Thesis Proposal: Models and Algorithms with Asymmetric Read and Write Costs

Yan Gu

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Guy E. Blelloch, Chair
Phillip B. Gibbons
Anupam Gupta
Jeremy T. Fineman (Georgetown University)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Keywords: Asymmetric Read and Write Costs, Computational Model, Lower Bounds, Non-Volatile Memory, Parallel Algorithms

Abstract

With the arrival of new emerging technologies for main memory, the cost of reading from memory will be significantly cheaper than the cost of writing to memory. This rises new challenges to algorithm design since many of the classic algorithms use approximately the same number of reads and writes. My thesis will study several aspects of this asymmetry between read and write costs. First, it will present and study several cost models that account for this asymmetry in various settings, including both sequential and parallel models. Then the thesis will introduce new algorithms and techniques that are more efficient on the new models compared to standard approaches. To justify whether the approaches and algorithms translate to better efficiency on real hardware, I will attempt to run experiments on upcoming memory systems (this will depend on when the hardware becomes available).

Contents

1 Introduction

Classic research on algorithms has focused on settings in which reads and writes (to the main memory) have similar cost. However, such assumption might not be the reality in the future since we are on the cusp of the emergence of a new wave of nonvolatile memory technologies that are projected to become the dominant type of main memory in the near future [26, 30, 35, 44]. A key property of these new memory technologies (e.g., phase-change memory, spin-torque transfer magnetic RAM, and memristor-based resistive RAM) is their asymmetric read-write costs: ***Writes can be an order of magnitude or more higher energy, higher latency, and lower (per-module) bandwidth than reads*** [3, 5, 11, 12, 16, 21, 22, 27, 29, 33, 38, 42]. Given 50 years of research on models in which memory access is assumed to be symmetric, this high cost inevitably arises the following challenges:

- How should existing computational models be modified to take account for this asymmetry between reads and writes, and how should such memory be modeled?
- How would this asymmetry impact algorithm design?
- What new techniques are useful for trading-off doing more cheaper operations (say more reads) in order to do fewer expensive operations (say fewer writes)?
- What are the fundamental limitations on such trade-offs (lower bounds)?

While several prior works, which are introduced in Section 2, have explored write-efficient algorithms for databases or for the unique properties of NAND Flash, our ongoing work seeks to develop a broader *theory* of algorithm design for asymmetric memories and provides a first step towards answer these fundamental questions. From the model side, my first paper on this topic [13] defined an Asymmetric PRAM model that differs from the classic PRAM in charging ω (an integer and $\omega \gg 1$) for writes (reads are unit cost), as well as a variety of external-memory-style models that transfer data in blocks. The follow-on paper [12] defined the sequential (M, ω)-Asymmetric RAM model that combines a small symmetric-cost memory of size M with a large asymmetric-cost memory. Later, the third paper [7] defined the Asymmetric Nested-Parallel model, which combines features of the sequential parallel model but with a distinctive memory allocation scheme. It is shown in the paper that the model, with its costs analyzed based on the computation DAG (with no notions of processors or scheduling), maps efficiently onto a more concrete machine model, when using a work-stealing scheduler. These models are sufficient to evaluate the cost for both sequential and parallel algorithms while considering the asymmetry between reads and writes.

With these models, a number of algorithmic results are presented in this thesis. For sequential algorithms, the lower and/or upper bounds are discussed on Fast Fourier Transform, sorting networks, comparison sorting, diamond DAG, longest common subsequence, edit distance, search

tree, priority queue, 2D convex hull, Delaunay triangulation, BFS, DFS, topological sort, biconnected components, strongly connected components, single-source/all-pair shortest-paths, and minimum spanning tree. A common theme in many of these algorithms is that they use redundant computations and trade off writes for reads. A more detailed summary of the results are given in Section 4.

This thesis also introduces various parallel algorithms for a number of fundamental problems. For the problems studied, our goal is to reduce the number of writes while preserving work-efficiency and low depth (a.k.a., low span). Reduced write, work efficient, low depth Asymmetric Nested-Parallel algorithms are presented for reduce, list contraction, tree contraction, comparison sorting, FFT, matrix multiplication, breadth-first search, ordered filter, planar convex hull, and minimum spanning tree algorithm. See Section 5 for details of results. All these algorithms significantly reduce the number of writes over the best prior algorithms. These algorithms reveal several interesting techniques for significantly reducing shared memory writes in parallel algorithms without asymptotically increasing the number of shared memory reads.

Although a number of algorithms on a variety of problems are introduced, it is interesting whether there exist some general *techniques* that can improve the performance of types of algorithms, and this might be one of the future (proposed) work. One example is the randomized incremental algorithms. All of these algorithms share a similar framework, and a general solution may reduce writes for all of these algorithms.

The current results stand on the theory side, but it is on the plan to actually run experiments on the algorithms in the future as soon as the new memory system is available. Based on the results, it will be possible to justify the algorithms and techniques that are useful in practice.

Thesis statement: The goal of this thesis is to develop efficient algorithms for asymmetric memory on read and write costs. This includes proposing modified computational models and techniques, and designing specific algorithms and experiment verification.

2 Background and Related Work

Emerging nonvolatile/persistent memory (NVM) technologies such as Phase-Change Memory (PCM), Spin-Torque Transfer Magnetic RAM (STT-RAM), and Memristor-based Resistive RAM offer the promise of significantly lower energy and higher density (bits per area) than DRAM. With byte-addressability and read latencies approaching or improving on DRAM speeds, these NVM technologies are projected to become the dominant memory within the decade [26, 30, 35, 44], as manufacturing improves and costs decrease.

Although these NVMs could be viewed as just a layer in the memory hierarchy that provides persistence, there is one important distinction: Writes are significantly more costly than reads, suffering from higher latency, lower per-chip bandwidth, higher energy costs, and endurance problems (a cell wears out after 10^8–10^{12} writes [35]). Thus, unlike DRAM, there is a significant (often an order of magnitude or more) asymmetry between read and write costs [3, 5, 21, 22,

29, 33, 38, 42], and more technical details are provided in Appendix A. Motivated by these techniques, the study of *write-efficient* (*write-limited*, *write-avoiding*) algorithms, which reduce the number of writes relative to existing algorithms, is of significant and lasting importance.

