit of progress per cache access.)

Intuitively, an optimally progressing algorithm has some notion of work it needs to accomplish to solve the given problem, and for any given interval of the memory profile that algorithm does within a constant factor as much work as the optimal algorithm would. An optimally progressing algorithm is optimally cache-adaptive [9].

To show an algorithm is optimal in the DAM model, we require an upper and lower bound on the progress any algorithm can make for its given problem. Let $\mathcal{P}$ be a problem. A "progress bound" $\rho_{\mathcal{P}}(M(t))$ or $\rho_{\mathcal{P}}(m(t))$ is an upper bound on the amount of progress any algorithm can make on problem $\mathcal{P}$ given memory profiles M or m with $M(t)$ or $m(t)$ cache size at time t, respectively. $\rho_{\mathcal{P}}(M(t))$ and $\rho_{\mathcal{P}}(m(t))$ are defined in terms of number of words and number of cache lines, respectively. We use a progress-requirement function to bound from below the progress any algorithm must be able to make. A progress requirement function $R_{\mathcal{P}}(N)$ is a lower bound on the amount of progress an algorithm must make to be able to solve all instances of $\mathcal{P}$ of size at most N.

We combine square profiles with progress bounds to show algorithms are optimally progressing in the cache-adaptive model. Cache-adaptive analysis with square profiles is easier than reasoning about arbitrary profiles because square profiles ensure that memory size stays constant for some time. Since the performance of an algorithm on an inner square profile is close enough to its performance on the original memory profile, we use inner square profiles in our progress-based analysis.

Finally, we formalize notions of progress over changing memory sizes. At a high level, we define progress of an algorithm A on an inner square profile $\square_M$ such that the sum of the progress of A over all of the squares of $\square_M$ is within a constant factor of the total progress of A on M. We call our progress function over a single square of square profile M and m, $\phi_A(\square_M)$ or $\phi_A(\square_m)$ since by definition a single square profile gives a cache size and our progress function ϕ_A takes as input a cache size.

Definition 9 (Progress Function [9]) *Given an algorithm A, a progress function $\phi_A(M(t))$ and $\phi_A(m(t))$ defined for A is the amount of progress that A can make given $M(t)$ words or $m(t)$ lines. We define the progress function given as input a memory profile M and m as $\Phi_A(M)$ and $\Phi_A(m)$ and it provides the amount of progress A can make on a given profile M and m, respectively.*

Let $M_1 \| M_2$ indicate concatenation of profiles M_1 and M_2.

Definition 10 ([9]) *Let M and U be any two profiles of finite duration. The duration of a profile M is the length of the ordered list representing M. Furthermore, let $M_1 \| M_2$ indicate concatenation of profiles M_1 and M_2. We say that **M is smaller than U, $M \prec U$,** if there exist profiles $L_1, L_2 \dots L_k$ and $U_0, U_1, U_2 \dots U_k$, such that $M = L_1 \| L_2 \dots \| L_k$ and $U = U_0 \| U_1 \| U_2 \dots \| U_k$, and for each $1 \leq i \leq k$,*

1. *If d_i is the duration of L_i, U_i has duration $\geq d_i$, and*
2. *as standalone profiles, L_i is always below U_i ($L_i(t) \leq U_i(t)$ for all $t \leq d_i$).*

Definition 11 (Monotonically Increasing Profiles [9]) *A function $f : \mathbb{N}^* \to \mathbb{N}$ which takes as its input a memory profile M is monotonically increasing if for any profiles M and U, $M \prec U$ implies $f(M) \leq f(U)$.*

Definition 12 (Square Additive [9]) *A monotonically increasing function $f : \mathbb{N}^* \to \mathbb{N}$ which takes as its input a single square $\square_M$ of a square profile M is square-additive if*

1. *$f(\square_M)$ is bounded by a polynomial in M,*
2. *$f(\square_{M_1} \| \cdots \| \square_{M_k}) = \Theta(\sum_{i=1}^{k} f(\square_{M_i}))$.*

Progress Bounds

We now formally define progress bounds in the cache-adaptive model and show how to use progress bounds to prove algorithms are optimally cache-adaptive. Given a memory profile M, a progress bound $\rho_{\mathcal{P}}(M(t))$ or $\rho_{\mathcal{P}}(m(t))$ is a function that takes a cache size $M(t)$ or $m(t)$ and outputs the amount of progress any algorithm could possibly achieve on that cache size.

Definition 13 (Progress Bound [9]) *A problem $\mathcal{P}$ of size N has a progress bound if there exists a monotonically increasing polynomial-bounded progress-requirement function $R : \mathbb{N} \to \mathbb{N}$ and a square-additive **progress limit function** $\mathbf{P}_{\mathcal{P}} : \mathbb{N}^* \to \mathbb{N}$ such that: For any profile M or m, if $\mathrm{P}_{\mathcal{P}}(M) < R_{\mathcal{P}}(N)$, then no memory-monotone algorithm running under profile M can solve all N. Let $\rho_{\mathcal{P}}(M(t))$ and $\rho_{\mathcal{P}}(m(t))$ for a problem $\mathcal{P}$ be a function $\rho_{\mathcal{P}} : \mathbb{N} \to \mathbb{N}$ that provides an upper bound on the amount of progress any algorithm can make on problem $\mathcal{P}$ given cache sizes $M(t)$ and $m(t)$. We also refer to both $\rho_{\mathcal{P}}$ and $\mathrm{P}_{\mathcal{P}}$ as the **progress bound** (although they are defined for different types of inputs).*

Throughout this paper, for purposes of clarity (when talking about a square profile), when we write $\rho_{\mathcal{P}}(\square_{M(t)})$ or $\rho_{\mathcal{P}}(\square_{m(t)})$, we mean $\rho_{\mathcal{P}}(M(t))$ and $\rho_{\mathcal{P}}(m(t))$.

Furthermore, we limit our analysis to "usable" memory profiles. If the cache size increases faster than an algorithm can pull lines into memory, then some of that cache is inaccessible and cannot be utilized. Thus we consider ***usable*** memory profiles.

Definition 14 (Usable Profiles [10]) *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.*

Next, we formalize "feasible" and "fitting" profiles to characterize how long it takes algorithms complete on various memory profiles.

Definition 15 (Fitting and Feasibility [9]) *For an algorithm A and problem instance I we say a profile M of length ℓ is **I-fitting** for A 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.)*

We now have the necessary language to formally define optimally progressing algorithms. Intuitively, optimally progressing algorithms do about as well as any other algorithm for the same problem regardless of the memory profile.

Definition 16 (Optimally Progressing [9]) *For an algorithm A that solves problem $\mathcal{P}$, 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 $\mathbf{P}_{\mathcal{P}}$** (or simply optimally progressing if $\mathrm{P}_{\mathcal{P}}$ is understood) if, for every integer N and N-feasible H-tall usable profile M, $\mathrm{P}_{\mathcal{P}}(M_N) = O(R_{\mathcal{P}}(N))$. We say that A is optimally progressing in the DAM model if, for every integer N and every constant H-tall profile M, $\mathrm{P}_{\mathcal{P}}(M_N) = O(R_{\mathcal{P}}(N))$.*

Bender *et al.* [9] showed that optimally progressing algorithms are optimally cache-adaptive.

Lemma 2 (Optimally Progressing Implies Adaptivity [9]) *If an algorithm A is optimally progressing, then it is optimally cache adaptive.*

Cache-adaptivity of Recursive Algorithms

We focus on recursive algorithms which decompose into sections which are somewhat cache intensive and linear scans in Section 3.

Definition 17 (Linear Scans) *Let $f(n)$ be a function of an input size n. A **linear scan** of size $O(f(n))$ is a sequence of operations which sequentially accesses groups of memory which are $O(B)$ in size and performs $O(B)$ (and at least one) operations on each group before accessing the next group.*

Finally, [9] analyzes recursive algorithms analogous to those in the Master Theorem. We use the following theorem about (a, b, c)-regular algorithms.

Definition 18 ((a, b, c)-regular [9]) *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 $(\boldsymbol{a})$ recursive calls on subproblems of size $(\boldsymbol{N/b})$, and*

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

Finally, we specify which algorithms we can apply our "scan-hiding" technique to. Scan-hiding generates optimally-adaptive algorithms from non-optimally-adaptive recursive algorithms with linear (or sublinear) scans. We can apply scan-hiding to $(\boldsymbol{a, b, c})$***-scan regular algorithms***.

Definition 19 ((a, b, c)-scan regular [9]) *Let $a \geq 1/b$, $0 < b < 1$, and $\mathcal{C} \geq 1$ be constants. A linear-space algorithm is $(\boldsymbol{a, b, c})$**-scan regular** if, for inputs of sufficiently large size N, it makes*

1. *exactly* $\boldsymbol{a}$ *recursive calls on subproblems of size* $\boldsymbol{N/b}$*, and*
2. $\Theta(1)$ *linear scans before, between, or after recursive calls, where the size of the biggest scan is* $\Theta(N^{\mathcal{C}})$ *where* $\log_b(a) > \mathcal{C}$*.*

Finally, we restate a theorem due to Bender *et al.* that determines which algorithms are immediately optimal and how far non-optimal algorithms are from optimal algorithms.

