Sorting with Asymmetric Read and Write Costs

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

Jeremy T. Fineman
Georgetown University
jfineman@cs.georgetown.edu

Phillip B. Gibbons
Intel Labs & CMU
gibbons@cs.cmu.edu

Yan Gu
Carnegie Mellon University
yan.gu@cs.cmu.edu

Julian Shun
Carnegie Mellon University
jshun@cs.cmu.edu

ABSTRACT

Emerging memory technologies have a significant gap between the cost, both in time and in energy, of writing to memory versus reading from memory. In this paper we present models and algorithms that account for this difference, with a focus on write-efficient sorting algorithms. First, we consider the PRAM model with asymmetric write cost, and show that sorting can be performed in $O(n)$ writes, $O(n \log n)$ reads, and logarithmic depth (parallel time). Next, we consider a variant of the External Memory (EM) model that charges $k > 1$ for writing a block of size B to the secondary memory, and present variants of three EM sorting algorithms (multi-way mergesort, sample sort, and heapsort using buffer trees) that asymptotically reduce the number of writes over the original algorithms, and perform roughly k block reads for every block write. Finally, we define a variant of the Ideal-Cache model with asymmetric write costs, and present write-efficient, cache-oblivious parallel algorithms for sorting, FFTs, and matrix multiplication. Adapting prior bounds for work-stealing and parallel-depth-first schedulers to the asymmetric setting, these yield parallel cache complexity bounds for machines with private caches or with a shared cache, respectively.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sorting and searching*

Keywords

Sorting, asymmetric read-write costs, non-volatile memory, persistent memory, write-efficient, write-avoiding, parallel algorithms, cache-oblivious algorithms, external memory model, mergesort, sample sort, I/O buffer tree, FFT, matrix multiplication.

1. INTRODUCTION

Emerging nonvolatile/persistent memory (NVM) technologies such as Phase-Change Memory (PCM), Spin-Torque Transfer Magnetic RAM (STT-RAM), and Memristor-based Resistive RAM (ReRAM) 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 [27, 40], 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 [27]). Thus, unlike DRAM, there is a significant (often an order of magnitude or more) asymmetry between read and write costs [3, 6, 16, 17, 23, 25, 31, 38], motivating the study of *write-efficient* (*write-limited*, *write-avoiding*) algorithms, which reduce the number of writes relative to existing algorithms.

Related Work. While read-write asymmetry in NVMs has been the focus of many systems efforts [13, 26, 39, 41, 42], there have been very few papers addressing this from the algorithmic perspective. 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 [7, 18, 21, 28, 30, 36] have looked at read-write asymmetries in the context of NAND flash memory. This work has focused on (i) the fact that in NAND flash chips, bits can only be cleared by incurring the overhead of erasing a large block of memory, and/or (ii) avoiding individual cell wear out. Eppstein et al. [18], for example, presented a novel cuckoo hashing algorithm that chooses where to insert/reinsert each item based on the number of writes to date for the cells it hashes to. Emerging NVMs, in contrast, do not suffer from (i) because they can write arbitrary bytes in-place. As for (ii), we choose not to focus on wear out (beyond reducing the total number of writes to all of memory) because system software (e.g., the garbage collector, virtual memory manager, or virtual machine hypervisor) can readily balance application writes across the physical memory over the long time horizons (many orders of magnitude longer than NAND Flash) before an individual cell would wear out from too many writes to it.

A few prior works [12, 36, 37] have looked at algorithms for asymmetric read-write costs in emerging NVMs, in the context of databases. Chen et al. [12] presented analytical formulas for PCM latency and energy, as well as algorithms for B-trees and hash joins that are tuned for PCM. For example, their B-tree variant does not sort the keys in a leaf node nor repack a leaf after a deleted key, thereby avoiding the write cost of sorting and repacking, at the expense of additional reads when searching. Similarly, Viglas [37] traded off fewer writes for additional reads by rebalancing a B^+-tree only if the cost of rebalancing has been amortized. Viglas [36] presented several "write-limited" sorting and join algorithms within the context of database query processing.

SPAA'15, June 13–15, 2015, Portland, OR, USA.

ACM 978-1-4503-3588-1/15/06 ...$15.00.
DOI: http://dx.doi.org/10.1145/2755573.2755604 .

Our Results. In this paper, we seek to systematically study algorithms under asymmetric read and write costs. We consider natural extensions to the RAM/PRAM models, to the External Memory model, and to the Ideal-Cache model to incorporate an integer cost, $k > 1$, for writes relative to reads. We focus primarily on sorting algorithms, given their fundamental role in computing and as building blocks for other problems, especially in the External Memory and Parallel settings—but we also consider additional problems.

We first observe that in the RAM model, it is well known that sorting by inserting each key into a balanced search tree requires only $O(n)$ writes with no increase in reads ($O(n \log n)$). Applying this idea to a carefully-tuned sort for the asymmetric CRCW PRAM yields a parallel sort with $O(n)$ writes, $O(n \log n)$ reads and $O(k \log n)$ depth (with high probability[1]).

Next, we consider an Asymmetric External Memory (AEM) model, which has a small primary memory (cache) of size M and transfers data in blocks of size B to (at a cost of k) 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. First, following [30, 36], we adapt multi-way mergesort by merging kM/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. Each merge makes k passes over the runs, using an in-memory heap to extract values for the output run for the pass. Our algorithm and analysis is somewhat simpler than [30, 36]. Second, we present an AEM sample sort algorithm that uses kM/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. We also show how this algorithm can be parallelized to run with linear speedup on the Asymmetric Private-Cache model (Section 2) with $p = n/M$ processors. Finally, our third sorting algorithm is an AEM heapsort using a buffer-tree-based priority queue. 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 k, which adds nontrivial changes to the data structure. All three sorting algorithms have the same asymptotic complexity on the AEM.

Finally, we define an Asymmetric Ideal-Cache model, which is similar to the AEM model in terms of M and B and having asymmetric read/write costs, but uses an asymmetric ideal replacement policy instead of algorithm-specified transfers. We extend important results for the Ideal-Cache model and thread schedulers to the asymmetric case—namely, the Asymmetric Ideal-Cache can be (constant factor) approximated by an asymmetric-LRU cache, and it can be used in conjunction with a work-stealing (parallel-depth-first) scheduler to obtain good parallel cache complexity bounds for machines with private caches (a shared cache, respectively). We use this model to design write-efficient cache-oblivious algorithms for sorting, Fast Fourier Transform, and matrix multiplication. Our sorting algorithm is adapted from [9], and again deals with the challenges of reducing the number of writes. All three algorithms use $\Theta(k)$ times more reads than writes and have good parallelism.

2. PRELIMINARIES AND MODELS

This section presents background material on NVMs and models, as well as new (asymmetric cost) models and results relating models. We first consider models whose parallelism is in the parallel transfer of the data in a larger block, then consider models with parallel processors.

Emerging NVMs. 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 such as Phase-Change Memory (PCM), Spin-Torque Transfer Magnetic RAM (STT-RAM), and Memristor-based Resistive RAM (ReRAM) each 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. An STT-RAM cell, for example, 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 [17] (these are the raw numbers at the materials level). A Memristor ReRAM cell uses a 100 ns write pulse duration, and an 8MB Memrister ReRAM 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 [38]—over two orders of magnitude differences in latency and energy. PCM is the most mature of the three technologies, and early generations are already available as I/O devices. A recent paper [25] 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 [23]. 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 [16]. 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.

Sorting. The sorting problem we consider is the standard comparison based sorting with n records each containing a key. We assume the input is in an unsorted array, and the output needs to be placed into a sorted array. Without loss of generality, we assume the keys are unique (a position index can always be added to make them unique).