Related Work. Read-write asymmetry has been the focus of many systems efforts [18, 34, 43, 45]. Reducing the number of writes has long been a goal in disk arrays, distributed systems, cache-coherent multiprocessors, and the like, but that work has not focused on NVMs and the solutions are not suitable for their properties. Several papers [6, 23, 25, 36, 37, 41] have looked at read-write asymmetries in the context of flash memory. However, due to the different physical properties between main memory and flash memory, these results cannot directly apply to the new emerging problems. A few prior works [17, 40, 41] have also looked at algorithms for asymmetric read-write costs in emerging NVMs, in the context of databases. Our proposed work extends far beyond this, providing a systematic study of models, algorithms, and runtime systems for asymmetric read-write costs.

Carson et al.'s work [16] nevertheless is similar to our results. They presented upper and lower bounds for various linear algebra problems and direct N-body methods under asymmetric read and write costs. However, they restricted their focus to the class of "communication-avoiding" algorithms, i.e., parallel algorithms that minimize the (unweighted) sum of reads and writes, instead of the overall cost. Their results are more useful for distributed or external memory setting, while we focus on sequential and share-memory parallel algorithms.

3 Computational Models

The first step towards designing algorithms on asymmetric memory is to propose appropriate machine models that estimate the cost of an algorithm running on an asymmetric memory. In this section, several models are introduced for various settings.

3.1 The Asymmetric RAM Model

Random-access machine (RAM) is an abstract machine in the general class of register machines. The RAM model is the most commonly used model for computational complexity analysis. In the RAM model, any operation and memory access (read or write) takes unit time. This model does not distinguish different levels in the memory hierarchy.

To define a new machine model of the computation on asymmetric memories, two modifications are required: first, a write should cost more than a read; second, a fast-memory should be included since the registers and caches will still be symmetric on read and write costs in the future. In the simplest model we consider, there is an asymmetric random-access memory such that reads cost 1 and writes cost $\omega \gg 1$, as well as a constant number of symmetric "registers" that can be read or written at unit cost. More generally, we consider settings in which the amount of symmetric memory is $M \ll n$, where n is the input size: Thus, we give the definition of

(M, ω)-*Asymmetric RAM (ARAM)*:

Definition 1. *The (M, ω)-Asymmetric RAM (ARAM) is comprised of a symmetric small-memory of size M, an asymmetric large-memory of unbounded size, and an integer write cost ω. The* ARAM cost Q *is the number of reads from large-memory plus ω times the number of writes to large-memory. The* time T *is Q plus the number of reads and writes to small-memory.*

The cost Q, measures the I/O cost of the main memory, which is similar to the I/O cost of the classic external memory model except that the block size are ignored. This cost is useful for I/O bounded algorithms like sorting, FFT, linear time graph algorithms and many data structures. The time T resembles the cost of a RAM model which measures the cost of both I/O and operations, and is used in analyzing computation-intensive algorithms like dynamic programming, graph and geometry algorithms.

3.2 The Asymmetric Nested-Parallel Model

For the purpose of parallelism, this thesis introduces a parallel variant [7] of the (M, ω)-Asymmetric RAM (ARAM) model to analyze algorithms. The goal is to allow for dynamic parallelism and to analyze algorithms using work and depth (also called span or critical path length). The nested-parallel model [9] therefore is used to extended with asymmetric memory. This thesis includes some detail on the model since there are some subtleties on how the small symmetric memory is defined in a dynamically parallel model, and some care was given to the particular formulation we give here. This thesis also describe a more concrete machine model and map costs from the (M, ω)-ARAM model to it.

In the nested-parallel model a computation starts and ends with a single ***root*** task. Each task has a constant number of registers, and runs a standard instruction set from a random access machine (RAM), except it has one additional instruction called FORK. The FORK instruction takes an integer n and creates n ***child*** tasks, which can run in parallel. Child tasks get a copy of the parents register values, with one special register getting an integer from 1 to n indicating which child it is. The parent task suspends until all its children finish[1] at which point it continues with the registers in the same state as when it suspended, except the program counter advanced by one. We say a computation has ***binary branching*** if $n = 2$. In the model a computation can be viewed as a (series-parallel) DAG in the standard way. We assume every instruction has a weight (cost). The ***work*** (W) is the sum of the weights of the instructions, and the ***depth*** (D) is the weight of the heaviest path. The ***nesting depth*** (δ) is the maximum depth of forked tasks during the computation.

In the (M, ω)-ARAM we assume a stack allocated symmetric small memory, and a heap allocated assymetric large memory. ***Stack allocated*** memory is memory allocated by a task, available to the task and its children, but invalid when the task finishes. It is under this model, for example, that the memory bounds for work stealing are shown [15]. ***Heap allocated*** memory is allocated by a task and can be accessed by any other task, including ancestor tasks (it is

[1] We assume, as in the RAM, there is a FINISH instruction.

completely shared memory). Each instruction has weight one, except writes to the heap memory, which have weight $\omega \geq 1$ (in practice, $\omega \gg 1$). In this paper we assume the amount of stack memory allocated by all but the leaf tasks (tasks with no forks) is constant. The amount of symmetric stack memory a leaf task can allocate is bounded by a parameter M_l. This separation into stack and heap allocated memory, and the distinction between leaf and non-leaf tasks for stack memory size, is made both because it is a convenient model for using the different memories, and also because it allows an efficient mapping onto a fixed number of processors, as justified below.

We call an algorithm to be a ***bulk-synchronous*** algorithm if there is only one level of nesting. The root task proceeds in a sequence of R rounds. In each round i it forks n_i child tasks (each a leaf) and waits for them to finish. The root task can run arbitrary computation between such rounds. We refer define the iteration count I as $\sum_{i=1}^{R} n_i$.

To justify the model here we outline a key result for mapping costs in the model onto a more concrete target machine. For the target machine we adapt the PRAM model as follows. We assume P processors, each a RAM running its own instructions with a small symmetric ***local*** memory of size M. The processors share an unbounded asymmetric ***global*** memory, to which concurrent reads and writes are allowed. We also allow any processor to read the local memory of another processor (concurrently), but not to write to it. A request to read the local memory of another processor is viewed as requiring a write out to the global memory in order to enable the read, and hence is charged ω. On each processor any write to the global memory also takes ω time. All other instructions take unit time. For synchronizing we assume an atomic fetch-and-add to the global memory. We refer to this model as the (M, ω)***-Asymmetric PRAM***.