Theorem 3 ((a, b, c)**-regular optimality [9])** *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 function* $\phi_A(\square_N) = \Theta(N^p)$*, for constant* p*. Assume* $B \geq 4$*. 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* $\Theta\left(\lg_{1/b} \frac{N}{\lambda}\right)$ *away from being optimally progressing and* $O\left(\lg_{1/b} \frac{N}{\lambda}\right)$ *away from being optimally cache-adaptive.*

3 Generalized Scan Hiding

In this section, we present a generalized framework for converting non-adaptive algorithms into adaptive algorithms via "scan hiding". Our generalized scan-hiding procedure can be applied to Master-Theorem-style recursive algorithms that contain "independent" linear scans in each level of the recursion.

At a high level, scan hiding breaks up long (up to linear) scans at each level of a recursive algorithm and distributes the pieces evenly throughout the entire algorithm execution. We define a ***recursion tree*** as the tree created from a recursive algorithm A such that each node of the tree contains all necessary subprocesses for the subproblem defined by that node. Figure 2 shows an example of scan hiding on the recursion tree for Strassen's algorithm. Each node of the Strassen recursion tree contains a set of scans and matrix multiplication operations as its subprocesses.

We now formalize our generalized scan-hiding technique. We apply scan hiding to Strassen's algorithm in Section 4 as a case study of our technique.

More specifically, we can generate adaptive algorithms from non-adaptive "scan-hideable algorithms".

Definition 20 (Scan-hideable Algorithms) *We can apply scan hiding to non-adaptive algorithms with the following characteristics to convert them to adaptive algorithms:*

- *Let the input size be* $n^{\mathcal{C}}$*. For most functions* $\mathcal{C} = 1$*, however, for dense graphs and matrices* $\mathcal{C} = 2$*.*
- A *is a* (a, b, c)*-scan regular algorithm and has a runtime that can be computed as a function that follows the Master Theorem style equations of the form* $T(n) = aT(n/b) + O(n^{\mathcal{C}})$ *in the DAM model where* $\log_b(a) > \mathcal{C}$ *for some constants* $a > 0$*,* $b \geq 1$*, and* $\mathcal{C} \geq 1$*.*
- *In terms of I/Os, the base case of* A *is* $T(M) = \frac{M}{B}$ *where* M *is the cache size.*
- *We define* work *to be the amount of computation* in words *performed in a square profile of size* m *by* m *by some subprocess of* A*. A subprocess is more* work consuming *if it uses more work in a square profile of size* m *by* m*. For example, a naive matrix multiplication subprocess is more work consuming than a scan since it uses* $(mB)^{\log_2 3}$ *work as opposed to a scan which uses* mB *work. At each level a linear scan is performed in conjunction with a more work consuming subprocess (in the case of Strassen, for example, the linear scan is performed in conjunction with the more work consuming matrix multiplication).*

- *Each of the more work consuming subprocesses in each node of the recursion tree only depends on the the results of the scans performed in the subtrees to the left of the path from the current node to the root.*

- *If each node's scans depend on the result of the subprocesses of the ancestors (including the parent) of the current node in the computational DAG of the algorithm.*

Our scan hiding technique involves hiding all scans "inside" the recursive structure in subcalls. If an algorithm (e.g. Strassen) requires an initial linear scan for even the first subcall, we cannot hide the first scan in recursive subcalls. Therefore, we show that an algorithm A is optimally progressing even if A having an initial scan of $O(n^{\mathcal{C}})$ length. We will be using $A_{\mathsf{scan\_hiding}}$ as the name for the algorithm using this scan hiding technique.

Lemma 4 *If the following are true:*

- *The optimal A algorithm in the DAM model (i.e. ignoring wasted time due to scans) takes total work $n^{\lg_b(a)}$ and respects the progress bound $\rho(m(t)) = d_0(m(t)B)^{\log_b(a)/\mathcal{C}}$ where d_0 is a constant greater than 0. Let m be a profile that starts at time step 0 and ends at time step T where the optimal A algorithm completes.*

- *Let us assign potential in integer units to accesses, much as we do for work.*

- *$A_{\mathsf{scan\_hiding}}$ is an algorithm which computes the solution to the problem and has total work $d_1 n^{\lg_b(a)}$ and has total potential $d_2 n^{\lg_b(a)}$ and completes $c_3(mB)^{\log_b(a)/\mathcal{C}}$ work and potential in any m by m square profile where d_1, d_2 and d_3 are all constants greater than 0 and where $mB < n^{\mathcal{C}}$.*

- *Finally, $A_{\mathsf{scan\_hiding}}$ must also have the property that if the total work plus potential completed is $(d_1 + d_2)n^{\lg_b(a)}$, $A_{\mathsf{scan\_hiding}}$ is guaranteed to have finished its last access.*

Then $A_{\mathsf{scan\_hiding}}$ is cache-adaptive.

PROOF. Let $m'(t)$ be the inner square profile of $m(t)$. When $n^{\mathcal{C}} < m'(t)B$ the entirety of A completes and $A_{\mathsf{scan\_hiding}}$ will complete given a constant factor expansion.

The optimal A algorithm in the worst case makes a constant factor $d_4 \geq 1$ less progress on the inner profile for some constant d_4.

With time augmentation $\frac{1}{d_3 \cdot d_4}$, $A_{\mathsf{scan\_hiding}}$ completes as much progress on this square as A did in the associated part of the profile, so over the entire profile $A_{\mathsf{scan\_hiding}}$ completes at least $n^{\lg_b(a)}$ work and potential. With time augmentation $\frac{d_1 \cdot d_2}{d_3 \cdot d_4}$, $A_{\mathsf{scan\_hiding}}$ completes at least $(d_1 + d_2)n^{\lg_b(a)}$ work and potential.

Thus $A_{\mathsf{scan\_hiding}}$ must have completed. □

Finally, we prove that algorithm A is optimally progressing. Specifically, we show that any algorithm A with running time of the form $T(n) = aT(n/b)+O(n^{\mathcal{C}})$ and with the characteristics specified in Definition 20 is optimally progressing,

Theorem 5 *Given an algorithm A with running time of the form $T(n) = aT(n/b) + O(n^{\mathcal{C}})$ and with the characteristics as specified in Definition 20, then A is optimally progressing with progress bound $\rho(m(t)) = (m(t)B)^{\frac{\log_b(a)}{\mathcal{C}}}$. To manage all the pointers we also require $m(t) \geq \log_b n$ for all t.*

PROOF. For this proof, we charge all writeouts to read-ins; therefore, we do not specifically argue the cache-adaptivity of the writeouts of our algorithm. Given any recursive algorithm that admits a recursive runtime equation that has the form $T(n) = aT(n/b) + O(n^{\mathcal{C}})$ in the DAM model, the required amount of

work by the algorithm is $O\left(n^{\log_b(a)}\right)$ and an optimally progressing algorithm performs $\Omega\left((mB)^{\frac{\log_b(a)}{\mathcal{C}}}\right)$ work for each inner square with side length m in the inner square profile of the usable profile.

We assume by the conditions of algorithms that fall under our framework that we must first perform a scan of size $O(n^{\mathcal{C}})$ as preprocessing. We show that the amount of total potential during the time the scan is performed does not exceed $O\left(n^{\log_b(a)}\right)$. Then, by Lemma 4, we know that A is still optimally progressing. We assign $O(n^{\log_b(a)-\mathcal{C}})$ potential to each of the $O(n^{\mathcal{C}})$ scans; thus for each scan we complete $\Omega\left(n^{\log_b(a)-\mathcal{C}}mB\right) = \Omega\left((mB)^{\frac{\log_b(a)}{\mathcal{C}}}\right)$ progress, as long as $m < n^{\mathcal{C}}$.

Scan hiding intersperses the scans with the more work consuming processes associated with the other parts of the algorithm A resulting in $\Omega\left((mB)^{\frac{\log_b(a)}{\mathcal{C}}}\right)$ work to be done at each level of the recursion in any m by m square where at least half of the cache misses are used to perform this work. We present an example of scan hiding on Strassen's algorithm in Appendix A.

In a recursive call solving a subproblem of size s this subproblem will be responsible for a scan of length at most $O(s + \log_b(n))$. Amortized each scan is of length $O(s)$, however, if pointers are passed around naively one may deal with the additive $O(\log_b(n))$ scan for some particular problem. The condition that $m(t) \geq \log_b n$ allows for naive splits of the sizes of scans to be acceptable by making the additive factor $\log_b(n)$ at most a factor of 2 of the size of the solved problem.

Thus, algorithm A is optimally progressing.

□

Since scan hiding amortizes the work of part of scan against each leaf node of the recursion tree, each leaf node must be sufficiently large to hide part of a scan. Therefore, we insist that $m(t) \geq \log_b n$. Note that given a specific problem one can usually find a way to split the scans such that this requirement is unnecessary. For general scan hiding, however, we use this minimum cache size to make passing pointers to scans easy and inexpensive.

As an immediate consequence of Theorem 5 above, we get the following corollary.

