When Are Cache-Oblivious Algorithms Cache Adaptive? A Case Study of Matrix Multiplication and Sorting

Arghya Bhattacharya
Stony Brook University, NY, USA

Abiyaz Chowdhury
Stony Brook University, NY, USA

Helen Xu
Lawrence Berkeley National Laboratory, CA, USA

Rathish Das
University of Waterloo, Canada

Rezaul A. Chowdhury
Stony Brook University, NY, USA

Rob Johnson
VMware Research, Palo Alto, CA, USA

Rishab Nithyanand
The University of Iowa, Iowa City, IA, USA

Michael A. Bender
Stony Brook University, NY, USA

Abstract

Cache-adaptive algorithms are a class of algorithms that achieve optimal utilization of dynamically changing memory. These memory fluctuations are the norm in today's multi-threaded shared-memory machines and time-sharing caches.

Bender et al. [8] proved that many cache-oblivious algorithms are optimally cache-adaptive, but that some cache-oblivious algorithms can be relatively far from optimally cache-adaptive on worst-case memory fluctuations. This worst-case gap between cache obliviousness and cache adaptivity depends on a highly-structured, adversarial memory profile. Existing cache-adaptive analysis does not predict the relative performance of cache-oblivious and cache-adaptive algorithms on non-adversarial profiles. Does the worst-case gap appear in practice, or is it an artifact of an unrealistically powerful adversary?

This paper sheds light on the question of whether cache-oblivious algorithms can effectively adapt to realistically fluctuating memory sizes; the paper focuses on matrix multiplication and sorting. The two matrix-multiplication algorithms in this paper are canonical examples of "(a, b, c)-regular" cache-oblivious algorithms, which underlie much of the existing theory on cache-adaptivity. Both algorithms have the same asymptotic I/O performance when the memory size remains fixed, but one is optimally cache-adaptive, and the other is not. In our experiments, we generate both adversarial and non-adversarial memory workloads. The performance gap between the algorithms for matrix multiplication grows with problem size (up to 3.8×) on the adversarial profiles, but the gap does not grow with problem size (stays at 2×) on non-adversarial profiles. The sorting algorithms in this paper are not "(a, b, c)-regular," but they have been well-studied in the classical external-memory model when the memory size does not fluctuate. The relative performance of a non-oblivious (cache-aware) sorting algorithm degrades with the problem size: it incurs up to 6× the number of disk I/Os compared to an oblivious adaptive algorithm on both adversarial and non-adversarial profiles.

To summarize, in all our experiments, the cache-oblivious matrix-multiplication and sorting algorithms that we tested empirically adapt well to memory fluctuations. We conjecture that cache-obliviousness will empirically help achieve adaptivity for other problems with similar structures.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms

Keywords and phrases Cache-adaptive algorithms, cache-oblivious algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2022.16

Supplementary Material *Software*: `https://github.com/ArghyaB118/cache_adaptivity`

Funding This research in part was funded by NSF grant CCF-1617618, CCF-1439084, CCF-1725543, the Canada Research Chairs Program, NSERC Discovery Grants, NSF grant CNS-1553510.
This research is funded in part by the Advanced Scientific Computing Research (ASCR) program

within the Office of Science of the DOE under contract number DE-AC02-05CH11231, the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.
This research was sponsored in part by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. 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.

1 Introduction

Applications running on multi-threaded and multi-core systems often experience fluctuations in the amount of physical memory available. For example, when concurrently running programs share RAM, the amount of RAM allocated to any particular program can change as other programs start or finish. Interestingly, even when identical copies of the same program run concurrently, these copies may acquire an unequal fraction of the RAM [20], resulting in further unpredictability.

These memory fluctuations can cause an application's performance to suffer [24, 26]. If the available memory becomes scarce, an application may **thrash**, which means that the application pages excessively, crippling performance [18, 19, 28]. If memory becomes abundant, the program may not take advantage of the larger available RAM and thus perform extraneous I/Os.

In this paper, we empirically show that some optimal cache-oblivious algorithms [21,22,31] – specifically matrix multiplication and sorting – adapt gracefully to most memory fluctuations, except for some adversarially constructed worst-case instances. Optimal **cache-oblivious** algorithms are asymptotically optimal for all fixed memory sizes without the knowledge of the memory size. Past theoretical work shows that optimal cache-oblivious algorithms can perform poorly on adversarially constructed memory fluctuations [7, 8].

Experimental approaches to adaptivity. Past experimental work [29, 30, 34, 35] focused on developing improved algorithms that adapt to a variety of memory profiles, where a **memory profile** is a curve representing the memory size as a function of time. For example, Pang et al. [29] introduced a memory-adaptive merge sort that dynamically adjusts the size of each subproblem at each level of the recursion as memory fluctuates. A similar technique is used by Pang et al. [30] to modify the GRACE hash join algorithm [23] so that it can adapt to changes in memory size. Zhang and Larson [34, 35] introduced techniques that balance the memory usage among many sorting programs running concurrently on a single machine sharing its memory.

Researchers have also explored environment-level modifications to adapt to dynamic memory profiles. For example, Brown et al. [12] introduced memory-management techniques for a DBMS that accommodate multiple concurrently running database workloads, where each workload may have its own individual memory requirements. Mills et al. [24–26] proposed a user interface that enables a user to request sufficient memory for their application.

Theoretical approaches to adaptivity. Barve and Vitter [2, 3] designed algorithms that have provable performance guarantees even when the memory size changes. They gave memory-adaptive algorithms for sorting, matrix multiplication, FFT, LU decomposition,

and permutation. To prove optimality, they extended the traditional **external-memory model** [1] – also called the **disk-access machine (DAM)** – to allow for changes in the memory size over time.

More recently, Bender et al. [8] showed that many cache-oblivious algorithms could adapt well to memory fluctuations. A **cache-oblivious algorithm** is a platform-independent algorithm that is not parameterized by properties of the memory hierarchy such as RAM or cache-line size [21, 22, 31]. An optimal cache-oblivious algorithm is universal in the sense that the algorithm is optimal for any value of the cache parameters, provided that these values do not change over time. (Past theoretical work on cache-oblivious algorithms use the terms memory, cache, and RAM interchangeably) [4, 5]. On the other hand, a cache-adaptive algorithm remains cache-optimal even when memory size changes over time. One possible way to deal with memory fluctuations is to start with an algorithm already known to be cache-oblivious and hope it also happens to be cache adaptive.