One useful aspect of our model is that although it might seem that every FORK would require at least one write to the global memory (and hence the (M, ω)-ARAM model would need to charge ω for every FORK instead of 1), this is not the case when using a work-stealing scheduler [15], as shown by the following theorem.

Theorem 3.2.1. *A computation with binary branching factor on the (M, ω)-ARAM model with W work, D depth, δ nesting depth, and M_l leaf stack memory, can be simulated on an (M, ω)-Asymmetric PRAM with P processors and $M = O(\delta + M_l)$ in expected $O(W/P + \omega D)$ time.*

The theorem provides justification for charging only unit cost for FORK, and for example, means that the standard Reduce via a binary tree incurs only $\Theta(n + \omega)$ work instead of $\Theta(\omega n)$ work on the (M, ω)-ARAM. Note also that our separate accounting for leaf stack memory in the (M, ω)-ARAM model, and the observation that the non-leaf tasks of all the algorithms in this paper each allocate only $O(1)$ stack memory, means that the bound in the lemma is only a constant number of writes per steal, whereas without the separate accounting, it would be $O(M_l)$ writes per steal.

The following lemma provides additional support for the model.

Lemma 3.2.2. *A bulk-synchronous computation with arbitrary branching on the (M, ω)-ARAM model with W work, D depth, R rounds, I iteration count, and M_l leaf stack memory, can be*

simulated on an (M, ω)-Asymmetric PRAM with P processors and $M = O(M_l)$ in $O((W + \omega I)P + D + \omega R)$ time.

3.3 The Asymmetric External Memory Model and Cache Models

The widely studied External Memory (EM) model [2] (also called I/O model, Disk Model and Disk Access Model) assumes a two level memory hierarchy with a fixed size primary memory (cache) of size M and a secondary memory of unbounded size. Both are partitioned into blocks of size B. Standard RAM instructions can be used within the primary memory, and in addition the model has two special *memory transfer* instructions: a *read* transfers (alternatively, copies) an arbitrary block from secondary memory to primary memory, and a *write* transfers an arbitrary block from primary to secondary memory. The I/O complexity of an algorithm is the total number of memory transfers. The *Asymmetric External Memory* (AEM) model [13] simply adds a parameter ω to the EM model, and charges this for each write of a block. Reading a block still has unit cost.

The Ideal-Cache model [24] is a variant of the EM model. The machine model is still organized in the same way with two memories each partitioned into blocks, but there are no explicit memory transfer instructions. Instead all addressable memory is in the secondary memory, but any subset of up to M/B of the blocks can have a copy resident in the primary memory (cache). Any reference to a resident block is a *cache hit* and is free. Any reference to a word in a block that is not resident is a *cache miss* and requires a memory transfer from the secondary memory. The cache miss can replace a block in the cache with the loaded block, which might require *evicting* a cache block. The I/O or *cache complexity* of an algorithm is the number of cache misses.

The main purpose of the Ideal-Cache model is for the design of *cache-oblivious algorithms*. These are algorithms that do not use the parameters M and B in their design, but for which one can still derive effective bounds on I/O complexity. This has the advantage that the algorithms work well for any cache sizes on any cache hierarchies.

This thesis defines the *Asymmetric Ideal-Cache* model [13] by distinguishing reads from writes, as follows. A cache block is *dirty* if the version in the cache has been modified since it was brought into the cache, and *clean* otherwise. When a cache miss evicts a clean block the cost is 1, but when evicting a dirty block the cost is $1 + \omega$, 1 for the read and ω for the write.

For the classic Ideal-Cache model, an optimal (offline) cache eviction policy is assumed—i.e., one that minimizes the I/O complexity. It is well known that the optimal policy can be approximated using the online least recently used (LRU) policy at a cost of at most doubling the number of misses, and doubling the cache size [39]. For the new model, we again assume an ideal offline cache replacement policy—i.e., minimizing the total I/O cost. Under this model we note that the LRU policy is no longer 2-competitive. However, the following variant is competitive within a constant factor. The idea is to separately maintain two equal-sized pools of blocks in the cache (primary memory), a read pool and a write pool. When reading a location, (i) if its block is in the read pool we just read the value, (ii) if it is in the write pool we copy the block to

the read pool, or (iii) if it is in neither, we read the block from secondary memory into the read pool. In the latter two cases we evict the LRU block from the read pool if it is full, with cost 1. The rules for the write pool are symmetric when writing to a memory location, but the eviction has cost $\omega + 1$ because the block is dirty. We call this the read-write LRU policy. This policy is competitive with the optimal offline policy:

Theorem 3.3.1. *For any sequence S of instructions, if it has cost $Q_I(S)$ on the Asymmetric Ideal-Cache model with cache size M_I, then it will have cost*

$$Q_L(S) \leq \frac{M_L}{(M_L - M_I)} Q_I(S) + (1+\omega) M_I/B$$

on an asymmetric cache with read-write LRU policy and cache sizes (read and write pools) M_L.

With the Ideal-Cache model, a Cache-Oblivious paradigm can be analyzed. Here algorithms are defined as nested parallel computations based on parallel loops, possibly nested. The depth of the computation is the longest chain of dependencies, and the depth of a parallel loop is the maximum of the depths of its iterates. The computation has a natural sequential order by converting each parallel loop to a sequential loop.

Using known scheduling results the depth and sequential cache complexity of a computation are sufficient for deriving bounds on parallel cache complexity. In particular, let D be the depth and Q_1 be the sequential cache complexity. Then for a p-processor shared-memory machine with private caches (each processor has its own cache) using a work-stealing scheduler, the total number of misses Q_p across all processors is at most $Q_1 + O(p\omega DM/B)$ with high probability [1, 11]. For a p-processor shared-memory machine with a shared cache of size $M + pBD$ using a parallel-depth-first (PDF) scheduler, $Q_p \leq Q_1$ [8, 11]. These bounds can be extended to multi-level hierarchies of private or shared caches, respectively [10].

4 Sequential Algorithms on the (M, ω)-ARAM Model

This section presents a number of lower and upper bounds for the (M, ω)-ARAM [12, 13], as summarized in Table 5.1. These results consider a number of fundamental problems and demonstrate how the asymptotic algorithm costs decrease as a function of M, e.g., polynomially, logarithmically, or not at all.

Lower bounds To start with, we show lower bounds on a variety of fundamental problems. For these problems, we show that these problems (or computations) cannot take advantage at all from cheaper reads, unless $\omega > M$, which is rarely to be true.