Corollary 6 *Given an algorithm A with running time of the form $T(N) = aT(N/b) + O(N)$ and with the characteristics as specified in Definition 20, A is cache-adaptive. If a node's subprocesses depend on the scans of the nodes in the left subtree, then we also require $m(t) \geq \log n$.*

Scan hiding directly broadens Theorem 7.3 in [9] to show cache-adaptivity for a specific subclass of Master Theorem style problems when $\log_b(a) > \mathcal{C}$.

4 Scan Hiding in Strassen's Algorithm

In this section, we apply scan hiding to Strassen's algorithm for matrix multiplication to produce an optimally cache-adaptive variant of Strassen from the classic non-optimal version. We call our modified adaptive version of Strassen `AdaptiveStrassen`. We sketch how to spread the scans out throughout the algorithm's recursive structure and show that our technique results in an optimally adaptive algorithm. For more details, pseudocode of the algorithm, and exact proofs of optimality, see Appendix A.

Currently the most efficient matrix multiplication algorithms in practice for very large matrices use the Strassen algorithm [**? ?**]. For example, the GNU Multi-Precision Library uses Strassen to perform matrix multiplication for decimal digits in the range $10,000$ to $40,000$, and the Java uses its Strassen implementation for above $74,000$ decimal digits. Previous work has shown that naive matrix multiplication that combines results at the end in a matrix addition is not cache-adaptive, but that it can be altered to be cache-adaptive by doing all the work in place [10].

In the classical Strassen algorithm, each recursive call begins with an addition of matrices and ends with an addition of matrices (see Appendix A for a discussion of Strassen in the RAM model). Doing this work in

place still leaves a recurrence of $T(n) = 7T(n/2) + O(n^2)$, which is not optimally progressing. Each of these sums relies on the sums the parent computed. In naive cubic matrix multiplication, the input to each leaf in the recurrence tree can be read off of the original matrix. In Strassen's algorithm, however, the input to each recursive subcall is generated by alterning the input from its parent. Specifically, the input at each level is the result of the multiplication of one submatrix and the sum of two submatrices. To produce the input for a sub-problem generated by Strassen of size 1 by 1, one would need to read off n values from the original matrix, resulting in a runtime of $O(n^{\lg(7)+1})$. The inplace approach therefore does not work because it introduces too much work at each leaf. We describe an adaptive version of Strassen via scan hiding.

Adaptive Strassen is Optimally Progressing

We will now describe the high level idea of `AdaptiveStrassen`.

The primary issue with naive Strassen[3] is the constraint that many of the summations in the algorithm must be computed before the next recursive call, leading to long blocks of cache-inefficient computation. The main insight behind our adaptive Strassen is that all of these summations do not need to be executed immediately prior to the recursive call that takes their result as input. We are thus able to spread out these calculations among other steps of the algorithm making its execution more "homogeneous".

Our algorithm requires an awareness of the size of the cache line B, though it can be oblivious to the size of the cache ($m(t)$ or $M(t)$). We further require that the number of cache lines in $m(t)$ be at least $\lg(n)$ at any time ($m(t) = \Omega(\lg(n))$).

To begin, the algorithm takes as input a pointer to input matrices X and Y. We are also given the side length of the matrix n and a pointer pZ to the location to write $X \times Y$.

We will create a set of arrays which will keep track of the pre-computed sums of matrices for input and an array to keep track of the post-computed sums of matrices for the output.

`AdaptiveStrassen` starts and ends with scans of length $O(n^2)$, which are small enough to still allow for adaptivity.

`AdaptiveStrassen` spreads the remaining sums evenly across sub-problems of size $\Omega(\sqrt{\lg(n)B})$. Intuitively, by spreading out our scans we avoid large stretches where we cannot use the large amount of memory as memory grows. More concretely, it lets us guarantee that in any box larger than $\lg(n)B$ we solve a problem that uses up at least a constant factor fraction of the memory allotted by the square.

We call the ***leftmost path*** of a node to be that node's left child and its left child downward until a leaf is reached. Similarly a ***rightmost path*** of a node is the path formed by all the right children from a node and its right decedents. In order to compute a sum at level i of the recursion, we must have saved its parent's sum from level $i-1$ of the recursion. Therefore, `AdaptiveStrassen` first computes all the scans in the leftmost path from the root of the recursion tree as inputs to the following subcalls.

At a high level, scan hiding "homogenizes" the long linear scans at the beginning of each recursive call across the entire algorithm. We will precompute each node's matrix sum by having an earlier sibling do this summation spread throughout its execution. Figure 2 shows an example of how to spread out the summations. We need to do some extra work to keep track of this precomputation, however. Specifically, we need to read in $O(\lg(n))$ bits and possibly incur $O(\lg(n))$ cache misses. `AdaptiveStrassen` is optimally progressing as long as $M(t) = \Omega(\lg(n)B)$.

We hide all "internal" scans throughout the algorithm and do the first scan required for the first recursive call of Strassen upfront. The first recursive call of Strassen does the precomputation and set up of $O(n^2)$ sized scans as well as the post-computation for the output which also consists of $O(n^2)$ scans.

This algorithm is described in Algorithm 2. We sketch the recursion tree of `AdaptiveStrassen` in Figure 2.

[3]We use naive Strassen to denote the canonical Strassen implementation that does scans at each node of the recursion tree.

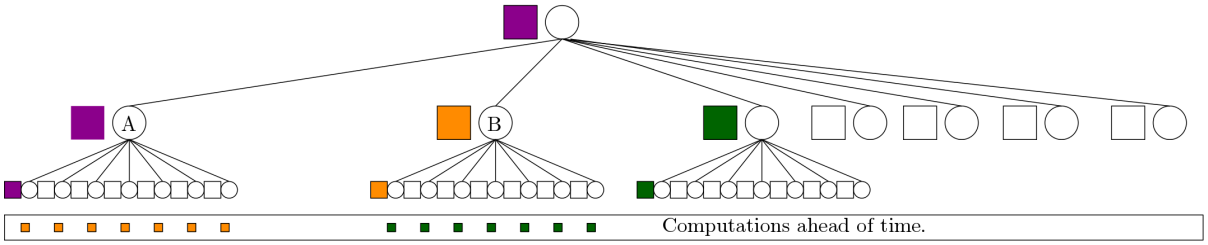


Figure 2: A sketch of scan hiding in Strassen's algorithm. The purple squares represent the scans on the rightmost path of the root. The orange squares represent the rightmost path of the second child of root. The scans represented in orange will be completed before the recursive call for B is made. The scans will be completed by the leaf nodes under A. We represent the look-ahead computations as the small squares in the rectangle labeled "Computations ahead of time".

We use an array P to precompute and store the sums needed for the input to the leftmost paths and the output sums needed for rightmost paths. P stores pointers to the $\lg(n)$ arrays which point to the arrays responsible for rightmost and leftmost paths of nodes at each of the $\lg(n)$ levels. (Note that there can be only one leftmost path from a node at level L being actively read from and only one that can be actively written to.)

Next, we will show that `AdaptiveStrassen` is adaptive using a potential argument to bound the non-optimality of `AdaptiveStrassen` on any profile. Given a memory profile M, we assign progress plus work in each square of M of a such that the total progress plus work done over the execution is $O(n^{\lg(7)})$, and on any $m(t)$ by $m(t)$ block of execution $\Omega((mB)^{\lg(7)})$ progress plus work are completed.

`AdaptiveStrassen` takes $O(n^{\lg(7)})$ time in the word-RAM model by Lemma 12.

Theorem 7 *Let X, Y be two matrices of size n^2 (side length n). Adaptive Strassen is optimally cache adaptive over all memory profiles m when $m(t) > \lg(n)$ $\forall t$ and the algorithm is aware of the size of the cache line size B with respect to the progress function $\rho(m(t)) = (m(t)B)^{\lg(7)/2}$.*

PROOF.

We can apply Theorem 5 where $a = 7$ and $b = 4$ because the Strassen recurrence is $T(N) = 7T(N/4) + O(N)$ when $N = n^2$.

The proof of adaptivity relies on two main accounting Lemmas: one for the $O(n^2)$ linear scan, and one for analyzing the recursive subcalls. Intuitively, the initial and final linear scans maintain adaptivity because: any algorithm with input size n^2 which reads its entire input must take n^2/B time. Therefore, a scan of size n^2 must take less time than running an algorithm with the recurrence $T(N) = 7T(N/4) + O(1)$ where N is the input size. We provide the full proof of adaptivity in

Or for concreteness we can apply Lemma 9 to both Lemma 10 and Lemma 11. Lemma 10 and Lemma 11 show that every $m(t) \times m(t)$ box completes $\Omega((mB)^{\lg_2(7)})$ work plus progress with total assigned progress $O(n^{\lg_2(7)})$. Thus, `AdaptiveStrassen` is cache adaptive by Lemma 9.

□

5 Experimental Results

We compare the faults and runtime of `MM-Scan` and `MM-Inplace` as described in [9] in the face of memory fluctuations. `MM-Inplace` is the in-place divide-and-conquer naive multiplication algorithm, while `MM-Scan` is not in place and does a scan at the end of each recursive call. `MM-Inplace` is cache adaptive while `MM-Scan` is not. We also empirically evaluate external-memory sorting algorithms in Appendix B.