Cache-obliviousness and cache-adaptivity. Bender et al. [8] showed that cache-oblivious algorithms are often (but not always) cache adaptive. For example, they showed that Lazy Funnel Sort (LFS) [10, 21], a cache-oblivious sorting algorithm, is optimally cache-adaptive. Follow-up work [7] provided an analytical framework for determining whether cache-oblivious algorithms having a certain recursive form ("(a, b, c)-regular") are also cache adaptive. Algorithms with (a, b, c)-regular recursive structure have a common form of divide-and-conquer cache-oblivious design [6, 7]; see Section 2 for more details. Depending on the settings of the parameters a, b, and c, these cache-oblivious algorithms are either asymptotically optimal or a logarithmic factor away from optimal. For example, cache-oblivious algorithms exist for matrix multiplication [21, 22], some of which are optimally cache-adaptive (MM-INPLACE) and some are a log factor away from optimally cache-adaptive (MM-SCAN) [7, 8]. MM-SCAN ($a = 8$, $b = 4$, and $c = 1$) performs an out-of-place matrix addition at the end of each recursive call and is suboptimal in the cache-adaptive model, while MM-INPLACE ($a = 8$, $b = 4$, and $c < 1$) performs the additions in-place and is optimally cache-adaptive; see Section 3.

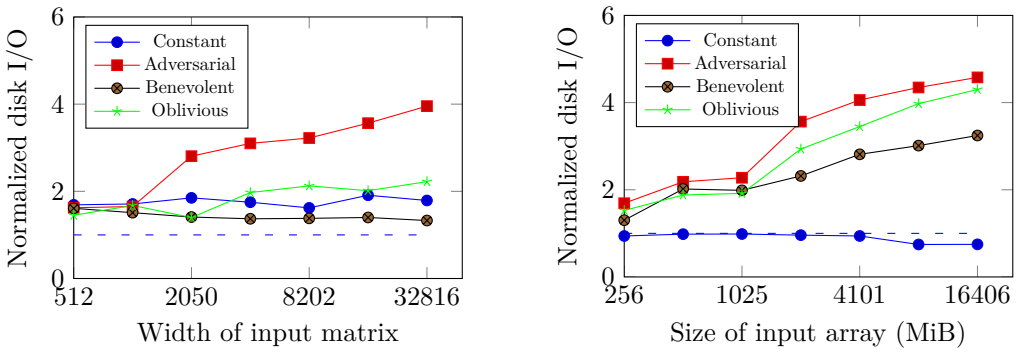
Connections between experimental and theoretical approaches. Existing theoretical approaches to cache adaptivity provide worst-case guarantees but leave open the question of how algorithms perform under arbitrary memory workloads. Specifically, past work showed that cache-oblivious (a, b, c)-regular algorithms are always at most a log factor away from being optimally cache-adaptive. However, this log factor is based on a worst-case memory workload that seems brittle and unlikely to appear in practice. Furthermore, this worst-case analysis does not capture how cache-oblivious algorithms actually perform under practical memory workloads.

In contrast, empirical solutions [12,23–26,29,30,34,35] are designed to improve performance under practical memory workloads. However, they lack worst-case analysis and therefore do not have worst-case performance guarantees.

Recent work [6] has gone beyond worst-case analysis. The authors apply smoothing techniques [32] on memory profiles and explore how well cache-oblivious algorithms adapt to smoothed memory profiles. They observe that the I/O-performance gap between cache-obliviousness and cache-adaptivity disappears given sufficient smoothing. Their results leave open the question of whether the performance gap between cache-oblivious and cache-adaptive algorithms persists across a larger space of memory profiles. If the performance gap turns out to be only a theoretical artifact rarely seen in practice, cache-oblivious algorithms

could be an exciting way to solve a wide range of problems with no known cache-adaptive solutions. Examples of such problems include Gaussian elimination [15], triangle counting [9], min-weight cycle [33], negative triangle detection and counting [33], replacement paths problem [33], etc.

Do these worst-case memory workloads arise naturally when no adversary controls the available memory? How feasible is it to construct such worst-case memory profiles?



(a) Normalized I/O of MM-SCAN to MM-INPLACE.

(b) Normalized I/O of EMS to LFS.

Figure 1 Normalized disk I/O of the non-adaptive algorithm to the cache-adaptive algorithm as a function of input size for matrix multiplication and sorting for four memory profiles: adversarial (worst-case), benevolent, constant, and oblivious. An increase in the y-axis means that the non-adaptive algorithm performs progressively worse relative to the adaptive algorithm.

Results

We empirically study how cubic-time matrix multiplication (MM) algorithms and external-memory sorting algorithms adapt to a range of different memory profiles. MM is a particularly interesting test case because not only is it the canonical example of an (a, b, c)-regular algorithm but also we can find a pair of MM algorithms, both with the same I/O-complexity in the cache-oblivious model, where one is optimally cache-adaptive and the other is a logarithmic factor away from optimal. For the case of MM, we evaluate MM-INPLACE and MM-SCAN and find that both can adapt to a wide range of memory profiles even though MM-SCAN is not provably cache-adaptive. In contrast, for the case of sorting, we evaluate LFS [10, 21] and External-memory Merge Sort (EMS) [1], a non-oblivious and non-adaptive algorithm, and find that EMS does not adapt well to a wide range of memory fluctuations. Table 1 illustrates some of the key properties of the tested algorithms.

Since the worst-case profile is tightly coupled with an algorithm's structure, we designed a **memory profile generator** that can simulate an adversarial by looking into a program's execution. It can generate the worst-case **adversarial memory profile** by increasing the available memory when the program cannot benefit from the extra memory and decreasing the memory when the program would benefit. It can also generate non-adversarial profiles that follow a program's execution.

First, we empirically study the MM-SCAN and MM-INPLACE algorithms for matrix multiplication, and we find that even though MM-SCAN is not optimally cache-adaptive, it adapts well to a wide range of memory fluctuations except the most adversarially constructed profiles. We measure how well an algorithm adapts to the memory fluctuations by measuring

Table 1 Properties of the algorithms for cubic-time matrix multiplication and external-memory sorting studied in this paper.

Algorithm	*Cache-oblivious*	*Cache-adaptive*
MM-SCAN [21, 22]	✓	✗
MM-INPLACE [21, 22]	✓	✓
EM Merge Sort [1]	✗	✗
Lazy Funnel Sort [10]	✓	✓

the disk I/Os it incurs during its execution. MM-SCAN and MM-INPLACE are both theoretically optimal when the memory does not fluctuate; empirically, Figure 1a shows that MM-INPLACE performs roughly 1.8× better than MM-SCAN across all problem sizes under a **constant memory profile** (fixed memory size). We used the memory profile generator to create profiles coupled with algorithm execution. Under the adversarial memory workload, MM-SCAN performs 1.6 – 3.8× more disk I/Os than MM-INPLACE. Critically, the performance gap grows with the input size. We also created a **benevolent memory profile** that is tightly synchronized with the algorithm's execution but non-adversarial in nature. Under the benevolent memory profile, MM-SCAN and MM-INPLACE perform roughly within 1.5× of each other for all problem sizes. Under **oblivious memory profiles**, i.e., profiles that are not tightly coupled with the algorithms' execution, we found that MM-SCAN incurs about 1.8× more I/Os than MM-INPLACE for all problem sizes. These results suggest that MM-SCAN possesses almost all the benefits of cache-adaptivity.