For FFT we show an $\Omega(\omega(n \log_{\omega M} n))$ lower bound on the ARAM cost, and a matching upper bound. Thus, even allowing for redundant (re)computation of nodes (to save writes), it is only possible to achieve asymptotic improvements with cheaper reads when $\omega > M$. Prior lower bound approaches for FFTs for symmetric memory fail to carry over to asymmetric memory, so a new lower bound technique is required. We use an interesting new accounting argument

Table 4.1: Summary of Our Results for the (M, ω)-ARAM

problem	**ARAM cost $Q(n)$ or $Q(n, m)$**	**ARAM time $T(n)$ or $T(n, m)$**
FFT	$\Theta(\omega n \log n / \log(\omega M))$	$\Theta(Q(n) + n \log n)$
sorting networks	$\Omega(\omega n \log n / \log(\omega M))$	$\Omega(Q(n) + n \log n)$
sorting (comparison)	$O(n(\log n + \omega))$	$\Theta(n(\log n + \omega))$
diamond DAG	$\Theta(n^2\omega/M)$	$\Theta(Q(n) + n^2)$
longest common subsequence, edit distance	$O(n^2\omega/\min(\omega^{1/3}M, M^{3/2}))$	$O(n^2(1 + \omega/\min(\omega^{1/3}M^{2/3}, M^{3/2})))$
search tree, priority queue	$O(\omega + \log n)$ per update	$O(\omega + \log n)$ per update
planar convex hull, triangulation	$O(n(\log n + \omega))$	$\Theta(n(\log n + \omega))$
BFS, DFS, topological sort, biconnected components, SCC	$\Theta(\omega n + m)$	$\Theta(\omega n + m)$
all-pairs shortest-path	$O(n^2(\omega + n/\sqrt{M}))$	$O(Q(n) + n^3)$
single-source shortest path	$O(\min(n(\omega + m/M), \omega(m + n \log n), m(\omega + \log n)))$	$O(Q(n, m) + n \log n)$
minimum spanning tree	$O(m \min(\log n, n/M) + \omega n)$	$O(Q(n, m) + n \log n)$

for fractionally assigning a unit weight for each node of the network to subcomputations that each have cost ωM. The assignment shows that each subcomputation has on average at most $M \log(\omega M)$ weight assigned to it, and hence the total cost across all $\Theta(n \log n)$ nodes yields the lower bound.

For sorting, we show the surprising result that on asymmetric memories, comparison sorting is asymptotically faster than sorting networks. This is in contrast with the RAM model (and parallel models such as the PRAM, I/O models, etc.), in which the asymptotic costs are the same! The lower bound leverages the same key partitioning lemma as in the FFT proof.

We present a tight lower bound for DAG computation on diamond DAGs that shows there is no asymptotic advantage of cheaper reads if we allow no recomputation. The proof is based on a conclusion from Cook and Sethi [19], and even simplifies the proof of the original symmetric setting by Hong and Kong [31].

Upper bound (algorithms) We also present several interesting algorithms that have lower cost and time on the (M, ω)-ARAM comparing to classic approaches. A common theme in many of the upper bounds is that they require redundant computation and a tradeoff between reads and writes.

For the diamond DAG, despite that there exist a tight lower bound if recomputation is prohibited, we show that allowing a vertex to be "partially" computed before all its immediate predecessors have been computed (thereby violating a DAG computation rule), we can beat the lower

Table 5.1: Our Results for the Asymmetric NP Model

problem	**work**	**depth**
Reduce	$\Theta(n+\omega)$	$\Theta(\log n+\omega)$
Ordered filter	$\Theta(n+\omega k)$†	$O(\omega \log n)$†
Sorting	$\Theta(n \log n+n\omega)$†	$O(\omega \log n)$†
List contraction	$\Theta(n)$	$O(\omega \log n)$†
Tree contraction	$\Theta(n)$	$O(\omega \log n)$†
Minimum spanning tree	$O(\alpha(n)m+$ $\omega n \log(\min(\frac{m}{n},\omega)))$†	$O(\omega$ polylog$(m))$†
2D convex hull	$O(n \log k+\omega n \log\log k)$‡	$O(\omega \log^2 n)$†
BFS tree	$\Theta(m+\omega n)$‡	$O(\omega\delta \log n)$†

k=output size; †=with high probability; ‡=expected; α=inverse Ackerman; δ=graph diameter

bound and show asymptotic advantage. Specifically, for both the longest common subsequence and edit distance problems (normally thought of as diamond DAG computations), we devise a new "path sketch" technique that leverages partial aggregation on the DAG vertices. Again we know of no other models in which such techniques are needed.

We also show how to adapt Dijkstra's single-source shortest paths algorithm using phases so that the priority queue is kept in small-memory. The key idea in this algorithm is to run Dijkstra's algorithm while only maintain a priority queue (Fibonacci heap) with limited size, which requires some non-trivial modifications to the priority queue.

We discuss how to adapt Borůvka's minimum spanning tree algorithm to reduce the number of shortcuts and hence writes that are needed.

Lastly, we analyze the cost and time of many basic problems and algorithmic building blocks which only require relatively straight-forward modifications of the algorithms. These problems and building blocks include sorting, search tree, priority queue, 2D convex hull, Delauney triangulation, BFS, DFS, topological sort, biconnected components, strongly connected components, and all-pairs shortest-paths.

5 Parallel Algorithms on the Asymmetric NP Model

This section introduces the parallel Algorithms on the Asymmetric NP Model [7, 13]. For each problem studied, our goal is to reduce the number of writes while preserving work-efficiency and low span (a.k.a., low depth). We present reduced write, work-efficient, low depth Asymmetric NP algorithms for a number of fundamental problems such as reduce, sorting, list contraction, tree contraction, breadth-first search, ordered filter, and planar convex hull. For the latter two problems, our algorithms are output-sensitive in that the work and number of writes decrease

with the output size. We also present a reduced write, low-depth minimum spanning tree algorithm that is nearly work-efficient (off by the inverse Ackermann function). See Table 5.1 for a summary of results. All these algorithms significantly reduce the number of writes over the best prior algorithms. While some of these results were relatively straightforward, our algorithms for tree contraction, sorting, MST, and convex hull are more novel. Our algorithms reveal several interesting techniques for significantly reducing shared memory writes in parallel algorithms without asymptotically increasing the number of shared memory reads.