The Asymmetric RAM model. This is the standard RAM model but with a cost $k > 1$ for writes, while reads are still unit cost.

The (Asymmetric) External Memory model. 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. Sorting n records can be performed in the EM model with I/O complexity

$$\Theta\Big(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B}\Big) \tag{1}$$

[1] *With high probability (w.h.p.)* means with probability $1 - n^{-c}$, for a constant c.

This is both an upper and lower bound [2]. The upper bound can be achieved with at least three different algorithms, a multi-way mergesort [2], a distribution sort [2], and a priority-queue (heap) sort based on buffer trees [4].

The *Asymmetric External Memory* (AEM) model simply adds a parameter k to the EM model, and charges this for each write of a block. Reading a block still has unit cost.

Throughout the paper, we assume that M and B are measured in terms of the number of data objects. If we are sorting, for example, it is the number records. We assume that the memory has an extra $O(\log M)$ locations just for storing a stack to compute with.

The (Asymmetric) Ideal-Cache model. The Ideal-Cache model [20] 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 model makes the *tall cache assumption* where $M = \Omega(B^2)$, which is easily met in practice. The I/O or *cache complexity* of an algorithm is the number of cache misses. 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 [35].

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. The I/O complexity of cache-oblivious sorting is asymptotically the same as for the EM model.

We define the *Asymmetric Ideal-Cache* model 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 $1+k$, 1 for the read and k for the write. Again, we 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 $k+1$ because the block is dirty. We call this the read-write LRU policy. This policy is competitive with the optimal offline policy:

LEMMA 2.1. *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*

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

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

PROOF. Partition the sequence of instructions into regions that contain memory reads to exactly M_L/B distinct memory blocks each (except perhaps the last). Each region will require at most M_L/B misses under LRU. Each will also require at least $(M_L - M_I)/B$ cache misses on the ideal cache since at most M_I/B blocks can be in the cache at the start of the region. The same argument can be made for writes, but in this case each operation involves evicting a dirty block. The $(1+k)M_I/B$ is for the last region. To account for the last region, in the worst case at the start of the last write region the ideal cache starts with M_I/B blocks which get written to, while the LRU starts with none of those blocks. The LRU therefore invokes an addition M_I/B write misses each costing $1+k$ (1 for the load and k for the eviction). Note that if the cache starts empty then we do not have to add this term since an equal amount will be saved in the first round. □

The Asymmetric PRAM model. In the *Asymmetric PRAM*, the standard PRAM is augmented such that each write costs k and all other instructions cost 1. In this paper we analyze algorithms in terms of work (total cost of the operations) and depth (parallel time using an unbounded number of processors). If we have depth $d(n)$ and separate the work into $w(n)$ writes and $r(n)$ other instructions, then the time on p processors is bounded by:

$$T(n,p) = O\left(\frac{kw(n) + r(n)}{p} + d(n)\right)$$

using Brent's theorem [24]. This bound assumes that work can be allocated to processors efficiently. We allow for concurrent reads and writes (CRCW), and for concurrent writes we assume an arbitrary write takes effect. Note that a parallel algorithm that require $O(D)$ depth in the PRAM model requires $O(kD)$ depth in the asymmetric PRAM model to account for the fact that writes are k times more expensive than reads.

The Asymmetric Private-Cache model. In the *Asymmetric Private-Cache* model (a variant of the Private-Cache model [1, 5]), each processor has its own primary memory of size M, and all processors share a secondary memory. We allow concurrent reads but do not use concurrent writes. As in the AEM model, transfers are in blocks of size B and transfers to the shared memory cost k.

The (Asymmetric) Low-depth Cache-Oblivious Paradigm. The final model that we consider is based on developing low-depth cache-oblivious algorithms [9]. In the model algorithms are defined as nested parallel computations based on parallel loops, possibly nested (this is a generalization of a PRAM). The depth of the computation is the longest chain of dependences—i.e., the depth of a sequential strand of computation is its sequential cost, 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. The cache complexity can be analyzed on the Ideal-Cache model under this sequential order.

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 $Q_1 + O(pDM/B)$ with high probability [1]. 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]. These bounds can be extended to multi-

Algorithm 1 ASYMMETRIC-PRAM SORT

Input: An array of records A of length n

1: Select a sample S from A independently at random with per-record probability $1/\log n$, and sort the sample.
2: Use every $(\log n)$-th element in the sorted S as splitters, and for each of the about $n/\log^2 n$ buckets defined by the splitters allocate an array of size $c\log^2 n$.
3: In parallel locate each record's bucket using a binary search on the splitters.
4: In parallel insert the records into their buckets by repeatedly trying a random position within the associated array and attempting to insert if empty.
5: Pack out all empty cells in the arrays and concatenate all arrays.
// Step 6 is an optional step used to obtain $O(k\log n)$ depth
6: **For** round $r \leftarrow 1$ to 2 **do**
  for each array A' generated in previous round
    Deterministically select $|A'|^{1/3} - 1$ samples as splitters and apply integer sort on the bucket number to partition A' into $|A'|^{1/3}$ sub-arrays.
7: **For each** subarray apply the asymmetric RAM sort.
8: Return the sorted array.

level hierarchies of private or shared caches, respectively [9]. Thus, algorithms with low depth have good parallel cache complexity.

Our asymmetric variant of the low-depth cache-oblivious paradigm simply accounts for k in the depth and uses the Asymmetric Ideal-Cache model for sequential cache complexity. We observe that the above scheduler bounds readily extend to this asymmetric setting. The $O(pDM/B)$ bound on the additional cache misses under work-stealing arises from an $O(pD)$ bound on the number of steals and the observation that each steal requires the stealer to incur $O(M/B)$ misses to "warm up" its cache. Pessimistically, we will charge $2M/B$ writes (and reads) for each steal, because each line may be dirty and need writing back before the stealer can read it into its cache and, once the stealer has completed the stolen work (reached the join corresponding to the fork that spawned the stolen work), the contents of its cache may need to be written back. Therefore for private caches we have $Q_P \leq Q_1 + O(pkDM/B)$. The PDF bounds extend because there are no additional cache misses and hence no additional reads or writes.

3. SORTING ON RAM/PRAM

The number of writes on an asymmetric RAM can be bound for a variety of algorithms and data structures using known techniques. For example, there has been significant research on maintaining balanced search trees such that every insertion and deletion only requires a constant number of rotations (see e.g., [29] and references within). While the motivation for that work is that for certain data structures rotations can be asymptotically more expensive than visiting a node (e.g., if each node of a tree maintains a secondary set of keys), the results apply directly to improving bounds on the asymmetric RAM. Sorting can be done by inserting n records into a balanced search tree data structure, and then reading them off in order. This requires $O(n\log n)$ reads and $O(n)$ writes, for total cost $O(n(k+\log n))$. Similarly, we can maintain priority queues (insert and delete-min) and comparison-based dictionaries (insert, delete and search) in $O(1)$ writes per operation.

We now consider how to sort on an asymmetric CRCW PRAM (arbitrary write). Algorithm 1 outlines a sample sort (with over-sampling) that does $O(n\log n)$ reads and $O(n)$ writes and has depth $O(k\log n)$. It is similar to other sample sorts [10, 19, 24]. We consider each step in more detail and analyze its cost.