Next, we evaluate the cache-oblivious Lazy Funnel Sort (LFS) [10, 21] and the non-oblivious External-memory Merge Sort (EMS) [1] (both are I/O-optimal when the memory is fixed), and we find that even though LFS has a computational overhead over EMS stemming from its cache-obliviousness; it outperforms EMS on a wide range of fluctuating memory profiles. Figure 1b suggests that External-memory Merge Sort has better I/O-performance than Lazy Funnel Sort for all problem sizes under the constant memory profile (roughly incurring 0.9× disk I/Os on average). On the other hand, LFS adapts when the memory fluctuates, whereas EMS fails to adapt. EMS performs progressively worse than LFS as the input size increases under all types of fluctuating memory workloads (adversarial, benevolent, and oblivious), incurring 1.3 – 7× more disk I/Os. Despite the computational overhead of cache-obliviousness, LFS performs significantly better than EMS under a range of fluctuating memory profiles.

These results suggest that for the problems of MM and sorting, cache-oblivious but non-adaptive algorithms adapt well, while non-oblivious algorithms do not. The tested cache-oblivious algorithms adapt well because they are agnostic to the cache size, while the cache-aware algorithms are optimized for specific cache sizes. We conjecture that cache-oblivious algorithms have most of the empirical benefits of cache adaptivity, and our results are consistent with the conjecture.

Paper overview. The rest of this paper is organized as follows.Section 2 reviews preliminaries about the cache-adaptive model and analysis necessary to understand the experimental design in this paper. Section 3 explains the memory profiles that we tested on. Section 4 describes the experimental setup and the algorithms' performance with fixed memory. Section 5 describes the memory-profile generator and the algorithms' performance under adaptively constructed memory profiles. Section 6 explores the algorithms' performance under obliviously constructed memory profiles. Section 7 provides concluding remarks and future directions.

2 Cache-adaptive analysis

This section reviews fundamentals of cache-adaptive and cache-oblivious analysis. It also explains how (a, b, c)-regular algorithms play a key role in the analysis.

Cache-adaptive model. The cache-adaptive model is an extension of the disk-access machine (DAM) model [1], where the size of memory available to an algorithm can change. In the DAM model, the machine has a two-level memory hierarchy comprising a cache/memory of size M and a disk of unbounded size. Data is transferred between disk and memory in blocks of size B, called I/Os. The cache-adaptive model extends the traditional DAM model by allowing the memory size to be a function of time. Each I/O takes one time step and computation is modeled as free and instantaneous. At time t, the memory available to a program is $M(t)$, which can change after each time step.

Cache-oblivious algorithms and (a, b, c)-regularity. An algorithm is cache-oblivious [21, 22, 31] if it is not parameterized by M and B. An optimal cache-oblivious algorithm is universal, in the sense that it runs optimally in the DAM model for all possible (fixed) values of M and B.

Cache-oblivious algorithms with a particular kind of divide-and-conquer structure, are said to be (a, b, c)**-regular**. An algorithm is (a, b, c)-regular for constants $a \geq 1$, $b > 1$, and $0 \leq c \leq 1$, if, for problem size N, its I/O complexity satisfies the following recurrence: $Q(N) = a \cdot Q(N/b) + \Theta(1 + N^c/B)$.

Specifically, the algorithm has a recursive calls on sub-problems of size N/b and $\Theta(1)$ **linear/sequential scans** before, between, or after the recursive calls, where the size of the largest scan is $\Theta(N^c)$.

The worst-case performance gap between obliviousness and adaptivity. DAM-optimal (a, b, c)-regular algorithms can be up to an $O(\log N)$-factor away from being optimally cache-adaptive [7]. Specifically, they are suboptimal in the cache-adaptive model when $a \geq b$ and $c \geq 1$ [7]. Intuitively, (a, b, c)-regular algorithms are not optimal when they contain large sequential reads through memory at each level of the recursion.

The worst-case memory profile. The logarithmic gap in I/O-performance of some (a, b, c)-regular algorithms from being optimally cache-adaptive is based on a worst-case memory workload that adaptively mimics the recursive structure of the algorithm and is tightly synchronized with the execution of the algorithm. Specifically, the worst-case profile provides extra memory to the non-adaptive algorithm during the linear scans (when it cannot benefit from the extra memory) and takes away the extra memory at the end of the linear scans (when it would be able to use the extra memory). The worst-case memory profile is constructed in an adaptive manner by following the execution of the non-adaptive algorithm, MM-SCAN in case of matrix multiplication and EMS in case of sorting.

Example: MM-SCAN and MM-INPLACE. To illustrate the parameter choices for (a, b, c)-regular algorithms, Bender et al. [7] compare two cache-oblivious cubic matrix multiplication algorithms: MM-SCAN and MM-INPLACE [16,21,22,31], and show that only MM-INPLACE is optimally cache adaptive, whereas MM-SCAN is a logarithmic-factor away from being optimally cache adaptive. MM-SCAN divides each input matrix into $b = 4$ blocks and perform $a = 8$ recursive multiplications on the blocks. It uses extra space to store an

intermediate matrix and performs all additions at once with a linear scan. The recurrence relation of MM-SCAN for problem size N (to multiply two matrices of size $\sqrt{N} \times \sqrt{N}$) is $Q(N) = 8Q(N/4) + O(1 + N/B)$. Hence, MM-SCAN has $c = 1$. MM-INPLACE has the same recursive structure, but it avoids the linear scan by performing the multiplications and additions in-place. The recurrence for MM-INPLACE has the same a and b as MM-SCAN, but has $c = 0$: $Q(N) = 8Q(N/4) + O(1)$. Both MM-SCAN and MM-INPLACE are optimal cache-oblivious algorithms, having an I/O cost of $O(N^{3/2}/(\sqrt{M}B))$.

In contrast, MM-INPLACE (with $c < 1$) is optimally cache-adaptive while MM-SCAN (with $c = 1$) is not. This example of MM-SCAN and MM-INPLACE exemplifies how changing the values of a, b, and c can determine whether a cache-oblivious algorithm is optimally cache adaptive or not. Here, the answer depends on whether $c < 1$ or $c = 1$.