Sorting Most of the existing parallel sorting algorithms (e.g. mergesort, quicksort, many kinds of sample sort) require $O(n \log n)$ writes. We discussed carefully-tuned version of sample sort that requires $O(n \log n)$ reads and operations, $O(n)$ writes, and has $O(\omega \log n)$ depth w.h.p. The algorithm contains three rounds of sample sort: one randomized sample sort with $n/\log^2 n$ pivots and over-sampling ratio of $\log n$, and two deterministic sample sort with $|A|^{1/3}$ splitters where A is the input of each recursive subproblem.

List and Tree Contraction The tree contraction problem seems to be hard using $o(n)$ writes for both sequential and parallel cases since all of the previous algorithms requires linear number of writes. To solve this problem, we proposed three algorithms as tools to construct a new algorithm. First, we introduced a new parallel list-partitioning algorithm using random sampling, which uses linear work (time T sequentially for the ARAM) and $O(\omega \log n)$ depth. Then we proposed a new tree-partitioning algorithm using the Euler tour of the tree. The main routine is to call the list-partitioning algorithm and thus the work and depth is the same as the list partition. Lastly, we discussed a sequential divide-and-conquer tree-contraction algorithm to solve the base cases using bounded local memory. Combining these three algorithm plus some extra steps, an arbitrary tree with n nodes can be contracted using linear work (time T sequentially for the ARAM) and $O(\omega \log n)$ depth, which seems to be optimal.

Other results We also consider some basic parallel primitives and algorithms [7]. Reduce, i.e. summing a sequence of values with respect to an associative function, can be done in $\Theta(n + \omega)$ work and $\Theta(\omega + \log n)$ depth on the Asymmetric NP model. These bounds may seem impossible since the processors need to communicate. However, recall that when simulating the Asymmetric NP on a machine, the forks and joins are in the stack memory and thus we account for the steals in the cost. We introduced an algorithm of output-sensitive ordered filter, with $\Theta(n + \omega k)$ work and $\Theta(\omega \log n)$ depth, which uses our sorting algorithm as a subroutine. Our other results include minimum spanning tree, which an additional exponential delaying is integrated into Karger, Klein and Tarjan's algorithm [32], BFS, which uses a similar exponential delaying subroutine, and

output-sensitive planar convex hull that requires non-trivial modifications to previous algorithms.

6 External-Memory and Cache-Oblivious Algorithms

This section introduces our results on External-Memory algorithms and Cache-Oblivious algorithm [13]. It is shown that several asymptotically-optimal algorithms can each be adapted to the asymmetric setting with reduced writes (and thus lead to asymptotically better costs on the asymmetric models).

6.1 External-Memory Algorithms

Recall the definition of an Asymmetric External Memory (AEM) model that has a small primary memory (cache) of size M and transfers data in blocks of size B to (at a cost of ω) and from (at unit cost) an unbounded external memory. We show that three asymptotically-optimal EM sorting algorithms can each be adapted to the AEM with reduced write costs.

Multi-way mergesort An l-way mergesort is a balanced tree of merges with each merge taking in l sorted arrays and outputting one sorted array consisting of all records from the input. We assume that once the input is small enough a different sort (the *base case*) is applied. For $l = M/B$ and a base case of $n \leq M$ (using any sort since it fits in memory), we have the standard EM mergesort. With these settings there are $\log_{M/B}(n/M)$ levels of recursion, plus the base case, each costing $O(n/B)$ memory operations.

Following [37, 41], we adapt multi-way mergesort by merging $O(\omega)M/B$ sorted runs at a time (instead of M/B as in the original EM version). This change saves writes by reducing the depth of the recursion. However, it s no longer possible to keep the base case in the primary memory, nor one block for each of the input array during a merge. Hence each merge makes $O(\omega)$ passes over the runs, using an in-memory heap to extract values for the output run for the pass. The new algorithm takes $O(\omega n/B)$ reads and $O(n/B)$ write per level, and $\log_{O(\omega)M/B}(n/M)$ levels of recursion Our algorithm and analysis is somewhat simpler than [37, 41].

Sample sort We present an sample sort algorithm on the Asymmetric External Memory model. Classic sample sort on External Memory model is an l-way randomized sample sort that finds $l = M/B$ pivots at each level of recursion so that these pivots approximately partition the elements into l buckets. This algorithm takes $O(n/B)$ reads and writes per level and $O(log_{M/B}(n/B))$ levels.

Our new sample sort on the Asymmetric External Memory uses $O(\omega)M/B$ splitters at each level of recursion (instead of M/B in the original EM version). Again, the challenge is to both find the splitters and partition using them while incurring only $O(N/B)$ writes across each level of recursion. Similar to the mergesort, the processing of each level contains $O(\omega)$ phases, and each phase take $O(N/B)$ reads, and the overall writes for all phases is $O(N/B)$. In this way, the

amount of level of recursion decreases to $O(\log_{\omega M/B}(n/M))$ We also show how this algorithm can be parallelized to run with linear speedup for up to $p = n/M$ processors.

Heapsort Finally, our third sorting algorithm is a heapsort using a buffer-tree-based priority queue. A buffer tree [4] is an augmented version of an (a, b)-tree [28], where $a = l/4$ and $b = l$ for large branching factor l. In the original buffer tree $l = M/B$. As an (a, b) tree, all leaves are at the same depth in the tree, and all internal nodes have between $l/4$ and l children (except the root, which may have fewer). Thus the height of the tree is $O(1 + \log_l n)$. Each INSERT and DELETE-MIN has an amortized cost of $O((1/B)(1 + \log_l n))$.

To reduce the number of writes we instead set $l = O(\omega)M/B$. Compared to the original EM algorithm, both our buffer-tree nodes and the number of elements stored outside the buffer tree are larger by a factor of $O(\omega)$, which adds nontrivial changes to the data structure: (1) the buffer tree nodes are larger by a factor $O(\omega)$, (2) consequently, the "buffer-emptying" process uses an efficient sort on $O(\omega M)$ elements instead of an in-memory sort on M elements, and (3) to support the priority queue, $O(\omega M)$ elements are stored outside the buffer tree instead of $O(M)$. With the new buffer tree, we can build a priority queue that supports n INSERT and DELETE-MIN operations with an amortized cost of $O((\omega/B)(1 + \log_{\omega M/B} n))$ reads and $O((1/B)(1 + \log_{\omega M/B} n))$ writes per operation. Using the priority queue to implement a sorting algorithm trivially results in a sort costing a total of $O((\omega n/B)(1 + \log_{\omega M/B} n))$ reads and $O((n/B)(1 + \log_{\omega M/B} n))$ writes.