Step 1 can use Cole's parallel mergesort [14] requiring $O(n)$ reads and writes w.h.p. (because the sample is size $\Theta(n/\log n)$ w.h.p.), and $O(k\log n)$ depth. In step 2 for sufficiently large c, w.h.p. all arrays will have at least twice as many slots as there are records belonging to the associated bucket [10]. The cost of step 2 is a lower-order term. Step 3 requires $O(n\log n)$ reads, $O(n)$ writes and $O(k+\log n)$ depth for the binary searches and writing the resulting bucket numbers. Step 4 is an instance of the so-called placement problem (see [32, 33]). This can be implementing by having each record select a random location within the array associated with its bucket and if empty, attempting to insert the record at that location. This is repeated if unsuccessful. Since multiple records might try the same location at the same time, each record needs to check if it was successfully inserted. The expected number of tries per record is constant. Also, if the records are partitioned into groups of size $\log n$ and processed sequentially within the group and in parallel across groups, then w.h.p. no group will require more than $O(\log n)$ tries across all of its records [32]. Therefore, w.h.p., the number of reads and writes for this step are $O(n)$ and the depth is $O(k\log n)$. Step 5 can be done with a prefix sum, requiring a linear number of reads and writes, and $O(k\log n)$ depth. At this point we could apply the asymmetric RAM sort to each bucket giving a total of $O(n\log n)$ reads, $O(n)$ writes and a depth of $O\big(k\log^2 n + \log^2 n\log\log n\big)$ w.h.p. (the first term for the writes and second term for the reads).

We can reduce the depth to $O(k\log n)$ by further deterministically sampling inside each bucket (step 6) using the following lemma:

LEMMA 3.1. *We can partition m records into $m^{1/3}$ buckets $M_1,\ldots,M_{m^{1/3}}$ such that for any i and j where $i<j$ all records in M_i are less than all records in M_j, and for all i, $|M_i| < m^{2/3}\log m$. The process requires $O(m\log m)$ reads, $O(m)$ writes, and $O(k\sqrt{m})$ depth.*

PROOF. We first split the m records into groups of size $m^{1/3}$ and sort each group with the RAM sort. This takes $O(m\log m)$ reads, $O(m)$ writes and $O\Big(km^{1/3}\log m\Big)$ depth. Then for each sorted group, we place every $\log m$'th record into a sample. Now we sort the sample of size $m/\log m$ using Cole's mergesort, and use the result as splitters to partition the remaining records into buckets. Finally, we place the records into their respective buckets by integer sorting the records based on their bucket number. This can be done with a parallel radix sort in a linear number of reads/writes and $O(k\sqrt{m})$ depth [32].

To show that the largest bucket has size at most $m^{2/3}\log m$, note that in each bucket, we can pick at most $\log m$ consecutive records from each of the $m^{2/3}$ groups without picking a splitter. Otherwise there will be a splitter in the bucket, which is a contradiction. □

Step 6 applies two iterations of Lemma 3.1 to each bucket to partition it into sub-buckets. For an initial bucket of size m, this process will create sub-buckets of at most size $O\Big(m^{4/9}\log^{5/3} m\Big)$. Plugging in $m = O\big(\log^2 n\big)$ gives us that the largest sub-bucket is of size $O\Big(\log^{8/9} n(\log\log n)^{5/3}\Big)$. We can now apply the RAM sort to each bucket in $O(k\log n)$ depth. This gives us the following theorem.

THEOREM 3.2. *Sorting n records can be performed using $O(n\log n)$ reads, $O(n)$ writes, and in $O(k\log n)$ depth w.h.p. on the Asymmetric CRCW PRAM.*

This implies

$$T(n) = O\left(\frac{n \log n + kn}{p} + k \log n\right)$$

time. Allocating work to processors is outlined above or described in the cited references. In the standard PRAM model, the depth of our algorithm matches that of the best PRAM sorting algorithm [14], although ours is randomized and requires the CRCW model. We leave it open whether the same bounds can be met deterministically and on a PRAM without concurrent writes.

4. EXTERNAL MEMORY SORTING

In this section, we present sorting algorithms for the Asymmetric External Memory model. We show how the three approaches for EM sorting—mergesort, sample sort, and heapsort (using buffer trees)—can each be adapted to the asymmetric case.

In each case we trade off a factor of k additional reads for a larger branching factor (kM/B instead of M/B), hence reducing the number of rounds. It is interesting that the same general approach works for all three types of sorting. The first algorithm, the mergesort, has been described elsewhere [30] although in a different model (their model is specific to NAND flash memory and has different sized blocks for reading and writing, among other differences). Our parameters are therefore different, and our analysis is new. To the best of our knowledge, our other two algorithms are new.

4.1 Mergesort

We use an l-way mergesort—i.e., 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. This gives the well-known overall bound from Equation 1 [2].

To modify the algorithm for the asymmetric case, we increase the branching factor and the base case by a factor of k, i.e. $l = kM/B$ and a base case of $n \leq kM$. This means that it is no longer possible to keep the base case in the primary memory, nor one block for each of the input arrays during a merge. The modified algorithm is described in Algorithm 2.

Each merge proceeds in a sequence of rounds, where a round is one iteration of the **while** loop starting on line 5. During each round we maintain a priority queue within the primary memory. Because operations within the primary memory are free in the model, this can just be kept as a sorted array of records, or even unsorted, although a balanced search tree can be a feasible solution in practice. Each round consists of two phases. The first phase (the **for** loop on line 6) considers each of the l input subarrays in turn, loading the current block for the subarray into the load buffer, and then inserting each record e from the block into the priority queue if not already written to the output (i.e. $e.key >$ *lastV*), and if smaller than the maximum in the queue (i.e. $e.key <$ *Q.max*). This might bump an existing element out of the queue. Also, if a record is the last in its block then it is marked and tagged with its subarray number.

The second phase (the **while** loop starting on line 8) starts writing the priority queue to the output one block at a time. Whenever reaching a record that is marked as the last in its block, the algorithm increments the pointer to the corresponding subarray and processes the next block in the subarray. We repeat the rounds until all records from all subarrays have been processed.

Algorithm 2 AEM-MERGESORT

Input: An array A of records of length n

1: **if** $|A| \leq kM$ **then** // base case
2: Sort A using $k|A|/B$ reads and $|A|/B$ writes, and return.
3: Evenly partition A into $l = kM/B$ subarrays $A_1, \ldots, A_l$ (at the granularity of blocks) and recursively apply AEM-MERGESORT to each.
4: **Initialize Merge.** Initialize an empty output array O, a load buffer and an empty store buffer each of size B, an empty priority queue Q of size M, an array of pointers $I_1, \ldots, I_l$ that point to the start of each sorted subarray, $c = 0$, and *lastV* $= -\infty$. Associated with Q is *Q.max*, which holds the maximum element in Q if Q is full, and $+\infty$ otherwise.
5: **while** $c < |A|$ **do**
6: **for** $i \leftarrow 1$ to l **do**
7: PROCESS-BLOCK(i).
8: **while** Q is not empty **do**
9: $e \leftarrow$ *Q.deleteMin*.
10: Write e to the store buffer, $c \leftarrow c + 1$.
11: If the store buffer is full, flush it to O and update *lastV*.
12: **if** e is marked as last record in its subarray block **then**
13: $i =$ *e.subarray*.
14: Increment I_i to point to next block in subarray i.
15: PROCESS-BLOCK(i).
16: $A \leftarrow O$. // Logically, don't actually copy

17: **function** PROCESS-BLOCK(subarray i)
18: **If** I_i points to the end of the subarray **then** return.
19: Read the block I_i into the load buffer.
20: **for all** records e in the block **do**
21: **if** $e.key$ is in the range (*lastV*, *Q.max*) **then**
22: If Q is full, eject *Q.max*.
23: Insert e into Q, and mark if last record in block.