From the above discussion, MM-SCAN may seem to be a strictly worse algorithm than MM-INPLACE because MM-SCAN is provably not cache adaptive. However, MM-SCAN is more parallelizable than MM-INPLACE [16] as it can achieve 8-way parallelism whereas MM-INPLACE can only achieve 4-way parallelism.

Example: EMS and LFS. Finally, we will review the gap between cache-adaptive and non-adaptive sorting algorithms to see an example of cache-adaptive analysis outside of (a, b, c)-regular algorithms. Both EMS and LFS achieve the disk I/O complexity, $O((N/B)\log_{M/B}(N/B))$ in the external-memory model [1], which is the lower bound for sorting when the cache-parameters remain fixed. However, EMS is not cache-adaptive [7] as it has large linear scans involved and any reduction of memory from the promised amount M during these phases affects the performance of the algorithm heavily. On the other hand, despite not having the same recursive structure as that of the (a, b, c)-regular algorithms, LFS [10, 21] is provably cache-adaptive [8]).

3 Memory profile design

This section presents the high-level design of our evaluation of cache-adaptive and non-adaptive algorithms on a variety of memory profiles for matrix multiplication and external-memory sorting. We test on adaptively constructed memory profiles (e.g. the worst-case profile) as well as "oblivious" memory profiles that are erratic but not adaptively constructed. To perform this evaluation, this study uses a memory profile that does not fluctuate with respect to time. We refer to this as the **constant memory profile** and we use it as a baseline in our experiments.

It is theoretically shown that under **adaptively constructed** worst-case memory workload the performance of two DAM-optimal algorithm diverges. Between two (a, b, c)-regular algorithms, MM-INPLACE is cache adaptive and MM-SCAN is non-adaptive as it is log-factor away from being optimally cache adaptive under this worst-case memory profile. We construct this worst-case **adversarial memory profile** by adapting to the execution of the non-adaptive algorithm and allowing it enough memory to store each recursive sub-problem during its linear scan, and low memory when the algorithm is not performing a linear scan. Since linear scans do not employ any locality of reference, the algorithm only requires $O(1)$ memory; any memory given more than that is not utilized during linear scan. For example, MM-SCAN contains a linear scan at the end of each recursion; in each linear scan of size $N \times N$, memory is increased to $5N^2$ to hold the input and output matrices as well as the intermediate results. However, the cache-adaptive algorithm opportunistically benefits from

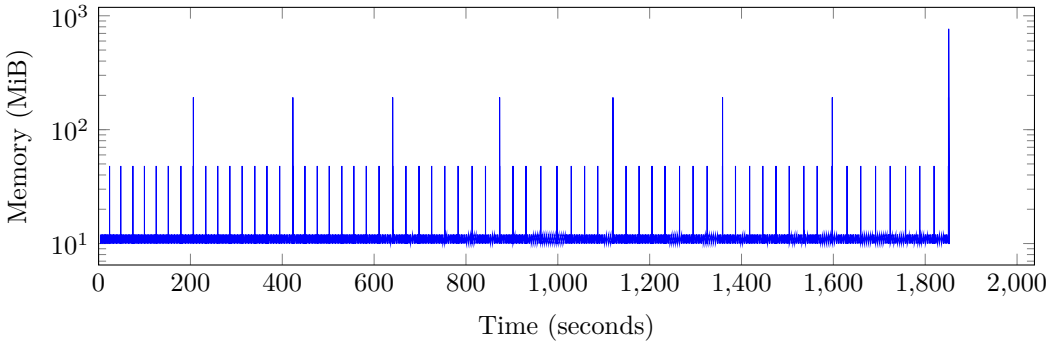


Figure 2 Sample adversarial memory profile generated from running MM-SCAN to multiply two square matrices of width **8192**. The structure of the memory profile mimics the recursive structure of MM-SCAN.

these memory increases, causing its I/O-performance to improve considerably relative to that of the non-adaptive algorithm. Figure 2 shows a sample adversarial memory profile for matrix multiplication.

Similarly, we construct another category of adaptive memory profiles, i.e., the **benevolent memory profile**, but these are designed to be benevolent to the non-adaptive algorithm. The non-adaptive algorithms are unable to use any extra memory that adversarial memory provides during the recursive sequential scans. In the case of the benevolent memory profile, during the sequential scans, instead of increasing the memory we adaptively decrease memory. When the linear scan ends, the memory is again increased to the initial value.

Both the adversarial and benevolent memory profiles are tightly synchronized with the execution of the non-adaptive algorithm. In fact, the adversarial memory profile mimics the recursive structure of the non-adaptive algorithm to bring out the worst performance relative to the adaptive algorithm. In contrast, the benevolent profile aims to minimize the advantage of the cache-adaptive algorithm over the non-adaptive algorithm. However, adversarial and benevolent memory profiles might be too pessimistic or too optimistic in terms of the relative performance of the non-adaptive algorithm and rarely found in the real-world applications. The beyond-worst-case analysis [6] shows the performance gap between MM-SCAN and MM-INPLACE disappears with sufficient smoothing on the worst-case profile, otherwise, the performance gap remains even under smoothed memory profiles. Hence the question remains, how do the non-adaptive algorithms perform under a range of memory profiles that are not generated in an adaptive manner (i.e. the profiles which do not adapt to the execution of the algorithm)?

To gain a more complete view of algorithm performance under memory fluctuations, we run algorithms under **oblivious memory profiles** that are generated non-synthetically and non-adaptively. These profiles are created independently of a given algorithm's execution. A sample oblivious profile can be obtained by running several programs concurrently in shared memory without limiting any of their memory usages [13, 14]. An evaluation under such oblivious workloads accounts for the fact that in practice, often a running process is forced to share available RAM with other concurrent memory-intensive programs. Hence, these oblivious profiles enable us to observe a more complete view of an algorithm's performance under memory fluctuations in a multi-program environment.

4 Evaluation on constant profiles

This section presents details of the experimental setup and studies the performance of the matrix multiplication and external-memory sorting algorithms described in Section 3 on constant memory profiles. All of the tested algorithms are theoretically optimal when the memory does not fluctuate. The empirical results confirm the theory: the performance gap between the cache-adaptive and non-adaptive algorithms does not grow with the problem size when the memory size is fixed.

Experimental setup. All experiments were conducted on a Dell Precision 5820 with an Intel®Core™i9-9900X 3.50GHz processor, two Samsung 16GB M378A2K43DB1-CTD Dual Rank Memory Modules, and Seagate Barracuda ST2000DM001 Desktop SATA Hard Drive. The operating system used is Linux Mint 19.1, Tessa, with kernel version 4.15.0-88. We used C++14 to implement the algorithms, g++ 9.3.0-17ubuntu1 20.04 as compiler, and bash scripts to run the experiments. We measure performance in disk I/Os using */proc*. At the start of each experiment, we use the `sync` system call to free page-cache and slab objects. We use Linux control groups (*cgroups*) to limit the memory available to a program. We run all the experiments with 25 trials and average the measured I/Os.