6.2 Cache-Oblivious Parallel Algorithms

This section presents low-depth cache-oblivious parallel algorithms for sorting and Fast Fourier Transform, with asymmetric read and write costs [11]. Both algorithms (i) have only polylogarithmic depth, (ii) are processor-oblivious (i.e., no explicit mention of processors), (iii) can be cache-oblivious or cache-aware, and (iv) map to low cache complexity on parallel machines with hierarchies of shared caches as well as private caches. We also present a linear-depth, cache-oblivious parallel algorithm for matrix multiplication. All three algorithms use $\Theta(\omega)$ fewer writes than reads.

Sorting We show the low-depth, cache-oblivious sorting algorithm from [10] can be adapted to the asymmetric case. The original algorithm is based on viewing the input as a $\sqrt{n} \times \sqrt{n}$ array, sorting the rows, partitioning them based on splitters, transposing the partitions, and then sorting the buckets. The original algorithm incurs $O((n/B)\log_M(n))$ reads and writes. To reduce the number of writes, our revised version partitions slightly differently: instead of viewing the input as a $\sqrt{n} \times \sqrt{n}$ array, we treat it as a $\sqrt{\omega n} \times \sqrt{n/\omega}$ array. Then for each row, we take $O(\omega)$ splitters and sort them. Similar to the External-Memory sorting algorithms, we can scan over the input for $O(\omega)$ times and partition the elements into $O(\sqrt{\omega n})$ buckets. This process recurses until the problem size fit into the small memory.

This new approach does extra reads to reduce the number of levels of recursion. The algorithm does $O\left((n/B)\log_{\omega M}(\omega n)\right)$ writes, $O\left((\omega n/B)\log_{\omega M}(\omega n)\right)$ reads, and has depth $O\left(\omega\log^2(n/\omega)\right)$ w.h.p.

Fast Fourier Transform We now consider a parallel cache-oblivious algorithm for computing the Discrete Fourier Transform (DFT). The algorithm we consider is based on the Cooley-Tukey FFT algorithm [20], and our description follows that of [24]. We assume that n at each level of recursion and ω are powers of 2. The standard cache-oblivious divide-and-conquer algorithm [24] views the input matrix as an $\sqrt{n}\times\sqrt{n}$ matrix, and incurs $O\left((n/B)\log_M(n)\right)$ reads and writes.

Similar to the sorting algorithm just mentioned, we change the algorithm such that now the input is treated as a $\omega\sqrt{n/\omega}\times\sqrt{n/\omega}$ array. Then for each submatrix with size $\omega\times\sqrt{n/\omega}$, we apply a brute-force calculation such that each value requires ω reads and operations, and the final value is written just once. Such computation of the FFT requires $O\left((n/B)\log_{\omega M}(\omega n)\right)$ writes, $O\left((\omega n/B)\log_{\omega M}(\omega n)\right)$ reads, and has depth $O(\omega\log n\log\log n)$.

Matrix Multiplication The standard cubic-work sequential algorithm trivially uses $O(n^3)$ reads and $\Theta(n^2)$ writes, one for each entry in the output matrix. For the EM model, the blocked algorithm that divides the matrix into sub-matrices of size $\sqrt{M}\times\sqrt{M}$ uses $O\left(n^3/(B\sqrt{M})\right)$ reads and $O(n^2/B)$ writes [14, 24].

Note that the standard cache-oblivious divide-and-conquer algorithm [14, 24] recurses on four parallel subproblems of size $n/2\times n/2$, resulting in $\Theta\left(n^3/(B\sqrt{M})\right)$ reads and writes. To reduce the writes by a factor of $\Theta(\omega)$, we instead recurse on ω^2 parallel subproblems (blocks) of size $n/\omega\times n/\omega$. Also, in the first round, we pick an integer b uniformly at random from 1 to $\lfloor\log_2\omega\rfloor$, and recurses on $b\times b$ subproblems. This new cache-oblivious matrix multiplication algorithm requires expected $O\left(n^3\omega/(B\sqrt{M}\log\omega)\right)$ reads and $O\left(n^3/(B\sqrt{M}\log\omega)\right)$ writes, and has $O(\omega n)$ depth. Comparing with the previous algorithm, the overall read and write cost of new algorithm expects to be reduced by a factor of $O(\log\omega)$.

7 Directions for Future Work

The previous sections in this thesis already introduce a lot of new results in models and algorithms. However, there are still plenty of interesting topics remaining to study. Here I list some of the potential directions for future work, which have some evidences that these problems should be solvable. Further research beyond these topics may also be considered as time permits.

7.1 Randomized Incremental Algorithms

In a recent ongoing work, I, together with Belloch, Shun and Sun, systematically studied randomized incremental algorithms and showed that most sequential random incremental algorithms are in fact parallel. The algorithms that are discussed include sorting; computational geometry problems of convex hull, Delaunay triangulation, and trapezoidal maps in two dimensions, linear programming, closest pair, and smallest enclosing disk in constant dimensions; and graph algorithms to compute least-element lists (for graph-distance embeddings) and strongly connected components.

We analyze the dependence between iterations in each algorithm, and show that the dependence structure is shallow for all of the algorithms, implying high parallelism. We identify three types of dependencies found in the algorithms studied and present a framework for analyzing each type of algorithms. The framework is then applied to each of the algorithms to give work-efficient polylogarithmic-depth parallel algorithms for most of the problems that we study.

It seems that generating the write-efficient versions of these algorithms are optimistic since they share almost the same computational (dependence) structure, and this structure has some unique properties (which are described in the paper). For example, all these algorithms try to "add" some primitives into the current solution of a subset of the primitives, and they can usually be processed together since they are less likely to have collisions with each other.

In that paper, a new concept is introduced and analyzed: an incremental algorithm can switch between an online version and an offline version. An online version of an incremental algorithm is to explicitly add a new randomly-chosen element into the existing configuration in each iteration. An example is to sort by inserting each key into a binary search tree successively. An offline version, on the other hand, keeps track of all the elements in the whole process. The corresponding example for sorting is the quicksort: once a pivot is chosen, each element (in the corresponding sublist) compares with the pivot simultaneously, instead of waiting until its corresponding iteration.