Furthermore, we also compare the performance of different external memory sorting algorithms under memory changes.

We test various algorithms and measure their faults and runtime with randomized profiles as described later in comparison to their performance without induced memory changes.

Each point on the graphs in Figures 3 and 6 represents the ratio of the average number of faults (or runtime) during the changing memory profile to the average number of faults (or runtime) without the modified adversarial profile.

We find that cache-adaptive (or nearly cache-adaptive) algorithms incur fewer faults with random memory fluctuations than algorithms that are farther from optimally progressing, lending empirical support to the cache-adaptive model.

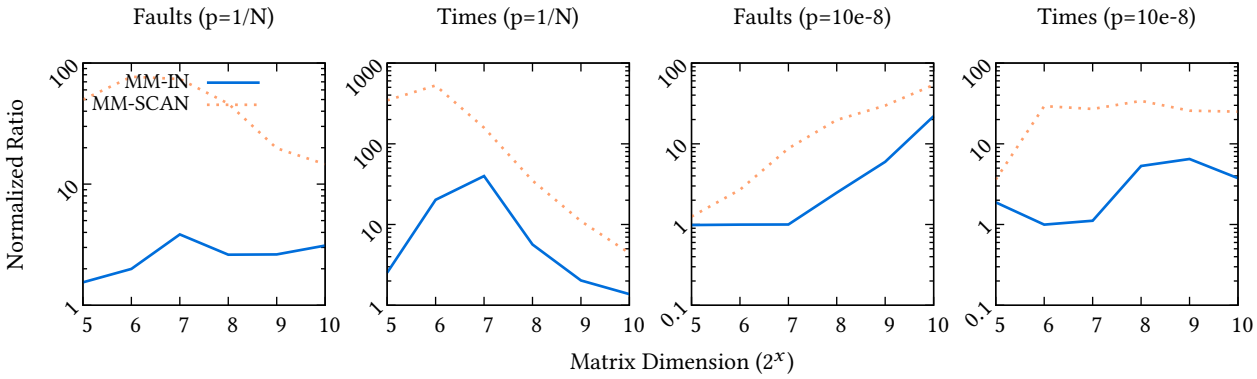


Figure 3: An empirical comparison of faults and runtime of `MM-Scan` and `MM-Inplace` under memory fluctuations. Each plot shows the normalized faults or runtime under a randomized version of the worst-case profile.
The first two plots show the faults and runtime during a random profile where the memory drops with probability $p = 1/N$ at the beginning of each recursive call.
Similarly, in the last two plots, we drop the memory with probability $p = 5 \times 10^{-8}$ at the beginning of each recursive call. Recall that the theoretical worst-case profile drops the memory at the beginning of each recursive call.

Naive Matrix Multiplication

We implemented `MM-Inplace` and `MM-Scan` and tested their behavior on a variety of memory profiles. The worst-case profile as described by Bender *et al.* [10] took too long to complete on any reasonably large input size for `MM-Scan`.

We measured the faults and runtime of both algorithms under a fixed cache size and under a modified version of the adversarial memory profile for naive matrix multiplication. Figure 3 shows the runtime and faults of both algorithms under a changing cache normalized against the runtime and faults of both algorithms under a fixed cache, respectively. We also provide details of the profile

Figure 3 shows that the relative number of faults that `MM-Scan` incurs during the random profile is higher than the corresponding relative number of faults due to `MM-Inplace` on a random profile drawn from the same distribution. As Bender *et al.* [9] shows, `MM-Scan` is a $\Theta(\lg N)$ factor from optimally progressing on a worst-case profile while `MM-Inplace` is optimally progressing on all profiles.

The relative faults incurred by `MM-Scan` grows at a non-constant rate. In contrast, the performance of `MM-Inplace` decays gracefully in spite of memory fluctuations. The large measured difference between `MM-Scan` and `MM-Inplace` may be due to the overhead of repopulating the cache after a flush incurred by `MM-Scan`.

System

We ran experiments on a node with and tested their behavior on a node with a two core Intel® Xeon™ CPU E5-2666 v3 at 2.90GHz. Each core has 32KB of L1 cache and 256 KB of L2 cache. Each socket has 25 Megabytes (MB) of shared L3 cache.

6 Conclusion

We present the first constructive method for converting non-adaptive recursive divide-and-conquer algorithms with scans into adaptive recursive algorithms through a new scan-hiding technique. For example, Strassen's algorithm for matrix multiplication is not immediately adaptive because of the scan at the beginning and end of each recursive call. Our construction applies to Strassen, Coppersmith-Winograd, Vassilevska Williams and Legall's ($o(n^3)$) matrix multiplication algorithms [32][13][34][20]. Scan hiding applies to all matrix multiplication algorithms which achieve their bound by bounding the matrix multiplication tensor, which include all currently known subcubic matrix multiplication algorithms.

Furthermore, our experiments suggest that the cache-adaptive model captures real-world performance trends. The adaptive naive matrix multiply performed significantly better even under variants of the theoretical worst-case profile. Our results suggest that performance differences due to cache adaptivity are not restricted to a theoretical, pathological case.

Our results and prior work raise both theoretical and experimental questions. For example, our technique of scan-hiding applies to many recursive algorithms but may not work for others with a superlinear step in the beginning. One example is cache oblivious 3SUM which begins by constructing a size $n \lg n$ data-structure in $sort(n) \lg n$ time and in each recursive step scans through $n'M/\lg M$ elements where n' is the size of the problem in the recursion[5]. Additionally, algorithms that start with long scans do not immediately admit cache-adaptive algorithms. Notably, Karstadt and Schwartz [23] gave an algorithm for matrix multiplication that saves a constant factor of $5/6$ but starts with a scan of length $O(n^2 \lg(n))$. Is there a variant of scan hiding for even longer scans? Are there techniques to port other algorithms and data structures to the cache-adaptive model?

Prior work gave constructions of worst-case profiles and showed that non-adaptive algorithms are not optimal on such profiles. Measuring natural memory fluctuations on real systems may give us insight into the real-world performance impact of cache-adaptivity.

Finally, we generate an adaptive algorithm from a non-adaptive algorithm in the case of Strassen. We also produce a general theorem for transforming many $(a, b, 1)$-regular algorithms (those with recurrences of the form $T(n) = aT(n)/b + n$) into cache-adaptive variants. This is the first instance of a general transformation on algorithms to generate adaptive algorithms. Our findings suggest that other classes of algorithms that are not initially adaptive may become adaptive through techniques such as scan-hiding. A future research direction includes building a full framework for cache-adaptive algorithmic transformations.

We conclude by explaining why we are optimistic about cache adaptivity. Classical external memory and cache-oblivious algorithms are well-studied and motivated by modern computer architectures with hierarchical memory. Practitioners have empirically studied performance in the face of memory fluctuations for years and have developed heuristics and experimentally fast algorithms for major operations such as database sorts and joins. However, the need for theoretical guarantees of cache adaptivity will only grow with the rise of multicore architectures and shared-memory programs.

Acknowledgements We would like to thank Michael Bender and Rob Johnson for their writing input and technical comments. Thanks to Rishab Nithyanand for contributions to the experimental section. We would like to thank Rezaul Chowdhury, Rathish Das, and Erik D. Demaine for helpful discussions. Finally, thanks to the anonymous reviewers from SPAA 2018 for their comments.

A `AdaptiveStrassen` Specification and Proofs of Optimality

In this section we review the details of Strassen's algorithm and show that a straightforward implementation with linear scans is not optimally progressing. Additionally, we apply scan hiding to Strassen as a concrete example of our technique and show that the resulting algorithm, called `AdaptiveStrassen`, is adaptive.

Strassen's Algorithm in the RAM Model

We provide the equations and pseudocode for Strassen's matrix multiplication algorithm in Algorithm 1. Recall that the recurrence for the runtime for Strassen's algorithm for multiplying two $n \times n$ matrices is $T(n) = 7T(n/2) + O(n^2)$ (in the RAM model).

STRASSEN(X, Y, Z):
Let X, Y be input matrices and Z be the output matrix. We define the matrix quadrants as follows for X (quadrants for Y and Z are defined in the same way):

$$X = \begin{bmatrix} X_{1,1} & X_{1,2} \\ X_{2,1} & X_{2,2} \end{bmatrix}.$$

Strassen's algorithm recursively computes 7 intermediate matrix products with 10 linear scans:

$$\begin{aligned}
S_1 &= (X_{1,1} + X_{2,2}) \cdot (Y_{1,1} + Y_{2,2}) \\
S_2 &= (X_{2,1} + X_{2,2}) \cdot Y_{1,1} \\
S_3 &= X_{1,1} \cdot (Y_{1,2} - Y_{2,2}) \\
S_4 &= X_{2,2} \cdot (Y_{2,1} - Y_{1,1}) \\
S_5 &= (X_{1,1} + X_{1,2}) \cdot Y_{2,2} \\
S_6 &= (X_{2,1} - X_{1,1}) \cdot (Y_{1,1} + Y_{1,2}) \\
S_7 &= (X_{1,2} - X_{2,2}) \cdot (Y_{2,1} + Y_{2,2}) .
\end{aligned}$$

The quadrants of the resulting Z matrix can be computed in terms of $S_1, \ldots, S_7$ as follows:

$$\begin{aligned}
Z_{1,1} &= S_1 + S_4 - S_5 + S_7 \\
Z_{1,2} &= S_3 + S_5 \\
Z_{2,1} &= S_2 + S_4 \\
Z_{2,2} &= S_1 - S_2 + S_3 + S_6 .
\end{aligned}$$

Algorithm 1: Strassen's algorithm for matrix multiplication.