Algorithm implementation descriptions. We implemented the two algorithms for MM-SCAN and MM-INPLACE directly from their description in past work on cache-adaptivity [7]. Section 2 provides a detailed description of the divide-and-conquer recursion of these two algorithms.

We explored two algorithms for sorting: External-memory Merge Sort (EMS) and Lazy Funnel Sort (LFS). We implemented an (M/B)-way external-memory merge sort that is compatible with the memory profile generator and chose $M = 256$ MiB and $B = 8$ MiB. Previous work [11] on engineering external sorting algorithms demonstrate that carefully engineered cache-aware sorting algorithms can perform up to $2\times$ better than cache-oblivious ones when the memory does not fluctuate. We expect our results regarding relative algorithm performance in the face of memory fluctuations to hold for other implementations of EMS and LFS because of the difference in the algorithm structure between the two algorithms. For LFS, we used an implementation from Olsen and Skov [27].

Problem sizes. We evaluate the relative performance of MM-SCAN to MM-INPLACE on the constant memory workloads as a function of input size. For each matrix multiplication experiment, we multiply two square matrices. We increase the input matrix width from 512 to 32768. For all the experiments, we provide a fixed memory of 10 MiB to the programs.

Similarly, we study the relative performance of EMS to LFS on the constant memory workloads. For each sorting experiment, we sort an integer array. We increase the input array length from 67 million to 1 billion. We provide a fixed memory of 256 MiB to the programs.

Results. Figure 1a shows that when memory size is fixed, cache-adaptive MM-INPLACE performs roughly $1.8\times$ better than non-adaptive MM-SCAN for all input sizes, i.e., their performance gap does not grow with problem size. On the other hand, Figure 1b shows that cache-aware non-adaptive EMS incurs less disk I/Os (roughly $0.9\times$ on average) than cache-adaptive LFS. This result shows the cache-oblivious sorting algorithm does not bear a major computational overhead when the memory does not fluctuate.

5 Evaluation on adaptive profiles

This section presents details of how we generate the memory fluctuations and studies the performance of the matrix multiplication and sorting algorithms described in Section 3 on adaptively-generated memory workloads. Specifically, it first introduces the memory profile generator used to construct the adaptive profiles. Using this profile generator, we explore two types of adaptive memory profiles: the adversarial profile, and the benevolent profile. Under adversarial workloads, the cache-adaptive algorithms perform up to 4.5× better than the non-adaptive algorithms with increasing problem size. On the other hand, the benevolent memory profile demonstrates that oblivious algorithms adapt well to non-adversarial memory fluctuations while non-oblivious algorithms suffer. MM-INPLACE is up to 1.5× better than MM-SCAN, but the gap does not grow with the problem size. In contrast, LFS performs increasingly better (up to 3.2×) than EMS as the problem size grows.

Memory profile generator. We designed a **memory profile generator** that runs a program under a particular memory workload. To fluctuate the memory available to a program, we first used *cgroups*, however *cgroups* does not allow us to decrease the memory available to a program (that the program already claimed) efficiently while the program is running. To address this, the memory profile generator uses *cgroups* to set an initial memory limit available to a program and runs a *balloon* program concurrently with the program within the *cgroups*. The balloon program uses the memory complement to what we intend the program to use. For example, if we want a program to use memory M until time t and thereafter memory $M + \delta$, and the *cgroups* has memory C, the balloon program uses memory $C - M$ until time t and thereafter it uses memory $C - M - \delta$. The initial memory given to a program remains the same as for the constant memory workload.

Using this memory profile generator, we generate two types of profiles, adversarial and benevolent; see Section 3. We measure algorithm performance using the same problem sizes as in Section 4.

Results for the adversarial memory profile

First, let us turn our attention to the adversarial memory workloads. Figure 1 illustrates that under the adversarial memory workloads, the cache-adaptive algorithms perform better than the non-adaptive algorithms, and the gap increases as the problem sizes grow. MM-INPLACE performs roughly up to 4× better than MM-SCAN and LFS performs roughly up to 4.5× better than EMS.

Time spent in linear scan. The performance gap between the cache-adaptive and non-adaptive algorithms grows with the problem size because the relative time spent in scans by the non-adaptive algorithms (MM-SCAN and EMS) grows with problem size. Table 2 shows that the relative time spent in scans grows with problem size for both MM-SCAN (up to 13%) and EMS (up to 76%). Hence, the performance gap also increases between the adaptive and non-adaptive algorithms with problem size. Cache-adaptive algorithms do not perform any such linear scans and take advantage of the extra memory in the adversarial profile. Cache-adaptive algorithms do not perform any such linear scans and take advantage of the extra memory in the adversarial profile.

Table 2 Percentage of running time that the non-adaptive algorithms spend on linear scans.

Algorithm	*Input size*	*% of runtime*
MM-SCAN	1024×1024	0.1
MM-SCAN	2048×2048	4.7
MM-SCAN	4096×4096	13.2
EM Merge Sort	512 MiB	43.6
EM Merge Sort	1 GiB	57.9
EM Merge Sort	2 GiB	76.8

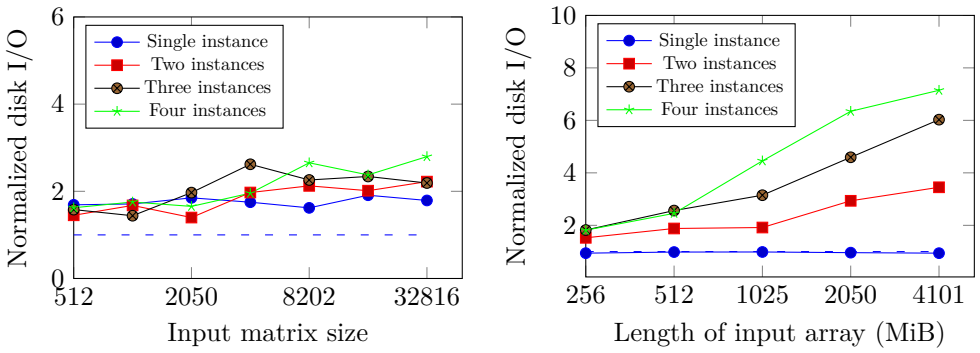
Results for the benevolent memory profile

Next, let us turn our attention to the algorithms' performance under the benevolent memory workloads. Figure 1a shows that MM-SCAN performs very close to MM-INPLACE (between $1.3 \times -1.6\times$) on the benevolent profiles since MM-SCAN is not affected by the memory reductions that occurred during the linear scans whereas MM-INPLACE is affected. On the other hand, Figure 1b shows that the gap between EMS and LFS grows with the problem size – EMS incurs up to $3.2\times$ more I/Os than LFS on the largest input. EMS is unable to adapt to memory reductions during linear scans while LFS adapts to all changes in memory.