To account for the space for the pointers $I = I_1, \ldots, I_l$, let $\alpha = (\log n)/s$, where s is the size of a record in bits, and n is the total number of records being merged. The cost of the merge is bounded as follows:

LEMMA 4.1. *$l = kM/B$ sorted sequences with total size n (stored in $\lceil n/B \rceil$ blocks, and block aligned) can be merged using at most $(k+1)\lceil n/B \rceil$ reads and $\lceil n/B \rceil$ writes, on the AEM model with primary memory size $(M + 2B + 2\alpha kM/B)$.*

PROOF. Each round (except perhaps the last) outputs at least M records, and hence the total number of rounds is at most $\lceil n/M \rceil$. The first phase of each round requires at most kM/B reads, so the total number of reads across all the first phases is at most $k\lceil n/B \rceil$ (the last round can be included in this since it only loads as many blocks as are output). For the second phase, a block is only read when incrementing its pointer, therefore every block is only read once in the second phase. Also every record is only written once. This gives the stated bounds on the number of reads and writes. The space includes the space for the in-memory heap (M), the load and store buffers, the pointers I ($\alpha kM/B$), and pointers to maintain the last-record in block information ($\alpha kM/B$). □

We note that it is also possible to keep I in secondary memory. This will double the number of writes because every time the algorithm moves to a new block in an input array i, it would need to write out the updated I_i. The increase in reads is small. Also, if one uses a balanced search tree to implement the priority queue Q then

the size increases by $< M(\log M)/s$ in order to store the pointers in the tree.

For the base case when $n \leq kM$ we use the following lemma.

LEMMA 4.2. *$n \leq kM$ records stored in $\lceil n/B \rceil$ blocks can be sorted using at most $k\lceil n/B \rceil$ reads and $\lceil n/B \rceil$ writes, on the AEM model with primary memory size $M + B$.*

PROOF. We sort the elements using a variant of selection sort, scanning the input list a total of at most k times. In the first scan, store in memory the M smallest elements seen so far, performing no writes and $\lceil n/B \rceil$ reads. After completing the scan, output all the $\min(M, n)$ elements in sorted order using $\lceil \min(M, n)/B \rceil$ writes. Record the maximum element written so far. In each subsequent phase (if not finished), store in memory the M smallest records larger than the maximum written so far, then output as before. The cost is $\lceil n/B \rceil$ reads and M/B writes per phase (except perhaps the last phase). We need one extra block to hold the input. The largest output can be stored in the $O(\log M)$ locations we have allowed for in the model. This gives the stated bounds because every element is written out once and the input is scanned at most k times. □

Together we have:

THEOREM 4.3. *Algorithm 2 sorts n records using*

$$R(n) \leq (k+1)\left\lceil \frac{n}{B} \right\rceil \left\lceil \log_{\frac{kM}{B}} \left(\frac{n}{B} \right) \right\rceil$$

reads, and

$$W(n) \leq \left\lceil \frac{n}{B} \right\rceil \left\lceil \log_{\frac{kM}{B}} \left(\frac{n}{B} \right) \right\rceil$$

writes on an AEM with primary memory size $(M + 2B + 2\alpha kM/B)$.

PROOF. The number of recursive levels of merging is bounded by $\left\lceil \log_{\frac{kM}{B}} \left(\frac{n}{kM} \right) \right\rceil$, and when we add the additional base round we have $1 + \left\lceil \log_{\frac{kM}{B}} \left(\frac{n}{kM} \right) \right\rceil = \left\lceil \log_{\frac{kM}{B}} \left(\frac{n}{kM} \frac{kM}{B} \right) \right\rceil = \left\lceil \log_{\frac{kM}{B}} \left(\frac{n}{B} \right) \right\rceil$. The cost for each level is at most $(k+1)\lceil n/B \rceil$ reads and $\lceil n/B \rceil$ writes (only one block on each level might not be full). □

4.2 Sample Sort

We now describe an l-way randomized sample sort [10, 19] (also called distribution sort), which asymptotically matches the I/O bounds of the mergesort. The idea of sample sort is to partition n records into l approximately equally sized buckets based on a sample of the keys within the records, and then recurse on each bucket until an appropriately-sized base case is reached. As with the mergesort, here we will use a branching factor $l = kM/B$. Again this branching factor will reduce the number of levels of recursion relative to the standard EM sample sort which uses $l = M/B$ [2]. We describe how to process each partition and the base case.

The partitioning starts by selecting a set of splitters. This can be done using standard techniques, which we review later. The splitters partition the input into buckets that w.h.p. are within a constant factor of the average size n/l. The algorithm now needs to bucket the input based on the splitters. The algorithm processes the splitters in k rounds of size M/B each, starting with the first M/B splitters. For each round the algorithm scans the whole input array, partitioning each value into the one of M/B buckets associated with the splitters, or skipping a record if its key does not belong in the current buckets. One block for each bucket is kept in memory. Whenever a block for one of the buckets is full, it is written out to memory and the next block is started for that bucket. Each k rounds reads all of the input and writes out only the elements associated with these buckets (roughly a $1/k$ fraction of the input).

The base case occurs when $n \leq kM$, at which point we apply the selection sort from Lemma 4.2.

Let n_0 be the original input size. The splitters can be chosen by randomly picking a sample of keys of size $m = \Theta(l \log n_0)$, sorting them, and then sub-selecting the keys at positions $m/l, 2m/l, \ldots, (l-1)m/l$. By selecting the constant in the Θ sufficiently large, this process ensures that, w.h.p., every bucket is within a constant factor of the average size [10]. To sort the samples apply a RAM mergesort, which requires at most $O(((l \log n_0)/B) \log(l \log n_0/M))$ reads and writes. This is a lower-order term when $l = O(n / \log^2 n)$, but unfortunately this bound on l may not hold for small subproblems. There is a simple solution—when $n \leq k^2M^2/B$, instead use $l = n/(kM)$. With this modification, we always have $l \leq \sqrt{n/B}$.

It is likely that the splitters could also be selected deterministically using an approach used in the original I/O-efficient distribution sort [2].

THEOREM 4.4. *The kM/B-way sample sort sorts n records using, w.h.p.,*

$$R(n) = O\left(\frac{kn}{B} \left\lceil \log_{\frac{kM}{B}} \left(\frac{n}{B} \right) \right\rceil \right)$$

reads, and

$$W(n) = O\left(\frac{n}{B} \left\lceil \log_{\frac{kM}{B}} \left(\frac{n}{B} \right) \right\rceil \right)$$

writes on an AEM with primary memory size $(M + B + M/B)$.

PROOF. (Sketch) The primary-memory size allows one block from each bucket as well as the M/B splitters to remain in memory. Each partitioning step thus requires $\lceil n/B \rceil + kM/B$ writes, where the second term arises from the fact that each bucket may use a partial block. Since $n \geq kM$ (this is not a base case), the cost of each partitioning step becomes $O(n/B)$ writes and $O(kn/B)$ reads. Because the number of splitters is at most $\sqrt{n} = O(n/\log^2 n)$, choosing and sorting the splitters takes $O(n/B)$ reads and writes. Observe that the recursive structure matches that of a sample sort with an effective memory of size kM, and that there will be at most two rounds at the end where $l = n/(kM)$. As in standard sample sort, the number of writes is linear with the size of the subproblem, but here the number of reads is multiplied by a factor of k. The standard samplesort analysis thus applies, implying the bound stated.

It remains only to consider the base case. Because all buckets are approximately the same size, the total number of leaves is $O(n/B)$—during the recursion, a size $n > kM$ problem is split into subproblems whose sizes are $\Omega(B)$. Applying Lemma 4.2 to all leaves, we get a cost of $O(kn/B)$ reads and $O(n/B)$ writes for all base cases. □

Extensions for the Private-Cache Model. The above can be readily parallelized. Here we outline the approach. We assume that there are $p = n/M$ processors. We use parallelism both within each partition, and across the recursive partitions. Within a partition we first find the l splitters in parallel. (As above, $l = kM/B$ except for the at most two rounds prior to the base case where $l = n/(kM)$.) This can be done on a sample that is a logarithmic factor smaller than the partition size, using a less efficient sorting algorithm such as parallel mergesort, and then sub-selecting l splitters from the sorted order. This requires $O(k(M/B + \log^2 n))$ time, where the second term ($O(k \log^2 n)$) is the depth of the parallel mergesort, and the first term is the work term $O((k/B)((n/\log n) \log n)/P) = O(kM/B)$.