Naive Strassen is Not Optimally Progressing

We refer to the straightforward implementation of Algorithm 1 as Naive Strassen.

Definition 21 *Naive Strassen computes the* 10 *matrix sums needed to produce the input for its* 7 *recursive calls, makes its* 7 *recursive calls each of which return an associated output matrix, then does the necessary* 8 *matrix sums to produce its output. This algorithm results in a recurrence of the form* $T(n) = 7T(n/2) + O(n^2)$*, or in terms of its input size* $N = n^2$ *we have the recurrence* $T(N) = 7T(N/4) + O(N)$.

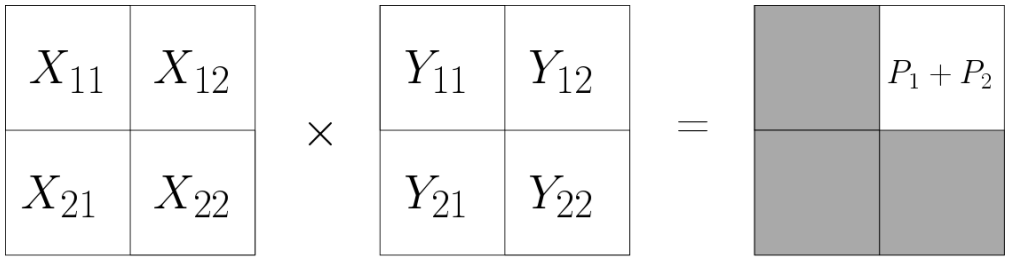
Lemma 8 *Naive Strassen is* not *optimally progressing with respect to the progress bound* $\mathrm{P}_{\mathcal{P}}(M) = M^{\lg(7)/2}$*, even if* $\forall t, M(t) > B \lg(n)$.

PROOF. By Theorem 7.3 from [9], if an algorithm with linear space complexity has a recurrence of the form $T(N) = aT(N/b) + O(N^c)$ with a tall cache assumption that $M(t) > \lg(n)B$ then the algorithm is a $\lg(N/(B \lg(n)))$ factor[4] away from being optimally progressing [9]. □

[4]This is only a constant if the full input size fits in $O(B \lg(n))$ words, which is a very small input size.

AdaptiveStrassen Example

We illustrate a small example of computing the upper right 2×2 result submatrix of a 4×4 Strassen call in Figure 4.



$$T_{11} = X_{11} \qquad P_1 = T_{11}T_{12}$$
$$T_{12} = Y_{12} - Y_{22} \qquad P_2 = T_{21}T_{22}$$
$$T_{21} = X_{11} + X_{12}$$
$$T_{22} = Y_{22}$$

Figure 4: Computing the upper right (2×2) output matrix of a 4×4 Strassen call. Before we can compute result submatrix, we have to compute P_1 and P_2. Before computing P_1 and P_2, however, we have to compute T_{12} and T_{21} which are sums of input submatrices. We will discuss how and when these are pre-computed given that P_1 and P_2 are the first two recursive calls made.

We show a part of the recursion tree of `AdaptiveStrassen`, our modified version of Strassen via scan hiding in Figure 5. At a high level, the multiplications in the leaves are spread evenly around additions, or broken-up scans. This even mix is what "homogenizes" the program, allowing for all squares in a memory profile to make within a constant factor of optimal progress.

`AdaptiveStrassen` first sets up the pre- and post-scans as input and output to the entire algorithm. It then completes the remaining work of the algorithm in recusive calls. We provide pseudocode for `AdaptiveStrassen` in Algorithm 2, which calls `AdaptiveStrassenRecurse` (Algorithm 3) for its recursive subcalls. Each leaf of `AdaptiveStrassenRecurse` takes the scans it must compute as well as the elements it must multiply. Additionally, we define a subroutine `ReturnSplitScans`(scans,B) that takes an array of pointers to scans as input and splits the scans into even portions, but of size no shorter than B. When the total size of scans is $< 7B$ some children will be handed empty lists.

We formalize how each node in the recursion tree distributes scans to its siblings in Algorithm 6.

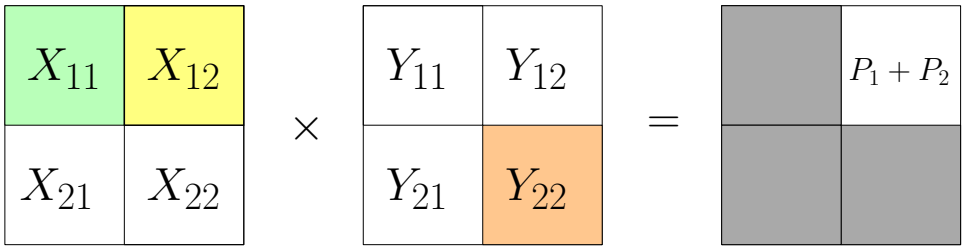
We showed that `AdaptiveStrassen` is optimally progressing in Section 3 but use the following specific proofs for conreteness and details of how scan hiding results in optimally progressing algorithms.

We first describe an assignment of work such that the total work done over the execution is $O(n^{log_2(7)})$ and that on any $m(t) \times m(t)$ block of execution, $\Omega((mB)^{\lg(7)/2})$ work is done. An optimal algorithm OPT for Strassen has $\Theta(n^{\lg(7)})$ accesses and has a progress function of $\rho(m(t)) = (m(t)B)^{\lg(7)/2}$.

We show adaptivity via a potential argument. At a high level, we assign "extra" work to each square of the square profile in which `AdaptiveStrassen` is not performing within a constant factor of OPT. The initial scans in `AdaptiveStrassen` may not be optimally progressing, but we describe an assignment of work and potential such that the overall algorithm is optimally progressing.

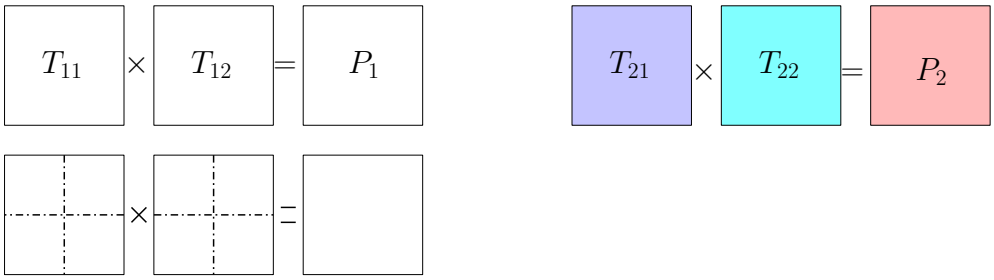
For example, suppose at time t_1 of an algorithm A can perform w_1 work on a square of size m_1 and at time t_2, A can perform w_2 work on a square of size m_1 where $w_1 < w_2$, then we can assign $w_2 - w_1$ *potential* to A at t_1.

If an A performs within a constant factor of OPT in terms of amount of *work and potential*, then it is cache-adaptive. Throughout most of the algorithm, we assign one unit of progress to each memory access an



$$T_{11} = X_{11}$$
$$T_{12} = Y_{12} - Y_{22}$$
$$T_{21} = X_{11} + X_{12}$$
$$T_{22} = Y_{22}$$

$$P_1 = T_{11}T_{12}$$
$$P_2 = T_{21}T_{22}$$



In the intial scan T_{11} and T_{12} will be pre-computed.

Seven Mulitplications for P_1 (dashed squares). Interspersed are additions (solid lines).

The matricies T_{21} and T_{22} must be computed before the mulitplication of P_1 finishes.