6 Evaluation on oblivious profiles

This section explains the setup for the oblivious memory workloads that are not synchronized with the recursive execution of the algorithm and shows that the I/O-performance gap between the two cache-oblivious MM algorithms disappears, but remains between the cache-oblivious and non-oblivious algorithm for sorting. Figure 1a illustrates that MM-SCAN incurs roughly $2\times$ more disk I/Os on average than MM-INPLACE for oblivious memory profiles as we increase the input size. Figure 1b, on the other hand, shows that EMS incurs increasingly worse than LFS with problem size incurring up to $4.3\times$ I/Os on average under oblivious memory profiles. In fact, Figure 3b shows EMS may perform up to $7\times$ more I/Os under oblivious memory workloads.

Experimental setup. This study runs multiple memory-intensive programs concurrently, all sharing the same RAM, to ensure that each program instance experiences memory fluctuations that do not follow the recursive execution of the algorithm. We create these oblivious memory fluctuations in two ways: by creating a **uniform** environment where multiple identically-sized copies of the same program runs concurrently, and by creating a **nonuniform** environment by creating nonuniformity in the concurrent programs. Such nonuniform concurrent programs may include program instances of the same program of different problem sizes, or different programs of same problem sizes. The concurrent programs share a fixed memory that is given at the starting of the experiment. For all the MM experiments, it is 10 MiB, and for all the sorting experiments, it is 256 MiB, if not stated otherwise. We keep the rest of the experimental setup the same as in the case of constant memory profiles (see Section 4).



(a) Normalized disk I/O of MM-SCAN to MM-INPLACE, instances share a memory of 10 MiB.

(b) Normalized disk I/O of EMS to LFS, instances share a memory of 256 MiB.

Figure 3 Average disk I/O of a non-adaptive program to the cache-adaptive program as a function of problem size when running up to 4 instances concurrently normalized to the same when a single instance runs concurrently.

Running uniform instances concurrently

First, let us turn our attention to the oblivious memory workloads that are generated by running multiple concurrent instances of the same program that share a memory [13, 14]. Figure 3a illustrates that cache-oblivious non-adaptive algorithm MM-SCAN performs close to cache-adaptive MM-INPLACE and incurs roughly up to 2× disk I/Os across all problem sizes. On the other hand, Figure 3b shows that the I/O-performance for the cache-aware algorithm EMS rapidly declines relative to the cache-adaptive LFS with increasing problem size (1.4 – 7.1×). Dice et al. [20] showed that multiple threads in shared memory may exhibit a "winner-takes-all" phenomenon where a thread may take memory from the others. However, Figure 3 shows that even if uniformly-sized program instances unevenly share the cache, cache-oblivious algorithms adapt well to memory fluctuations.

Setup. We evaluate the average I/O-performance of the cache-adaptive and non-adaptive algorithms by running concurrent uniform instances, i.e., multiple identically-sized copies of the same program. This multi-program environment setup ensures that each program instance runs under a memory profile that is erratic due to the uncertainty stemming from the concurrent programs' memory requirement but independent of the recursive structure of the particular program under consideration. We perform the study on uniform instances in two ways. First, we vary the input problem sizes given a number of concurrent program instances. Next, we vary the number of concurrent instances in the range of 1 – 4 for a given problem size and observe **slowdown**, the degradation of average I/O-performance of a program instance.

Slowdown. Figure 4 illustrates that cache-oblivious algorithms incur relatively small slowdowns even as the number of concurrent instances increases. When we run up to 4 MM-SCAN instances concurrently, the average disk I/Os stays within 1.2× than the disk I/Os incurred by MM-SCAN when a single concurrent instance runs. The MM-INPLACE instances also face almost no slowdown in a similar environment. In contrast, concurrent EMS instances face up to 6× more disk I/Os when compared to an EMS program that does not share memory with other concurrent programs. However, LFS faces a negligible slowdown in the same scenario.

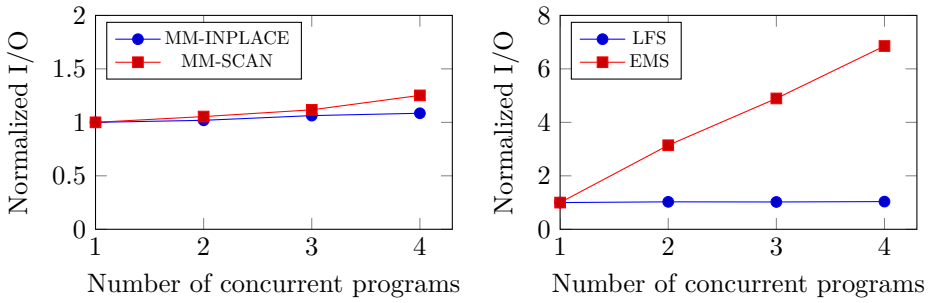
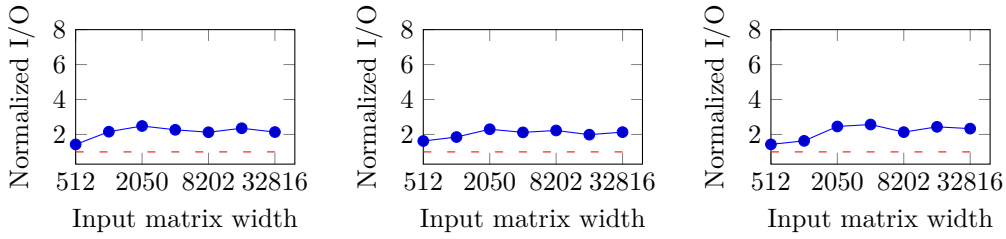


Figure 4 Average normalized disk I/O of a program instance when up to four program instances run concurrently normalized to the disk I/O when a single instance runs.



(a) 3 concurrent instances of the same size.

(b) 3 concurrent instances of different sizes.

(c) 3 concurrent instances each of MM-SCAN and MM-INPLACE.