Both versions are optimal in terms of work (or time complexity sequentially) so previous research did not emphasize this difference. However, from the example of sorting, we can see that the offline version usually has relatively lower depth, since each element knows the place it should be "added" to the configuration immediately in its iteration. Thus the paper mainly uses the offline version. The problem is that, since the offline version requires to keep track of all elements in all iterations, it definitely requires more writes. The online version postpones this tracking process, so that usually only a constant number of writes are required per element. Thus, the online version is not only more intuitive but also generally requires less writes.

A straightforward question to ask is then:

(Potential Topic 1)**. *Can the randomized incremental algorithms be optimal on both work and depth, and also write-efficient?***

To solve it, we may want to figure out a hybrid solution of the online and offline versions, with other tools like exponential delaying.

7.2 Data Structures

The algorithms studied in previous sections are already based on many data structures. For example, a write-efficient random binary search tree is used in the sequential sorting algorithm, and a write-efficient variant of Fibonacci heap is used in the single-source shortest-path algorithm, both in Section 4. Despite of designing problem-based data structure, it is also interesting to know whether the commonly-used data structures can be write-efficient, such that we can apply them directly in the algorithms, just like the symmetric setting (equal read and write cost).

(Potential Topic 2)**. *Can the commonly-used data structures, such as search trees, heaps, augmented trees, etc., be write-efficient? If so, can they be parallelized?***

The answer of sequential solutions seems promising. For example, a red-black tree is a candidate that each insertion and deletion takes amortized constant work. It seems that AVL tree, weight-balance tree and treap should all have the property, although further analysis is required. The parallel version of such algorithms seems much harder, but at least for treap, the solution should exist.

Balanced binary search tree is useful in many applications and can be used to implement a priority queue. However, some specific types of heap, like a Fibonacci heap, are still useful in certain situations, and it is meaningful to know their performance, i.e. the required numbers of writes, in different applications.

Augmented trees usually require logarithmic writes per operation, which is less efficient in terms of the number of writes. Similarly to the external sorting algorithms in Section 6, it seems possible to trade off writes for reads. It is interesting to study the best trade-off point, and the parallel implementations (to handle concurrency).

These data structures are just some examples that may lead to interesting results. Other data structures will also be studied if time permits.

7.3 Experiment Verification

In spite of the interesting theoretical results mentioned in this thesis, since the motivation of this research is driven by practical considerations, it is essential to confirm whether our new algorithms and techniques actually perform better on the new memory system. Although the new memory is not available at this time, an internal version for testing might be accessible soon, and such experiment should get started immediately.

(Potential Topic 3)**. *Do the new algorithms have better performance on the new memory? If so, by how much? Also, what techniques generally perform well, and why or why not?***

Here we use the sorting algorithm to be an example again, which is one of the most commonly-used algorithms in all different styles of systems and environments. The classic solutions are based on quicksort (constant pivots for sequential implementation or multiple pivots for parallel implementation). The costs of these algorithms in our new models are high, but will they actually perform well since the latency may be hidden by the caches? Nevertheless, at least for what I understand, the parallel implementation is very likely to be bottlenecked by writes. The performances of the classic algorithms will be the baseline to be compared with.

In different sections of this thesis, we mentioned six different new sorting algorithms in various settings. For the sequential and parallel sorting algorithm on ARAM, the focus is to minimize the cost (I/O cost and time) of the algorithm, rather than actually proposing an efficient implementation. In the experiment, it is also important to discuss the details to write the code so that they will be both efficient in terms of complexity and real-world performance. For external and cache-oblivious sorting algorithms, they have already corresponded to efficient implementations, so it is interesting to know that which of them is faster than others (especially in parallel) and understand the reasons. Lastly, the final question is, do the new algorithms perform better than the baseline algorithms, and does the speedup (or reduction on energy consumption) follow our theoretical analyses?

Similar questions can be ask to other fundamental primitives for algorithm design, like search trees and graph algorithms including BFS, shortest-path, etc. If the answers of Topic 3 on these primitives are positive, then a straight-forward but significant next step is to yield a library that abstracts the interfaces of these problems and provide efficient implementations. In this way, further algorithm design under this setting can directly use this implementation, rather than constructing every subroutine from scratch.

(Potential Topic 4)**. *To write a library that provides efficient implementations of fundamental algorithms is of significant importance.***

A Motivation: Detail Information of Asymmetric Memory

While DRAM stores data in capacitors that typically require refreshing every few milliseconds, and hence must be continuously powered, emerging NVM technologies store data as "states" of the given material that require no external power to retain. Energy is required only to read the cell or change its value (i.e., its state). While there is no significant cost difference between reading and writing DRAM (each DRAM read of a location not currently buffered requires a write of the DRAM row being evicted, and hence is also a write), emerging NVMs incur significantly higher cost for writing than reading. This large gap seems fundamental to the technologies themselves: to change the physical state of a material requires relatively significant energy for a sufficient duration, whereas reading the current state can be done quickly and, to ensure the state is left unchanged, with low energy. Existing research has shown such asymmetry, for example:

- A cell in Spin-Torque Transfer Magnetic RAM can be read in 0.14 ns but uses a 10 ns writing pulse duration, using roughly 10^{-15} joules to read versus 10^{-12} joules to write [21] (these are the raw numbers at the materials level).
- A Memristor Resistive RAM cell uses a 100 ns write pulse duration, and an 8MB Memristor RRAM chip is projected to have reads with 1.7 ns latency and 0.2 nJ energy versus writes with 200 ns latency and 25 nJ energy [42], over two orders of magnitude differences in latency and energy.
- Phase-change memory is the most mature of the three technologies, and early generations are already available as I/O devices. A recent paper [33] reported 6.7 μs latency for a 4KB read and 128 μs latency for a 4KB write. Another reported that the sector I/O latency and bandwidth for random 512B writes was a factor of 15 worse than for reads [29]. As a future memory/cache replacement, a 512MB PCM memory chip is projected to have 16 ns byte reads versus 416 ns byte writes, and writes to a 16MB PCM L3 cache are projected to be up to 40 times slower and use 17 times more energy than reads [22].

While these numbers are speculative and subject to change as the new technologies emerge over time, there seems to be sufficient evidence that writes will be considerably more costly than reads in these NVMs. Thus, studying write-efficient (parallel) algorithms and systems is of significant, lasting importance.

Note that, unlike SSDs and earlier versions of phase-change memory products, these emerging memory products will sit on the processor's memory bus and be accessed at byte granularity via loads and stores (like DRAM). Thus, the time and energy for reading can be roughly on par with DRAM, and depends primarily on the properties of the technology itself relative to DRAM.