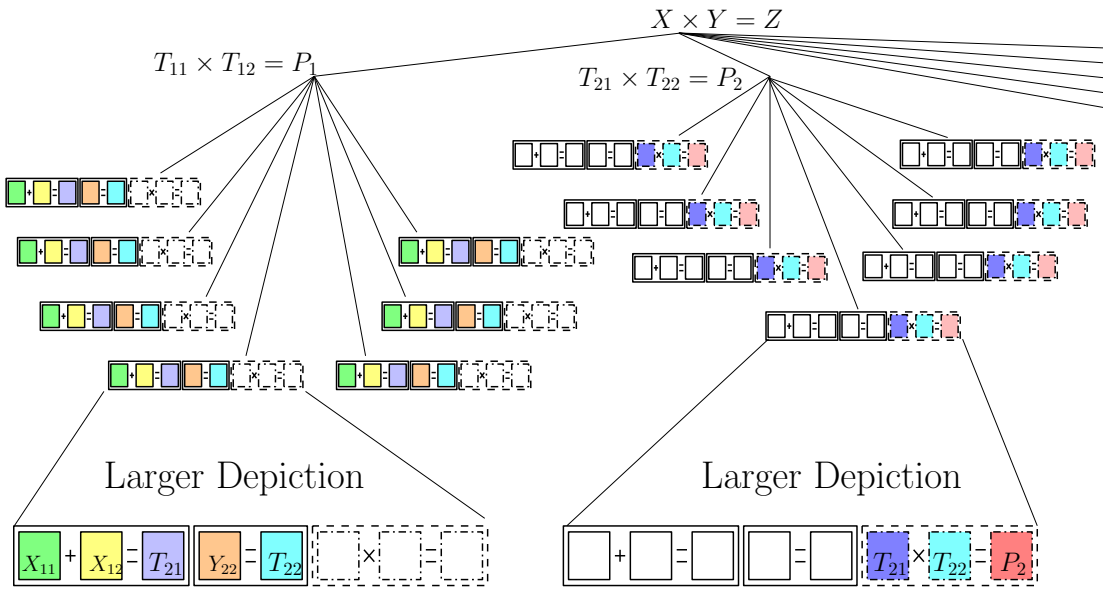


Figure 5: The pre-computation scan of size $O(n^2)$ would in this case pre-compute T_{11} and T_{21}. Then, all multiplications can be done. Assume that the smallest size of subproblem (3 small boxes) fit in memory. Then we show how the (dotted line boxes not filled in) multiplications needed for P_1 can be inter-spersed with the (complete line and colored in) additions or scans needed to pre-compute T_{21} and T_{22}. Note that T_{21} and T_{22} will be done computing before we try to compute the multiplication of P_2. Thus, we can repeat the process of multiplies interspersed with pre-computation during the multiplication for P_2. The additions or scans during P_2 will be for the inputs to the next multiplication, P_3 (not listed here). The multiplications in P_2 are computed based on the pre-computed matrices T_{21} and T_{22} (dotted line boxes filled in).

ADAPTIVESTRASSEN(PX,PY,PZ,N):
▷We define the global variables
Set $start\_n = n$;
Set $numLeaves = 0$;
▷We use the length of a cache line, as mentioned.
Set $B =$ length of a cache line;
▷We initialize the sum arrays for input and outputs.
▷We need a place to store are pre-computed and post-computed scans.
for $s \in \{x_1 = 0, x_2 = 1, y_1 = 2, y_2 = 3, z_1 = 4, z_2 = 5\}$ **do**
  Set $P[s] =$ pointer array of length $\lg(n) + 1$;
  for $i \in [0, \lg(n)]$ **do**
    $P[s][i] =$ pointer array of length $i + 1$;
    for $j \in [0, i - 1]$ **do**
      $P[s][i][j] =$matrix of size 2^j by 2^j;
Set $IsActive =$ is a pointer array of length $\lg(n)$;
Set $IsPrecompute =$ is a pointer array of length $\lg(n)$;
for $i \in [0, \lg(n)]$ **do**
  $IsActive[i] = [P[x_1][i], P[y_1][i], P[z_1][i]]$;
  $IsPrecompute[i] = [P[x_2][i], P[y_2][i], P[z_2][i]]$;
▷Now initialize $P[x_1][\lg(n)]$ and $P[y_1][\lg(n)]$ to be the input matrices for the spines
DOPRECURSESPINE(IsActive)
▷We will let the input and scans be represented by pointers for two places to read and one to write.
▷Next, we will give the length of the input.
▷Finally, we will also for convenience use a last entry to mark $\times, +, -, copy$
Let $input = [P[x_1][\lg(n)], P[y_1][\lg(n)], P[z_1][\lg(n)], n, \times]$;
Let $scans = []$;
▷Make the recursive call to Strassen where we hide scans
ADAPTIVESTRASSENRECURSE$(n, \lg(n), input, scans)$
▷Output from multiplications need to be summed together
DOPOSTPROCESSSPINE(IsActive, IsPrecompute)

Algorithm 2: AdaptiveStrassen(pX,pY,pZ,n)

algorithm makes. We reassign progress in our potential argument and show that `AdaptiveStrassen` makes steady progress throughout its execution. Specifically, we assigned progress to the initial and end scans because they are "harder" and are not optimally progressing. We will refer to our reassigned progress as work.

Lemma 9 *Let the input matrices to an instance of Strassen have size $n \times n$. If the following are true:*

- *The optimal Strassen algorithm takes time $n^{\lg(7)}$ in the RAM model.*
- *The optimal Strassen algorithm respects the progress function $\rho_{\mathcal{P}}(\square_m(t)) = c_0 \left(m(t)\right)^{\lg(7)/2}$ where c_0 is a constant such that $c_0 > 0$. Let $M(t)$ be a profile that starts at time step 0 and ends at time step T when the optimal Strassen algorithm completes.*
- *A is an algorithm which computes matrix multiplication and has total work $c_1 n^{\lg(7)}$, total potential $c_2 n^{\lg(7)}$, and completes $c_3(m(t)B)^{\lg(7)/2}$ work+potential in any $m(t) \times m(t)$ square of a profile where c_1, c_2 and c_3 are all constants such that $c_1, c_2, c_3 > 0$ and where $mB < n^2$.*
- *If the total progress plus potential completed is $(c_1 + c_2)n^{\lg(7)}$ during A's execution, A is guaranteed to have finished its last access.*

```
ADAPTIVESTRASSENRECURSE(N,LEVEL, INPUT, SCANS):
if n > 1 then
    ADAPTIVESTRASSENRECURSESPLIT(n,level, input, scans);
else if n ≤ 1 then
    ▷Do the multiplication you were asked to do
    Set input[0] * input[1] = input[2];
    ▷Do the additions you were asked to do by your parent
    for scan in scans do
        ▷Read all the information about your scan out
        Set in1 = scan[0];
        Set in2 = scan[1];
        Set out = scan[2];
        Set length = scan[3];
        Set op = scan[4];
        for i ∈ [0, length − 1] do
            ▷Do the requested operation to the input and write to output
            out[i] = op(in1[i], in2[i]);
```

Algorithm 3: AdaptiveStrassenRecurse(n,level, input, scans)

```
PRESCAN(INX, INY, OUTX, OUTY, N):
▷Now we want to split the input scan into seven parts.
childScan[0].append([inX[0], inX[3n/4], outX, +]);
childScan[0].append([inY[0], inY[3n/4], outY, +]);
childScan[1].append([inX[2n/4], inX[3n/4], outX, +]);
childScan[1].append([inY[0], inY[0], outY, copy]);
childScan[2].append([inX[3n/4], inX[3n/4], outX, copy]);
childScan[2].append([inY[2n/4], inY[0], outY, −]);
childScan[3].append([inX[0], inX[1n/4], outX, +]);
childScan[3].append([inY[3n/4], inY[3n/4], outY, copy]);
childScan[4].append([inX[3n/4], inX[0], outX, −]);
childScan[4].append([inY[0], inY[2n/4], outY, +]);
childScan[5].append([inX[1n/4], inX[3n/4], outX, −]);
childScan[5].append([inY[2n/4], inY[3n/4], outY, +]);
```

Algorithm 4: preScan(inX, inY, outX, outY, n)

Then A is cache adaptive.

PROOF. Let work be progress plus potential.

Let M be a profile and M' be the inner square profile of M such that for all t, $m(t)B < n^2$. If $m(t)B \geq n^2$, the entire problem can fit into cache and A will complete given a constant factor expansion.

The optimal Strassen algorithm makes at most a constant factor $c_4 > 0$ less progress on the inner profile M' than it did on the original profile M.

A makes at least as much progress on each square of M with time augmentation $1/(c_3 \cdot c_4)$ as OPT does in the non-augmented corresponding square. Therefore, A completes at least $n^{\lg(7)}$ work and potential over the entire profile. A completes at least $(c_1 + c_2)n^{\lg(7)}$ work and potential with time augmentation $(c_1 \cdot c_2)/(c_3 \cdot c_4)$.

POSTSCAN(INZ, OUTZ):
$childScan[1].append([outZ, inZ[0], inZ[0], +])$;
$childScan[1].append([outZ, inZ[3], inZ[3], +])$;
$childScan[2].append([outZ, inZ[3], inZ[3], 1])$;
$childScan[3].append([outZ, inZ[1], inZ[1], +])$;
$childScan[3].append([outZ, inZ[3], inZ[3], +])$;
$childScan[4].append([outZ, inZ[0], inZ[0], +])$;
$childScan[4].append([outZ, inZ[2], inZ[2], +])$;
$childScan[5].append([outZ, inZ[0], inZ[0], -])$;
$childScan[5].append([outZ, inZ[1], inZ[1], +])$;
$childScan[6].append([outZ, inZ[3], inZ[3], +])$;
$childScan[7].append([outZ, inZ[0], inZ[0], +])$;

Algorithm 5: postScan(inZ, outZ)

Thus A must have completed. □

We show the recursive part of Cache Adaptive Strassen is optimally progressing when $m(t) > \lg(n)$. This is not a surprise given that, if N is the initial input size, it has a recurrence of the form $T(n) = 7T(n/2) + O(\min\{\lg(N), n^2\})$.