The algorithm groups the input into $n/(kM)$ chunks of size kM each. As before we also group the splitters into k rounds of size

M/B each. Now in parallel across all chunks and across all rounds, partition the chunk based on the round. We have $n/(kM) \times k = n/M$ processors so we can do them all in parallel. Each will require kM reads and M writes. To ensure that the chunks write their buckets to adjacent locations (so that the output of each bucket is contiguous) we will need to do a pass over the input to count the size of each bucket for each chunk, followed by a prefix sum. This can be done before processing the chunks and is a lower-order term. The time for the computation is $O(kM/B)$.

The processors are then divided among the sub-problems proportional to the size of the sub-problem, and we repeat. The work at each level of recursion remains the same, so the time at each level remains the same. For the base case of size $\leq kM$, instead of using a selection sort across all keys, which is sequential, we find k splitters and divide the work among k processors to sub-select their part of the input, each by reading the whole input, and then sorting their part of size $O(M)$ using a selection sort on those keys. This again takes $O(kM/B)$ time. The total time for the algorithm is therefore:

$$O\left(k\left(\frac{M}{B} + \log^2 n\right)\left\lceil 1 + \log_{\frac{kM}{B}}\left(\frac{n}{kM}\right)\right\rceil\right)$$

with high probability. This is linear speedup assuming $\frac{M}{B} \geq \log^2 n$. Otherwise the number of processors can be reduced to maintain linear speedup.

4.3 I/O Buffer Trees

This section describes how to augment the basic buffer tree [4] to build a priority queue that supports n INSERT and DELETE-MIN operations with an amortized cost of $O((k/B)(1 + \log_{kM/B} n))$ reads and $O((1/B)(1 + \log_{kM/B} n))$ writes per operation. Using the priority queue to implement a sorting algorithm trivially results in a sort costing a total of $O((kn/B)(1 + \log_{kM/B} n))$ reads and $O((n/B)(1 + \log_{kM/B} n))$ writes. These bounds asymptotically match the preceding sorting algorithms, but some additional constant factors are introduced because a buffer tree is a dynamic data structure.

Our buffer tree-based priority queue for the AEM contains a few differences from the regular EM buffer tree [4]: (1) the buffer tree nodes are larger by a factor k, (2) consequently, the "buffer-emptying" process uses an efficient sort on kM elements instead of an in-memory sort on M elements, and (3) to support the priority queue, $O(kM)$ elements are stored outside the buffer tree instead of $O(M)$, which adds nontrivial changes to the data structure.

4.3.1 Overview of a buffer tree

A buffer tree [4] is an augmented version of an (a, b)-tree [22], where $a = l/4$ and $b = l$ for large branching factor l. In the original buffer tree $l = M/B$, but to reduce the number of writes we instead set $l = kM/B$. 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 $O(1 + \log_l n)$. An internal node with c children contains $c - 1$ keys, stored in sorted order, that partition the elements in the subtrees. The structure of a buffer tree differs from that of an (a, b) tree in two ways. Firstly, each leaf of the buffer tree contains between $lB/4$ and lB elements stored in l blocks.[2] Secondly, each node in the buffer tree also contains a dense unsorted list, called a ***buffer***, of partially inserted elements that belong in that subtree.

We next summarize the basic buffer tree insertion process [4]. Supporting general deletions is not much harder, but to implement a priority queue we only need to support deleting an entire leaf. The insertion algorithm proceeds in two phases: the first phase moves elements down the tree through buffers, and the second phase performs the (a, b)-tree rebalance operations (i.e., splitting nodes that are too big). The first phase begins by appending the new element to the end of the root's buffer. We say that a node is ***full*** if its buffer contains at least lB elements. If the insert causes the root to become full, then a ***buffer-emptying process*** commences, whereby all of the elements in the node's buffer are sorted then distributed to the children (appended to the ends of their buffers). This distribution process may cause children to become full, in which case they must also be emptied. More precisely, the algorithm maintains a list of internal nodes with full buffers (initially the root) and a separate list of leaves with full buffers. The first phase operates by repeatedly extracting a full internal node from the list, emptying its buffer, and adding any full children to the list of full internal or leaf nodes, until there are no full internal nodes.

Note that during the first phase, the buffers of full nodes may far exceed lB, e.g., if all of the ancestors' buffer elements are distributed to a single descendant. Sorting the buffer from scratch would therefore be too expensive. Fortunately, each distribution process writes elements to the child buffers in sorted order, so all elements after the lB'th element (i.e., those written in the most recent emptying of the parent) are sorted. It thus suffices to split the buffer at the lB'th element and sort the first lB elements, resulting in a buffer that consists of two sorted lists. These two lists can trivially be merged as they are being distributed to the sorted list of children in a linear number of I/O's.

When the first phase completes, there may be full leaves but no full internal nodes. Moreover, all ancestors of each full leaf have empty buffers. The second phase operates on each full leaf one at a time. First, the buffer is sorted as above and then merged with the elements stored in the leaf. If the leaf contains $X > lB$ elements, then a sequence of (a, b)-tree rebalance operations occur whereby the leaf may be split into $\Theta(X/(lB))$ new nodes. These splits cascade up the tree as in a typical (a, b)-tree insert.

4.3.2 Buffer tree with fewer writes

To reduce the number of writes, we set the branching factor of the buffer tree to $l = kM/B$ instead of $l = M/B$. The consequence of this increase is that the buffer emptying process needs to sort $lB = kM$ elements, which cannot be done with an in-memory sort. The advantage is that the height of the tree reduces to $O(1+\log_{kM/B} n)$.

LEMMA 4.5. *It costs $O(kX/B)$ reads and $O(X/B)$ writes to empty a full buffer containing X elements using $\Theta(M)$ memory.*

PROOF. By Lemma 4.2, the cost of sorting the first kM elements is $O(k^2M/B)$ reads and $O(kM/B)$ writes. The distribute step can be performed by simultaneously scanning the sorted list of children along with the two sorted pieces of the buffer, and outputting to the end of the appropriate child buffer. A write occurs only when either finishing with a child or closing out a block. The distribute step thus uses $O(kM/B + X/B)$ reads and writes, giving a total of $O(k^2M/B + X/B)$ reads and $O(kM/B + X/B)$ writes including the sort step. Observing that full means $X > kM$ completes the proof. □

[2]Arge [4] defines the "leaves" of a buffer tree to contain $\Theta(B)$ elements instead of $\Theta(lB)$ elements. Since the algorithm only operates on the parents of those "leaves", we find the terminology more convenient when flattening the bottom two levels of the tree. Our leaves thus correspond to what Arge terms "leaf nodes" [4] (not to be confused with leaves) or equivalently what Sitchinava and Zeh call "fringe nodes" [34].

THEOREM 4.6. *Suppose that the partially empty block belonging to the root's buffer is kept in memory. Then the amortized cost of each insert into an n-element buffer tree is $O((k/B)(1 + \log_{kM/B} n))$ reads and $O((1/B)(1 + \log_{kM/B} n))$ writes.*

PROOF. This proof follows from Arge's buffer tree performance proofs [4], augmented with the above lemma. We first consider the cost of reading and writing the buffers. The last block of the root buffer need only be written when it becomes full, at which point the next block must be read, giving $O(1/B)$ reads and writes per insert. Each element moves through buffers on a root-to-leaf path, so it may belong to $O(1 + \log_{kM/B} n)$ emptying processes. According to Lemma 4.5, emptying a full buffer costs $O(k/B)$ reads and $O(1/B)$ writes per element. Multiplying these two gives an amortized cost per element matching the theorem.