Bibliography

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theory Comput. Sys.*, 35(3), 2002. 3.3

[2] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *CACM*, 31(9), 1988. 3.3

[3] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. Onyx: A prototype phase change memory storage array. In *HotStorage*, 2011. 1, 1

[4] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1), 2003. 6.1

[5] M. Athanassoulis, B. Bhattacharjee, M. Canim, and K. A. Ross. Path processing using solid state storage. In *ADMS*, 2012. 1, 1

[6] A. Ben-Aroya and S. Toledo. Competitive analysis of flash-memory algorithms. In *ESA*, 2006. 1

[7] Naama Ben David, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, Charlie McGuffey, and Julian Shun. Parallel algorithms with asymmetric read and write costs. *arXiv preprint*, 2016. 1, 3.2, 3.3

[8] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *SPAA*, 2004. 3.3

[9] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46(2):281–321, March 1999. 3.2

[10] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Low-depth cache oblivious algorithms. In *Symposium on Parallelism in Algorithms and Architectures*, pages 189–199, 2010. 3.3, 6.2

[11] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, and Julian Shun. Sorting with asymmetric read and write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures*, 2015. 1, 3.3, 6.2

[12] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, and Julian Shun. Efficient algorithms under asymmetric read and write costs. *arXiv preprint arXiv:1511.01038*, 2015. 1, 3.3

[13] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, and Julian Shun. Sorting with asymmetric read and write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2015. 1, 3.3, 3.3, 3.3, 3.3

[14] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of

dag-consistent distributed shared-memory algorithms. In *SPAA*, 1996. 6.2

[15] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM (JACM)*, 46(5), September 1999. 3.2

[16] Erin Carson, James Demmel, Laura Grigori, Nicholas Knight, Penporn Koanantakool, Oded Schwartz, and Harsha Vardhan Simahdri. Write-avoiding algorithms. Technical Report UCB/EECS-2015-163, U.C. Berkeley, 2015. 1, 1

[17] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *CIDR*, 2011. 1

[18] Sangyeun Cho and Hyunjin Lee. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *MICRO*, 2009. 1

[19] Stephen Cook and Ravi Sethi. Storage requirements for deterministic polynomial time recognizable languages. *Journal of Computer and System Sciences (JCSS)*, 13(1):25 – 37, 1976. 3.3

[20] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19, 1965. 6.2

[21] X. Dong, X. Wu, G. Sun, Y. Xie, H. Li, and Y. Chen. Circuit and microarchitecture evaluation of 3D stacking magnetic RAM (MRAM) as a universal memory replacement. In *DAC*, 2008. 1, 1, A

[22] X. Dong, N. P. Jouupi, and Y. Xie. PCRAMsim: System-level performance, energy, and area modeling for phase-change RAM. In *ICCAD*, 2009. 1, 1, A

[23] D. Eppstein, M. T. Goodrich, M. Mitzenmacher, and P. Pszona. Wear minimization for cuckoo hashing: How not to throw a lot of eggs into one basket. In *SEA*, 2014. 1

[24] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS*, 1999. 3.3, 6.2

[25] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2), 2005. 1

[26] HP. HP, SanDisk partner on memristor, ReRAM technology. http://www.bit-tech.net/news/hardware/2015/10/09/hp-sandisk-reram-memristor, October 2015. 1, 1

[27] J. Hu, Q. Zhuge, C. J. Xue, W.-C. Tseng, S. Gu, and E. Sha. Scheduling to optimize cache utilization for non-volatile main memories. *IEEE Transactions on Computers*, 63(8), 2014. 1

[28] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17, 1982. 6.1

[29] IBM. www.slideshare.net/IBMZRL/theseus-pss-nvmw2014, 2014. 1, 1, A

[30] Intel. Intel and Micron produce breakthrough memory technology. http://newsroom.intel.com/community/ intel_newsroom/blog/2015/07/28/intel-and-

micron-produce-breakthrough-memory-technology, July 2015. 1, 1

[31] Hong Jia-Wei and Hsiang-Tsung Kung. I/o complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333. ACM, 1981. 3.3

[32] David R Karger, Philip N Klein, and Robert E Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM (JACM)*, 42(2):321–328, 1995. 3.3

[33] H. Kim, S. Seshadri, C. L. Dickey, and L. Chu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *FAST*, 2014. 1, 1, A

[34] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *ISCA*, 2009. 1

[35] J. S. Meena, S. M. Sze, U. Chand, and T.-Y. Tseng. Overview of emerging nonvolatile memory technologies. *Nanoscale Research Letters*, 2014. 1, 1

[36] Suman Nath and Phillip B. Gibbons. Online maintenance of very large random samples on flash storage. *VLDB J.*, 19(1), 2010. 1

[37] Hyoungmin Park and Kyuseok Shim. FAST: flash-aware external sorting for mobile database systems. *Journal of Systems and Software*, 82(8), 2009. doi: 10.1016/j.jss.2009.02.028. URL `http://dx.doi.org/10.1016/j.jss.2009.02.028`. 1, 6.1

[38] M. K. Qureshi, S. Gurumurthi, and B. Rajendran. *Phase Change Memory: From Devices to Systems*. Morgan & Claypool, 2012. 1, 1

[39] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2), 1985. ISSN 0001-0782. doi: 10.1145/2786.2793. URL `http://doi.acm.org/10.1145/2786.2793`. 3.3

[40] S. D. Viglas. Adapting the B^+-tree for asymmetric I/O. In *ADBIS*, 2012. 1

[41] Stratis Viglas. Write-limited sorts and joins for persistent memory. *PVLDB*, 7(5), 2014. URL `http://www.vldb.org/pvldb/vol7/p413-viglas.pdf`. 1, 6.1

[42] C. Xu, X. Dong, N. P. Jouppi, and Y. Xie. Design implications of memristor-based RRAM cross-point structures. In *DATE*, 2011. 1, 1, A

[43] Byung-Do Yang, Jae-Eun Lee, Jang-Su Kim, Junghyun Cho, Seung-Yun Lee, and Byoung-Gon Yu. A low power phase-change random access memory using a data-comparison write scheme. In *ISCAS*, 2007. 1

[44] Yole Developpement. Emerging non-volatile memory technologies, 2013. 1, 1

[45] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *ISCA*, 2009. 1