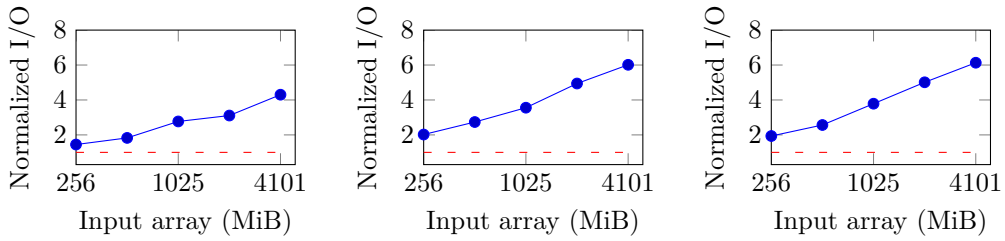
Figure 5 Average normalized disk I/O of non-adaptive MM-SCAN to cache-adaptive MM-INPLACE as a function of problem size with 30 MiB memory given. The y-axis indicates that MM-SCAN performs within a constant factor of MM-INPLACE.

Running nonuniform instances concurrently

Next, let us turn our attention to how the algorithms perform when nonuniform program instances run concurrently and share a fixed memory. The nonuniform programs have different memory requirements at every time step, and therefore are more likely to cause more dramatic memory fluctuations compared to the uniform case. We observe that along with the cache-adaptive algorithms, cache-oblivious MM-SCAN also adapts to the fluctuations caused by nonuniform instances. MM-SCAN incurs up to 2.5× more disk I/Os than MM-INPLACE even when the memory fluctuations are more dramatic in the nonuniform case than in the uniform case, as shown in Figure 5. On the other hand, EMS incurs up to 6× more disk I/Os than LFS, which is more than the gap in the uniform case as demonstrated in Figure 6. Since the memory fluctuations in the nonuniform environment are likely to be more dramatic compared to the uniform environment, the performance gap between EMS and LFS is likely to be more than in the uniform environment because EMS is non-oblivious. Indeed, EMS performs increasingly worse than LFS under the oblivious memory workloads generated in nonuniform environments.

Setup. We create nonuniformity among the program instances in two different manners. In all experiments in Figures 5 and 6, we set the memory size to 30 MiB for MM and to 768 MiB for sorting and retain the problem sizes as mentioned in Section 4.

First, we concurrently run 3 instances of the same algorithm on different input sizes. For each experiment, we select 3 input sizes from the available set of sizes uniformly at random. The algorithm in each instance runs sequentially, but the instances run concurrently with

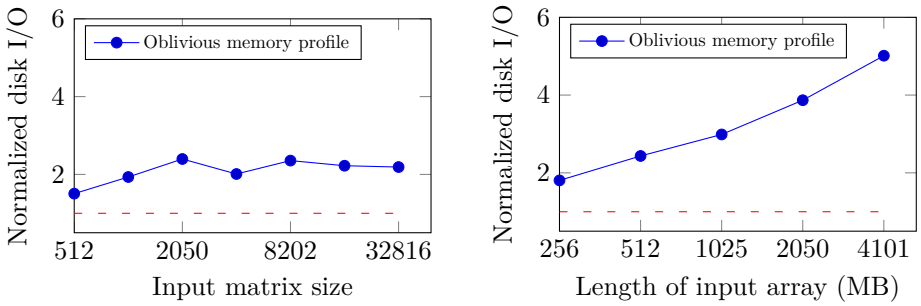


(a) 3 concurrent instances of the same size.

(b) 3 concurrent instances of different sizes.

(c) 3 concurrent instances of each of EMS and LFS.

Figure 6 Average normalized disk I/O of non-adaptive EMS to cache-adaptive LFS as a function of problem size with 768 MiB memory given. An increase in the y-axis indicates that EMS performs progressively worse than LFS.



(a) Normalized disk I/O of MM-SCAN to MM-INPLACE.

(b) Normalized disk I/O of EMS to LFS.

Figure 7 Normalized disk I/O of the non-adaptive to the cache-adaptive algorithm for variable problem size for oblivious memory workloads that we generated by running TPC-C concurrently with the programs. An increase in the y-axis means that the non-adaptive algorithm performs progressively worse relative to the adaptive algorithm.

each other. For each experiment, since the smaller instances may complete earlier than the larger ones, we repeat any instances that have finished until all instances have finished. To measure I/O, we report only the first run of each instance per experiment. As a baseline, we also run 3 uniform program instances to evaluate the effect of nonuniform input sizes.

Second, we concurrently run different algorithms for the same problem on the same input size. Specifically, we run 3 instances of the cache-adaptive algorithm (MM-INPLACE or LFS) and 3 instances of the non-adaptive algorithm (MM-SCAN or EMS) concurrently.

Running a database program concurrently

Finally, we turn our attention to how the MM and sorting algorithms perform when run under oblivious memory workloads that we simulated by running a memory-intensive database program concurrently. The database program is not adversarial and therefore not tied to any particular algorithm structure, so we expect the cache-oblivious algorithms to adapt better to the resulting memory fluctuations. Figure 7 confirms this hypothesis – MM-SCAN performs very close to MM-INPLACE (roughly incurring 2.1× disk I/Os), while EMS performs increasingly worse (up to 6×) than LFS as the problem size grows.

Setup. As the oblivious memory profiles are generated in a non-synthetic manner (see Section 3), and the memory fluctuations are oblivious to what the algorithm does, we choose to run a concurrent memory-intensive database program with the desired algorithm to evaluate the algorithm's performance. We generated such an oblivious memory workload by running the TPC-C [17] workload concurrently with either a cache-adaptive or a non-adaptive program and present the normalized disk I/Os of the non-adaptive algorithms to the adaptive algorithms. TPC-C is a popular online transaction processing (OLTP) benchmark used to measure database management systems. Both the MM and sorting algorithms share the memory with the memory-intensive TPC-C program.

7 Conclusion

We investigated how some cache-oblivious and cache-adaptive (and neither) algorithms perform under a wide range of memory profiles. Specifically, we focused on matrix-multiplication algorithms as representatives of (a, b, c)-regular algorithms and sorting algorithms as representatives of cache-optimized but not (a, b, c)-regular algorithms. We experimentally exhibited the gap in I/O-performance between the cache-adaptive and non-adaptive algorithms for MM and sorting under adversarially-generated memory workloads. To do so, we needed to design a profile generator that can dynamically change the amount of available memory depending on a program's execution. On the other hand, under our oblivious memory workloads, cache-oblivious MM algorithms performed close to each other, whereas the non-oblivious sorting algorithm still performed worse than the cache-oblivious (and adaptive) algorithm. We conjecture that what we have seen in our experiments applies more generally. That is, we conjecture that cache-oblivious programming is a powerful way of empirically achieving cache-adaptivity. This could be important because as mentioned in Section 1, there are many more known cache-oblivious algorithms than known cache-adaptive algorithms. Our results provide hope for the large body of work on cache-oblivious algorithms to empirically perform well even when the cache size fluctuates.