Lemma 10 *`AdaptiveStrassenRecurse` is optimally progressing if*

1. *$\forall t$, $m(t) > \lg(n)$ and*
2. *`AdaptiveStrassenRecurse` is aware of the size of the cache line size B with respect to the progress bound $\rho(m(t)) = (m(t)B)^{\lg(7)/2}$.*

PROOF. We will show that in any square of size $m(t) \times m(t)$, `AdaptiveStrassenRecurse` does $\Omega((mB)^{\lg(7)/2})$ work, as long as $mB < n^2$.

We will assign work to the multiplications at the leaves of the recursion. That is, we count each multiplication operation as making progress in terms of work. The total number of such multiplications is $O(n^{\lg(7)})$.

We will show that `AdaptiveStrassenRecurse` incurs only a constant factor more misses than classic Strassen. Suppose that we reached a level of the recursion where the side length of the matrices $x < \sqrt{mB/10}$, i.e. the problem at this level of the recursion fits in memory.

The length of the list of scans is $\min(\lg(n), x/B)$ words. Therefore, reading in the list incurs at most $\min\left(\lg(n)/B, x/B^2\right)$ cache misses.

The total size of $P[s][i]$, the additional scans that each leaf needs to do, for all $s \in \{x_1 = 0, x_2 = 1, y_1 = 2, y_2 = 3, z_1 = 4, z_2 = 5\}$ for $i < \lg(x)$ is $< 5x^2$, so all of these smaller scans require no extra cache misses. Additionally, the multiplications only require reading in the size of the problem once the problem fits in cache.

We will now show tht the cache misses due to extra scans passed down to each elaf from interleaved scans is at most the size of the cache. The total number of cache misses from scans is bounded by the size of the scans assigned to each child node. Additionally, `AdaptiveStrassenRecurse` may incur one cache miss from each of the levels of P. That is, the number of faults due to interleaving scans is at most

$$\lg(n) + (1/B) \sum_{i=0}^{\lg(n)-\lg(x)} x^2 4^i/(7^i) < 4/3x^2/B + \lg(n).$$

ADAPTIVESTRASSENRECURSESPLIT(N,LEVEL, INPUT, SCANS):
▷Now we want to split the scan into seven parts
▷Specifically come up with 7 lists each haveing about $1/7$ of the total scan work
Let $childScan$ = RETURNSPLITSCANS(scans,B);
▷Also each child needs to do the scans for its sibling
▷These scans represent additions needed to produce matrices
▷for the input to Strassen.
Let $outX = IsPrecompute[level-1][0]$;
Let $outY = IsPrecompute[level-1][1]$;
Let $outZ = IsPrecompute[level-1][2]$;
Let $inX = input[0]$;
Let $inY = input[1]$;
Let $inZ = input[2]$;
▷Each input has to copy or add matrices
PRESCAN(inX, inY, outX, outY, n);
▷We also need the child nodes to write the outputs of the previous multiplications to the output for the
▷parent multiplication following Strassen's equation.
POSTSCAN(childScan, inZ, outZ);
▷Here we have 7 calls to do the recursive tasks
for $i \in [0,6]$ **do**
  ▷Make the call to each child
  Set $\ell = level - 1$;
  Set $si = childScan[i]$;
  Set $inChild = [outX, outY, outZ, \times]$;
  ADAPTIVESTRASSENRECURSE(n/2, ℓ, inChild, si);
▷Switch the active and preComputation pointers so my sibling can use the information
$active = IsActive[level]$;
$IsActive[level] = IsPrecompute[level]$;
$IsPrecompute[level] = IsActive[level]$;

Algorithm 6: AdaptiveStrassenRecurseSplit(n,level, input, scans)

Therefore, the number of cache misses incurred by a problem of size x is at most $(4/3+5)x^2/B + \lg(n)$ when $x < \sqrt{mB/10}$. A full call to `AdaptiveStrassenRecurse` $(x, level)$ does $x^{\lg(7)}$ work.

Suppose we reached an $m(t) \times m(t)$ square in which at least half of the square requires cache misses for `AdaptiveStrassenRecurse`. We can compute at least 1 call to `AdaptiveStrassenRecurse` $(x, level, input, scans)$ where $(mB - \lg(n))/(6 \times 2 \times 2) < x < (mB - \lg(n))/(6 \times 2)$ in that square. Thus, if $mB > 4\lg(n)B$ then `AdaptiveStrassenRecurse` completes at least $(mB)^{\lg(7)/2}/24 = \Omega((mB)^{\lg(7)/2})$ work.

Note that every $m(t) \times m(t)$ square in the profile must either be at least half scans or at least half calls to `AdaptiveStrassenRecurse`. Therefore, `AdaptiveStrassenRecurse` is optimally progressing in every square. □

Next, we show that the linear scans at the beginning and end of `AdaptiveStrassen` do not preclude adaptivity via a potential argument.

Lemma 11 *Let M be a square profile. For all t, `AdaptiveStrassen` completes at least $\Omega((m(t)B)^{\lg(7)/2-1}m(t)B)$ work plus potential on each $m(t) \times m(t)$ square.*

Furthermore, if `AdaptiveStrassen` completed all the work plus potential as per an intial assignment of work and potential, then it must have completed its last access.

PROOF. First we will consider the time that initial scans and end scans take, or the work in `AdaptiveStrassen` excluding the work of `AdaptiveStrassenRecurse`.

Since the pre and post-scans require allocatin sextra space, we first compute how long these allocations take. The size of the array P of scans is

$$\sum_{i=0}^{\lg(n)} \sum_{j=0}^{\lg(n)-i} \frac{n^2}{4^i 4^j} < \sum_{i=0}^{\lg(n)} \frac{4n^2}{3 \cdot 4^i} < \frac{16n^2}{9}.$$

Therefore, the allocation must do a scan of length $16/9n^2$. We start with a pre-scan of length $\sum_{i=0}^{\lg(n)} n^2/(4^i) < 4/3n^2$ and end with a post-scan of the same length.

We will assign progress to these scans such that the total potential is $O(n^{\lg(7)})$. We assign $n^{\lg(7)-2}$ potential to each operation of the pre and post-scans. The total potential assigned to the scans is $n^2 n^{\lg(7)-2} = O(n^{\lg(7)})$. Thus, in an $m(t) \times m(t)$ square we complete $\Omega(n^{\lg(7)-2} m(t) B)$ progress. Note that $n^{\lg(7)-2} mB = \Omega((mB)^{\lg(7)/2-1} mB)$ as long as $mB < n^2$. If $mB > n^2$, `AdaptiveStrassen` could just have completed all of its work in one square with augmentation. □

Finally we want to show that we don't introduce any computational overhead.

Lemma 12 *`AdaptiveStrassen` takes $O(n^{\lg(7)})$ time in the word-RAM model.*

PROOF. The running time for the pre and post-scans is $O(n^2)$.

Let N be the initial input size. The recurrence for the runtime of `AdaptiveStrassenRecurse` is $T(n) = 7T(n/2) + \min(\lg(N), n^2)$. Therefore, $T(n) = 7T(n/2) + O(n^2) = O(n^{\lg(7)})$. □

B Sorting Experiments

We compared the cache performance of different sorting algorithms from the standard template library following STL for XXL datasets (`STXXL`) [15] with three different sorting algorithms in Figure 6. In order to measure performance with memory changes, we first chose an initial memory size M and ran each algorithm while changing the memory size in the range $[100MB, 2\text{M}]$ every second. We used Linux cgroups to control the memory available to each algorithm.

The three sorting algorithms from `STXXL` are as follows.

1. `std::sort` from the `C++` standard library (`libstdc++`), which implements introspective sort (introsort), a hybrid sorting algorithm which uses quick sort until a maximum recursion depth, at which point it switches to heap sort [27**?**].

2. `stxxl::sort` from `STXXL`. The library implements an asynchronous variant on standard k-way merge sort as described in [16].

3. Cache-oblivious funnel sort implemented in [28].

The sorting algorithms have different structures, so we measured the performance of each algorithm on profiles independent of algorithm structure. The performance of the "more adaptive" sorting algorithms is therefore not a result of friendlier profiles but because the profiles are independent of the algorithm structure. In practice, profiles are often not tied to algorithm structure (e.g. fluctuations based on other parallel computations), so is meaningful to compare the algorithms over randomized profiles.

Sorting algorithms that have better cache-adaptive guarantees incurred relatively fewer faults during a random profile. Specifically, `std::sort` incurred relatively more faults than both funnel sort and

`stxxl::sort`, Funnel sort and `stxxl::sort` are closer to adaptivity than `std::sort`, so they incur fewer faults when the size of memory changes. A possible contributing factor to the difference between the observed adaptivity of the experiments is that funnel sort at `stxxl::sort` are engineered for external-memory computations, while `std::sort` is not.