We next consider the cost of rebalancing operations. Given the choice of (a, b)-tree parameters, the total number of node splits is $O(n/(lB))$ [4, Theorem 1] which is $O(n/(kM))$. Each split is performed by scanning a constant number of nodes, yielding a cost of $O(kM/B)$ reads and write per split, or $O(n/(kM) \cdot kM/B) = O(n/B)$ reads and writes in total or $O(1/B)$ per insert. □

4.3.3 An efficient priority queue with fewer writes

The main idea of Arge's buffer tree-based priority queue [4] is to store a working set of the $O(lB)$ smallest elements resident in memory. When inserting an element, first add it to the working set, then evict the largest element from the working set (perhaps the one just inserted) and insert it into the buffer tree. To extract the minimum, find it in the working set. If the working set is empty, remove the $\Theta(lB)$ smallest elements from the buffer tree and add them to the working set. In the standard buffer tree, $l = M/B$ and hence operating on the working set is free because it fits entirely in memory. In our case, however, extra care is necessary to maintain a working set that has size roughly k times larger.

Our AEM priority queue follows the same idea except the working set is partitioned into two pieces, the alpha working set and beta working set. The ***alpha working set***, which is always resident in memory, contains at most $M/4$ of the smallest elements in the priority queue. The ***beta working set*** contains at most $2kM$ of the next smallest elements in the data structure, stored in $O(kM/B)$ blocks. The motivation for having a beta working set is that during DELETE-MIN operations, emptying elements directly from the buffer tree whenever the alpha working set is empty would be too expensive—having a beta working set to stage larger batches of such elements leads to better amortized bounds. Coping with the interaction between the alpha working set, the beta working set, and the buffer tree, is the main complexity of our priority queue. The beta working set does not fit in memory, but we keep a constant number of blocks from the beta working set and the buffer tree (specifically, the last block of the root buffer) in memory.

We begin with a high-level description of the priority-queue operations, with details of the beta working set deferred until later. For now, it suffices to know that we keep the maximum key in the beta working set in memory. To insert a new element, first compare its key against the maximums in the alpha and beta working set. Then insert it into either the alpha working set, the beta working set, or the buffer tree depending on the key comparisons. If the alpha working set exceeds maximum capacity of $M/4$ elements, move the largest element to the beta working set. If the beta working set hits its maximum capacity of $2kM$ elements, remove the largest kM elements and insert them into the buffer tree.

To delete the minimum from the priority queue, remove the smallest element from the alpha working set. If the alpha working set is empty, extract the $M/4$ smallest elements from the beta working set (details to follow) and move them to the alpha working set. If the beta working set is empty, perform a buffer emptying process on the root-to-leftmost-leaf path in the buffer tree. Then delete the leftmost leaf and move its contents to the beta working set.

The beta working set. The main challenge is in implementing the beta working set. An unsorted list or buffer allows for efficient inserts by appending to the last block. The challenge, however, is to extract the $\Theta(M)$ smallest elements with $O(M/B)$ writes—if $k > B$, each element may reside in a separate block, and we thus cannot afford to update those blocks when extracting the elements. Instead, we perform the deletions implicitly.

To facilitate implicit deletions, we maintain a list of ordered pairs $(i_1, x_1), (i_2, x_2), (i_3, x_3), \ldots$, where (i, x) indicates that all elements with index at most i and key at most x are invalid. Our algorithm maintains the invariant that for consecutive list elements (i_j, x_j) and (i_{j+1}, x_{j+1}), we have $i_j < i_{j+1}$ and $x_j > x_{j+1}$ (recall that all keys are distinct).

To insert an element to the beta working set, simply append it to the end. The invariant is maintained because its index is larger than any pair in the list.

To extract the minimum $M/4$ elements, scan from index 0 to i_1 in the beta working set, ignoring any elements with key at most x_1. Then scan from $i_1 + 1$ to i_2, ignoring any element with key at most x_2. And so on. While scanning, record in memory the $M/4$ smallest valid elements seen so far. When finished, let x be the largest key and let i be the length of the beta working set. All elements with key at most x have been removed from the full beta working set, so they should be implicitly marked as invalid. To restore the invariant, truncate the list until the last pair (i_j, x_j) has $x_j > x$, then append (i, x) to the list. Because the size of the beta working set is growing, $i_j < i$. It should be clear that truncation does not discard any information as (i, x) subsumes any of the truncated pairs.

Whenever the beta working set grows too large ($2kM$ valid elements) or becomes too sparse (k extractions of $M/4$ elements each have occurred), we first rebuild it. Rebuilding scans the elements in order, removing the invalid elements by packing the valid ones densely into blocks. Testing for validity is done as above. When done, the list of ordered pairs to test invalidity is cleared.

Finally, when the beta working set grows too large, we extract the largest kM elements by sorting it (using the selection sort of Lemma 4.2).

Analyzing the priority queue. We begin with some lemmas about the beta working set.

LEMMA 4.7. *Extracting the $M/4$ smallest valid elements from the beta working set and storing them in memory costs $O(kM/B)$ reads and amortized $O(1)$ writes.*

PROOF. The extraction involves first performing read-only passes over the beta working set and list of pairs, keeping one block from the working set and one pair in memory at a time. Because the working set is rebuilt after k extractions, the list of pairs can have at most k entries. Even if the list is not I/O efficient, the cost of scanning both is $O(kM/B + k) = O(kM/B)$ reads. Next the list of pairs indicating invalid elements is updated. Appending one new entry requires $O(1)$ writes. Truncating and deleting any old entries can be charged against their insertions. □

The proof of the following lemma is similar to the preceding one, with the only difference being that the valid elements must be moved and written as they are read.

LEMMA 4.8. *Rebuilding the beta working set costs $O(kM/B)$ reads and writes.* □

THEOREM 4.9. *Our priority queue, if initially empty, supports n INSERT and DELETE-MIN operations with an amortized cost of $O((k/B)(1+\log_{kM/B} n))$ reads and $O((1/B)(1+\log_{kM/B} n))$ writes per operation.*

PROOF. Inserts are the easier case. Inserting into the alpha working set is free. The amortized cost of inserting directly into the beta working set (a simple append) is $O(1/B)$ reads and writes, assuming the last block stays in memory. The cost of inserting directly into the buffer tree matches the theorem. Occasionally, the beta working set overflows, in which case we rebuild it, sort it, and insert elements into the buffer tree. The rebuild costs $O(kM/B)$ reads and writes (Lemma 4.8), the sort costs $O(k^2M/B)$ reads and $O(kM/B)$ writes (by Lemma 4.2), and the kM buffer tree inserts cost $O((k^2M/B)(1+\log_{kM/B} n))$ reads and $O((kM/B)(1+\log_{kM/B} n))$ writes (by Theorem 4.6). The latter dominates. Amortizing against the kM inserts that occur between overflows, the amortized cost per insert matches the theorem statement.

Deleting the minimum element from the alpha working set is free. When the alpha working set becomes empty, we extract $M/4$ elements from the beta working set, with a cost of $O(kM/B)$ reads and $O(1)$ writes (Lemma 4.7). This cost may be amortized against the $M/4$ deletes that occur between extractions, for an amortized cost of $O(k/B)$ reads and $O(1/M)$ writes per delete-min. Every k extractions of $M/4$ elements, the beta working set is rebuilt, with a cost of $O(kM/B)$ reads and writes (Lemma 4.8) or amortized $O(1/B)$ reads and writes per delete-min. Adding these together, we so far have $O(k/B)$ reads and $O(1/B)$ writes per delete-min.