References

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

2 Rakesh D. Barve. *Algorithmic Techniques To Overcome The I/O Bottleneck*. PhD thesis, Duke University, 1998.

3 Rakesh D. Barve and Jeffrey Scott Vitter. A theoretical framework for memory-adaptive algorithms. In *Proc. 40th Annual Symposium on the Foundations of Computer Science (FOCS)*, pages 273–284, New York City, NY, 1999. IEEE.

4 Michael A. Bender, Gerth Stølting Brodal, Rolf Fagerberg, Dongdong Ge, Simai He, Haodong Hu, John Iacono, and Alejandro López-Ortiz. The cost of cache-oblivious searching. In *Proceedings of the 44th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 271–280, Cambridge, MA, 2003. IEEE.

5 Michael A. Bender, Gerth Stølting Brodal, Rolf Fagerberg, Dongdong Ge, Simai He, Haodong Hu, John Iacono, and Alejandro López-Ortiz. The cost of cache-oblivious searching. *Algorithmica*, 61(2):463–505, 2011.

6 Michael A. Bender, Rezaul A. Chowdhury, Rathish Das, Rob Johnson, William Kuszmaul, Andrea Lincoln, Quanquan C Liu, Jayson Lynch, and Helen Xu. Closing the gap between cache-oblivious and cache-adaptive analysis. In *Proc. 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 63–73, virtual event, USA, 2020. ACM.

7 Michael A. Bender, Erik D. Demaine, Roozbeh Ebrahimi, Jeremy T. Fineman, Rob Johnson, Andrea Lincoln, Jayson Lynch, and Samuel McCauley. Cache-adaptive analysis. In *Proc. 28rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 135–144, Asilomar State Beach, CA, 2016. ACM.

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

9 Andreas Björklund, Rasmus Pagh, Virginia Vassilevska Williams, and Uri Zwick. Listing triangles. In *Proc. of 41st International Colloquium on Automata, Languages, and Programming*, pages 223–234, Copenhagen, Denmark, 2014. Springer.

10 Gerth Stølting Brodal and Rolf Fagerberg. Cache oblivious distribution sweeping. In *Proc. 29th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 426–438, Malaga, Spain, 2002. Springer-Verlag.

11 Gerth Stølting Brodal, Rolf Fagerberg, and Kristoffer Vinther. Engineering a cache-oblivious sorting algorithm. *ACM Journal of Experimental Algorithmics*, 12:1–23, 2007.

12 Kurt P Brown, Michael James Carey, and Miron Livny. Managing memory to meet multiclass workload response time goals. In *Proc. of the 19th International Conference on Very Large Data Bases (VLDB)*, pages 328–328, Dublin, Ireland, 1993. IEEE.

13 Rezaul A. Chowdhury, Pramod Ganapathi, Jesmin Jahan Tithi, Yuan Tang, Charles A. Bachmeier, Bradley C Kuszmaul, Charles E. Leiserson, and Armando Solar Lezama. Autogen: Automatic discovery of efficient recursive divide-&-conquer algorithms for solving dynamic programming problems. In *Proc. of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 1–12, Barcelona, Spain, 2016. ACM.

14 Rezaul A. Chowdhury, Pramod Ganapathi, Stephen Tschudi, Jesmin Jahan Tithi, Charles Bachmeier, Charles E. Leiserson, Armando Solar-Lezama, Bradley C. Kuszmaul, and Yuan Tang. Autogen: Automatic discovery of efficient recursive divide-&-conquer algorithms for solving dynamic programming problems. *ACM Transactions on Parallel Computing*, 4(1):1–12, 2017.

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

16 T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, Cambridge, Massachusetts, 2001.

17 Transaction Processing Performance Council. Tpc-c benchmark. Technical Report 5.10.1, TPC, 2009. URL: `http://www.tpc.org/tpcc/`.

18 Peter J. Denning. Thrashing: its causes and prevention. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, AFIPS '68, pages 915–922, San Francisco, CA, 1968. AFIPS.

19 Peter J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, 6(1):64–84, 1980.

20 Dave Dice, Virendra J. Marathe, and Nir Shavit. Brief announcement: Persistent unfairness arising from cache residency imbalance. In *Proc. of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 82–83, Prague, Czech Republic, 2014. ACM.

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

22 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4, 2012.

23 Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Computing*, 1:63–74, 1983.

24 Richard T Mills, Andreas Stathopoulos, and Dimitrios S Nikolopoulos. Adapting to memory pressure from within scientific applications on multiprogrammed cows. In *Proc. 8th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 71–88, Santa Fe, New Mexico, 2004. IEEE.

25 Richard T Mills, Chuan Yue, Andreas Stathopoulos, and Dimitrios S Nikolopoulos. Runtime and programming support for memory adaptation in scientific applications via local disk and remote memory. *Journal of Grid Computing*, 5:213–234, 2007.

26 Richard Tran Mills. *Dynamic adaptation to CPU and memory load in scientific applications*. PhD thesis, The College of William and Mary, 2004.

27 Jesper Holm Olsen, Søren Skov, and Frederik Rønn. Cpp implementations of a cache-oblivious sorting method. Private communication, June 2003.

28 J. Ousterhout. Scheduling techniques for concurrent systems. In *Proc. of the 3rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 22–30, Miami, Florida, 1982. IEEE.

29 HweeHwa Pang, Michael J. Carey, and Miron Livny. Memory-adaptive external sorting. In *Proc. 19th International Conference on Very Large Data Bases (VLDB)*, pages 618–629, Dublin, Ireland, 1993. Morgan Kaufmann.

30 HweeHwa Pang, Michael J Carey, and Miron Livny. Partially preemptible hash joins. In *Proc. 5th ACM SIGMOD International Conference on Management of Data (COMAD)*, pages 59–68, Washington, DC, 1993. ACM.

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

32 Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis: An attempt to explain the behavior of algorithms in practice. *Communications of the ACM*, 52(10):76–84, 2009.

33 Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *Proc. of the 51st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 645–654, Las Vegas, NV, USA, 2010. IEEE.

34 Weiye Zhang and Per-Äke Larson. A memory-adaptive sort (MASORT) for database systems. In *Proc. 6th International Conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, pages 41–54, Toronto, Ontario, Canada, 1996. IBM Press.

35 Weiye Zhang and Per-Äke Larson. Dynamic memory adjustment for external mergesort. In *Proc. 23rd International Conference on Very Large Data Bases (VLDB)*, pages 376–385, Athens, Greece, 1997. Morgan Kaufmann.