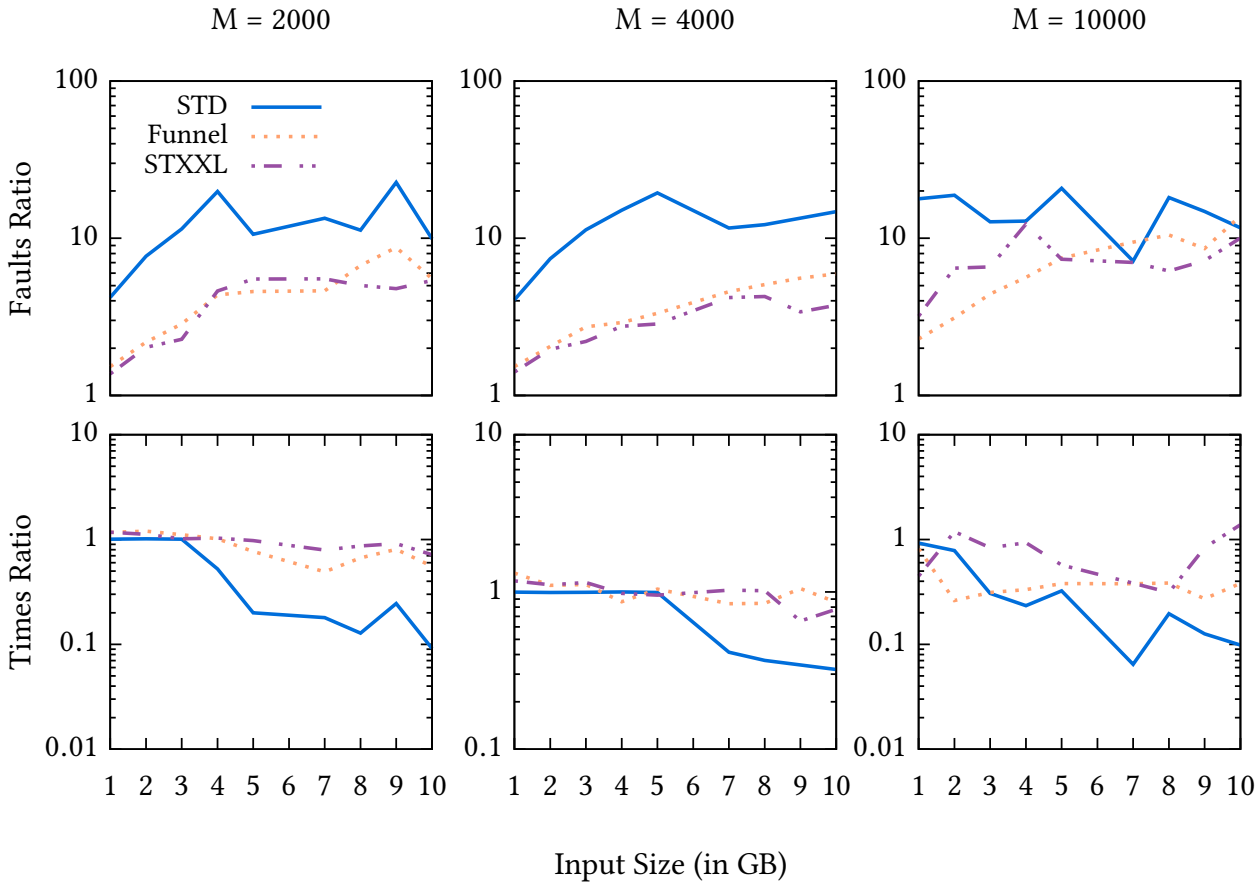


Figure 6: Each point on the plot represents the ratio of the faults incurred during a random profile to the faults incurred on the same input with a fixed, unchanging profile. In the third and fourth plots, we show the faults incurred during different sorting algorithms on changing memory profiles. Each of these two plots represents a different starting memory M in Megabytes. The random profile changes the memory to anywhere in the range $[100, \mathrm{M}]$ Megabytes each second. We normalize the faults incurred during the random profile on a certain input against the faults incurred when the available memory is fixed at M at the beginning of execution.

References

[3]]gmp-strassen [n. d.]. GMP FFT Multiplication. https://gmplib.org/manual/FFT-Multiplication.html. Accessed: 2018-02-01.

[3]]java-strassen [n. d.]. JAVA BigInteger Class. https://raw.githubusercontent.com/tbuktu/bigint/master/src/main/java/java/math/BigInteger.java. Accessed: 2018-02-01.

[3]]std-sort [n. d.]. libstdc++: stl_algo.h Source File. https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01347.html. Accessed: 2018-02-13.

[4] Alok Aggarwal and Jeffrey S. Vitter. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31, 9 (Sept. 1988), 1116–1127.

[5] Ilya Baran, Erik D Demaine, and Mihai Pătraşcu. 2008. Subquadratic algorithms for 3SUM. *Algorithmica* 50, 4 (2008), 584–596.

[6] Rakesh D. Barve and Jeffrey S. Vitter. 1998. *External Memory Algorithms with Dynamically Changing Memory Allocations*. Technical Report. Duke University.

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

[8] Laszlo A. Belady. 1966. A Study of Replacement Algorithms for a Virtual-storage Computer. *IBM Journal of Research and Development* 5, 2 (June 1966), 78–101.

[9] 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 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*. 135–144. https://doi.org/10.1145/2935764.2935798

[10] 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.

[11] Gerth S. Brodal and Rolf Fagerberg. 2002. Cache Oblivious Distribution Sweeping. In *Proc. 29th International Colloquium on Automata, Languages and Programming (ICALP)*. Springer-Verlag, 426–438.

[12] Kurt P. Brown, Michael J. 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.

[13] Don Coppersmith and Shmuel Winograd. 1987. Matrix Multiplication via Arithmetic Progressions. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing* (New York, New York, USA) *(STOC '87)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/28395.28396

[14] Erik D. Demaine. 2002. Cache-Oblivious Algorithms and Data Structures. *Lecture Notes from the EEF Summer School on Massive Data Sets* 8, 4 (2002), 1–249.

[15] Roman Dementiev, Lutz Kettner, and Peter Sanders. 2008. STXXL: Standard Template Library for XXL Data Sets. *Software: Practice and Experience* 38, 6 (2008), 589–637.

[16] Roman Dementiev and Peter Sanders. 2003. Asynchronous Parallel Disk Sorting. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 138–148.

[17] 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

[18] 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.

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

[20] François Le Gall. 2014. Powers of tensors and fast matrix multiplication. In *International Symposium on Symbolic and Algebraic Computation, ISSAC 2014, Kobe, Japan, July 23-25, 2014*. 296–303.

[21] Pieter Ghysels and Wim Vanroose. 2014. Hiding Global Synchronization Latency in the Preconditioned Conjugate Gradient Algorithm. *Parallel Comput.* 40, 7 (2014), 224 – 238. https://doi.org/10.1016/j.parco.2013.06.001 7th Workshop on Parallel Matrix Algorithms and Applications.

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

[23] Elaye Karstadt and Oded Schwartz. 2017. Matrix Multiplication, a Little Faster. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2017, Washington DC, USA, July 24-26, 2017*. 101–110. https://doi.org/10.1145/3087556.3087579

[24] Richard T. Mills. 2004. *Dynamic Adaptation to CPU and Memory Load in Scientific Applications*. Ph. D. Dissertation. The College of William and Mary.

[25] 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.

[26] Richard T. Mills, Chuan Yue, Andreas Stathopoulos, and Dimitrios S. Nikolopoulos. 2007. Runtime and Programming Support for Memory Adaptation in Scientific Applications via Local Disk and Remote Memory. *Journal of Grid Computing* 5, 2 (01 Jun 2007), 213–234. https://doi.org/10.1007/s10723-007-9075-7

[27] David R. Musser. 1997. Introspective sorting and selection algorithms. (1997).

[28] Jesper Holm Olsen and Søren Christian Skov. 2002. Cache-oblivious algorithms in practice. (2002).

[29] 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.

[30] 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.

[31] Daniel D. Sleator and Robert E. Tarjan. 1985. Amortized Efficiency of List Update and Paging Rules. *Commun. ACM* 28, 2 (February 1985), 202–208.

[32] Volker Strassen. 1969. Gaussian Elimination is Not Optimal. *Numer. Math.* 13, 4 (Aug. 1969), 354–356. https://doi.org/10.1007/BF02165411

[33] Volker Strumpen and Thomas L. Casavant. 1994. Exploiting Communication Latency Hiding for Parallel Network Computing: Model and Analysis. In *Proceedings of 1994 International Conference on Parallel and Distributed Systems*. 622–627. https://doi.org/10.1109/ICPADS.1994.590409

[34] Virginia Vassilevska Williams. 2012. Multiplying matrices faster than Coppersmith-Winograd. In *STOC*. 887–898.

[35] Jeffrey S. Vitter. 2001. External Memory Algorithms and Data Structures: Dealing With Massive Data. *Comput. Surveys* 33, 2 (2001).

[36] Jeffrey S. Vitter. 2006. Algorithms and Data Structures for External Memory. *Foundations and Trends in Theoretical Computer Science* 2, 4 (2006), 305–474.

[37] 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.

[38] 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–.

[39] 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.