It remains to analyze the cost of refilling the beta working set when it becomes empty. The cost of removing a leaf from the buffer tree is dominated by the cost of emptying buffers on a length-$O(\log_{kM/B} n)$ path. Note that the buffers are not full, so we cannot apply Lemma 4.5. But a similar analysis applies. The cost per node is $O(k^2M/B + X/B)$ reads and $O(kM/B + X/B)$ writes for an X-element buffer. As with Arge's version of the priority queue [4], the $O(X/B)$ terms can be charged to the insertion of the X elements, so we are left with a cost of $O(k^2M/B)$ read and $O(kM/B)$ writes per buffer. Multiplying by $O(1+\log_{kM/B} n)$ levels gives a cost of $O((k^2M/B)(1+\log_{kM/B} n))$ reads and $O((kM/B)(1+\log_{kM/B} n))$ writes. Because each leaf contains at least $kM/4$ elements, we can amortize this cost against at least $kM/4$ deletions, giving a cost that matches the theorem. □

With this priority queue, sorting can be trivially implemented in $O((kn/B)(1+\log_{kM/B} n))$ reads and $O((n/B)(1+\log_{kM/B} n))$ writes, matching the bounds of the previous sorting algorithms.

5. CACHE-OBLIVIOUS PARALLEL ALGORITHMS

In this section we present low-depth cache-oblivious parallel algorithms for sorting and Fast Fourier Transform, with asymmetric read and write costs. 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 using the results of Section 2. We also present a linear-depth, cache-oblivious parallel algorithm for matrix multiplication. All three algorithms use $\Theta(k)$ fewer writes than reads.

5.1 Sorting

We show how the low-depth, cache-oblivious sorting algorithm from [9] 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 and does extra reads to reduce the number of levels of recursion. The algorithm does $O((n/B)\log_{kM}(kn))$ writes, $O((kn/B)\log_{kM}(kn))$ reads, and has depth $O\big(k\log^2(n/k)\big)$ w.h.p.

The algorithm uses matrix transpose, prefix sums and mergesort as subroutines. Efficient parallel and cache-oblivious versions of these algorithm are described in [9]. For an input of size n, prefix sums has depth $O(k\log n)$ and requires $O(n/B)$ reads and writes, merging two arrays of lengths n and m has depth $O(k\log(n+m))$ and requires $O((n+m)/B)$ reads and writes, and mergesort has depth $O\big(k\log^2 n\big)$ and requires $O((n/B)\log_2(n/M))$ reads and writes. Transposing an $n \times m$ matrix has depth $O(k\log(n+m))$ and requires $O(nm/B)$ reads and writes.

Our cache-oblivious sorting algorithm works recursively, with a base case of $n \le M$, at which point any parallel sorting algorithm with $O(n\log n)$ reads/writes and $O(k\log n)$ depth can be applied (e.g. [14]).

Figure 1 illustrates the steps of the algorithm. Given an input array of size n, the algorithm first splits it into $\sqrt{nk}$ subarrays of size $\sqrt{n/k}$ and recursively sorts each of the subarrays. This step corresponds to step (a) in Figure 1.

Then the algorithm determines the splitters by sampling. After the subarrays are sorted, every $(\log n)$'th element from each row is sampled, and these $n/\log n$ samples are sorted using a cache-oblivious mergesort. Then $\sqrt{n/k}-1$ evenly-distributed splitters are picked from the sorted samples to create $\sqrt{n/k}$ buckets. The algorithm then determines the boundaries of the buckets in each subarray, which can be implemented by merging the splitters with each row, requiring $O(k\log n)$ depth and $O(n/B)$ writes overall. This step is shown as step (b) in Figure 1. Notice that on average the size of each bucket is $O\Big(\sqrt{nk}\Big)$, and the largest bucket has no more than $2\sqrt{nk}\log n$ elements.

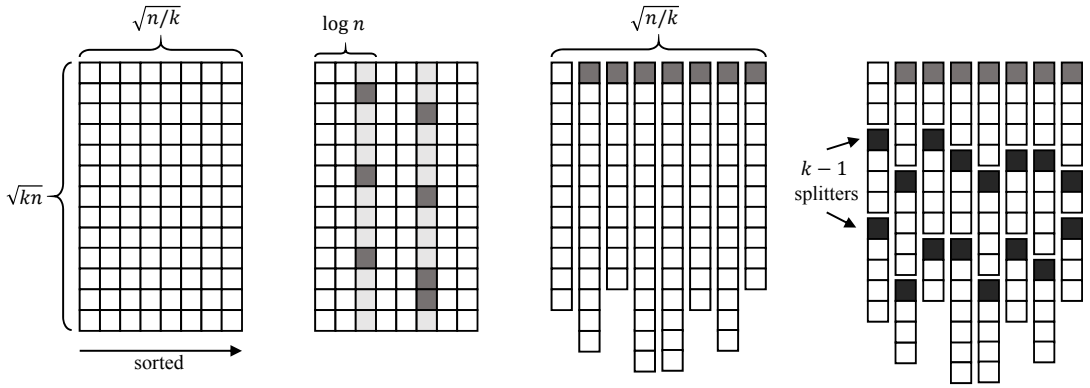
After the subarrays are split into $\sqrt{n/k}$ buckets, prefix sums and a matrix transpose can be used to place all keys destined for the same bucket together in contiguous arrays. This process is illustrated as step (c) in Figure 1. This process requires $O(n/B)$ reads and writes, and $O(k\log n)$ depth.

The next step is new to the asymmetric algorithm and is the part that requires extra reads. As illustrated in Figure 1 (d), $k-1$ pivots are chosen from each bucket to generate k sub-buckets. We sample $\max\{k, \sqrt{kn}/\log n\}$ samples, apply a mergesort, and evenly pick $k-1$ pivots in the sample. Then the size of each sub-bucket can be shown to be $O\Big(\sqrt{n/k}\log n\Big)$ w.h.p. using Chernoff bounds. We then scan each bucket for k rounds to partition all elements into k sub-buckets, and sort each sub-bucket recursively.

THEOREM 5.1. *Our cache-oblivious sorting algorithm requires $O((kn/B)\log_{kM}(kn))$ reads, $O((n/B)\log_{kM}(kn))$ writes, and $O\big(k\log^2(n/k)\big)$ depth w.h.p.*

PROOF. All the subroutines except for the recursive calls do $O(n/B)$ writes. The last partitioning process to put elements into sub-buckets takes $O(kn/B)$ reads and the other subroutines require fewer reads. The overall depth is dominated by the mergesort to find the first $\sqrt{n/k}$ pivots, requiring $O\big(k\log^2(n/k)\big)$ depth per level of recursion. Hence, the recurrence relations (w.h.p.) for read I/O's (R), write I/O's (W), and depth (D) are:

$$R(n) = O\left(\frac{kn}{B}\right) + \sqrt{kn}\cdot R\left(\sqrt{\frac{n}{k}}\right) + \sum_{i=1}^{\sqrt{kn}} R(n_i)$$



(a) The input is split into $\sqrt{kn}$ subarrays, and the subarrays are sorted recursively.

(b) $\sqrt{n/k}$ splitters (grey) are picked from $n/\log n$ samples (light grey).

(c) All elements are put into buckets by prefix-sums and matrix transpose operations.

(d) Another $k-1$ splitters (black) are picked in each bucket so that elements are logically placed into smaller sub-buckets. Each sub-bucket is then sorted recursively.

Figure 1: The low-depth cache-oblivious algorithm on asymmetric read and write costs to sort an input array of size n.

$$W(n) = O\Big(\frac{n}{B}\Big) + \sqrt{kn} \cdot W\left(\sqrt{\frac{n}{k}}\right) + \sum_{i=1}^{\sqrt{kn}} W(n_i)$$

$$D(n) = O\Big(k \log^2 \frac{n}{k}\Big) + \max_{i=1,\cdots,\sqrt{kn}} \{D(n_i)\}$$

where n_i is the size of the i'th sub-bucket, and $n_i = O\Big(\sqrt{n/k} \log n\Big)$ and $\sum n_i = n$. The base case for the I/O complexity is $R(M) = W(M) = O(M/B)$. Solving these recurrences proves the theorem. □

5.2 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 [15], and our description follows that of [20]. We assume that n at each level of recursion and k are powers of 2. The standard cache-oblivious divide-and-conquer algorithm [20] views the input matrix as an $\sqrt{n} \times \sqrt{n}$ matrix, and incurs $O((n/B) \log_M(n))$ reads and writes. To reduce the number of writes, we modify the algorithm to the following:

1. View input of size n as a $k\sqrt{n/k} \times \sqrt{n/k}$ matrix in row-major order. Transpose the matrix.
2. Compute a DFT on each row of the matrix as follows
   (a) View the row as a $k \times \sqrt{n/k}$ matrix
   (b) For each row i
      i. Calculate the values of column DFTs for row i using the brute-force method. This will require k reads (each row) and 1 write (row i) per value.
      ii. Recursively compute the FFT for the row.
   (c) Incorporate twiddle factors
   (d) Transpose $k \times \sqrt{n/k}$ matrix
3. Transpose matrix
4. Recursively compute an FFT on each length $\sqrt{n/k}$ row.
5. Transpose to final order.

The difference from the standard cache-oblivious algorithms is the extra level of nesting in step 2, and the use of a work-inefficient DFT over k elements in step 2(b). The transposes in steps 1, 2(d) and 3 can be done with $O(n/B)$ reads/writes and $O(k \log n)$ depth. The brute-force calculations in step 2(b)i require a total of kn reads (and arithmetic operations) and n writes. This is where we waste a k factor in reads in order to reduce the recursion depth. The twiddle factors can all be calculated with $O(n)$ reads and writes. There are a total of $2k\sqrt{n/k}$ recursive calls on problems of size $\sqrt{n/k}$.

Our base case is of size M, which uses M/B reads and writes. The number of reads therefore satisfies the following recurrence:

$$R(n) = \begin{cases} O(n/B) & \text{if } n \le M \\ 2k\sqrt{n/k}\; R\Big(\sqrt{n/k}\Big) + O(kn/B) & \text{otherwise} \end{cases}$$

which solves to $R(n) = O((kn/B) \log_{kM}(kn))$, and the number of writes is

$$W(n) = \begin{cases} O(n/B) & \text{if } n \le M \\ 2k\sqrt{n/k}\; W\Big(\sqrt{n/k}\Big) + O(n/B) & \text{otherwise} \end{cases}$$

which solves to $W(n) = O((n/B) \log_{kM}(kn))$.

Since we can do all the rows in parallel for each step, and the brute-force calculation in parallel, the only important dependence is that we have to do step 2 before step 5. This gives a recurrence $D(n) = 2D\Big(\sqrt{n/k}\Big) + O(k \log n)$ for the depth, which solves to $O(k \log n \log\log n)$.

We note that the algorithm as described requires an extra transpose and an extra write in step 2(b)i relative to the standard version. This might negate any advantage from reducing the number of levels, however we note that these can likely be removed. In particular the transpose in steps 2(d) and 3 can be merged by viewing the results as a three dimensional array and transposing the first and last dimensions (this is what the pair of transposes does). The write in step 2(b)i can be merged with the transpose in step 1 by combining the columnwise FFT with the transpose, and applying it k times.

5.3 Matrix Multiplication

In this section, we consider matrix multiplication in the asymmetric read/write setting. The standard cubic-work sequential algorithm trivially uses $O\big(n^3\big)$ reads and $\Theta\big(n^2\big)$ writes, one for each entry in the output matrix. For the EM model, the blocked algo-

rithm that divides the matrix into sub-matrices of size $\sqrt{M} \times \sqrt{M}$ uses $O\left(n^3/(B\sqrt{M})\right)$ reads [11, 20]. Because we can keep each $\sqrt{M} \times \sqrt{M}$ sub-matrix of the output matrix in memory until it is completely computed, the number of writes is proportional to the number of blocks in the output, which is $O\left(n^2/B\right)$. This gives the following simple result:

THEOREM 5.2. *External-memory matrix multiplication can be done in $O\left(n^3/(B\sqrt{M})\right)$ reads and $O\left(n^2/B\right)$ writes.*

We now turn to a divide-and-conquer algorithm for matrix multiplication, which is parallel and cache-oblivious. We assume that we can fit $3M$ elements in cache instead of M, which only affects our bounds by constant factors. Note that the standard cache-oblivious divide-and-conquer algorithm [11, 20] 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 $\Theta(k)$, we instead recurse on k^2 parallel subproblems (blocks) of size $n/k \times n/k$. On each level of recursion, computing each output block of size $n/k \times n/k$ requires summing over the k products of $n/k \times n/k$ size input matrices. We assume the recursive calls writing to the same target block are processed sequentially so that all writes (and reads) can be made at the leaves of the recursion to their final locations.

For the purpose of analysis we consider a base case when the problem is of size $k\sqrt{M} \times k\sqrt{M}$. At this point each of its subproblems of size $\sqrt{M} \times \sqrt{M}$ is writing into an output block of size M, which fits in cache. Therefore since we do the products for the output blocks sequentially, the result can stay in cache and only be written when all k are done. The number of writes is therefore M/B per output block and k^2M/B total. For reads all of the k^3 subproblems might require reading both inputs so there are $2k^3M/B$ reads. The non-base recursive calls do not contribute any significant reads or writes since all reading and writing to the arrays is done at the leaves. This gives us the following recurrence for the number of writes for an $n \times n$ matrix:

$$W(n) = \begin{cases} k^2M/B & \text{if } n < k\sqrt{M} \\ k^3W(n/k) + O(1) & \text{otherwise} \end{cases}$$

This solves to $W(n) = O\left(n^3/(kB\sqrt{M})\right)$, which is a factor of k less than for the standard EM or cache-oblivious matrix multiply.[3] The number of reads satisfies:

$$R(n) = \begin{cases} 2k^3M/B & \text{if } n < k\sqrt{M} \\ k^3R(n/k) + O(1) & \text{otherwise} \end{cases}$$

This solves to $R(n) = O\left(n^3/(B\sqrt{M})\right)$, which is the same as for the standard EM or cache-oblivious matrix multiply.

Because the products contributing to a block are done sequentially, the depth of the algorithm satisfies the recurrence $D(n) = kD(n/k)+O(1)$ with base case $D(1) = k$, which solves to $O(kn)$. This gives the following theorem:

THEOREM 5.3. *Our cache-oblivious matrix multiplication algorithm requires $O\left(n^3/(B\sqrt{M})\right)$ reads, $O\left(n^3/(kB\sqrt{M})\right)$ writes, and $O(kn)$ depth.*

[3]Note that for this analysis we assume the initial problem is of size $n = k^i\sqrt{M}$ for some integer i.

6. CONCLUSION

Motivated by the high cost of writing relative to reading in emerging non-volatile memory technologies, we have considered a variety of models for accounting for read-write asymmetry, and proposed and analyzed a variety of sorting algorithms in these models. For the asymmetric RAM and PRAM models, we have shown how to reduce the number of writes from $O(n \log n)$ to $O(n)$ without asymptotically increasing the other costs (reads, parallel depth). For the asymmetric external memory models (including the cache-oblivious model) the reductions in writes are mostly more modest, often increasing the base of a logarithm, and at the cost of asymptotically more reads. However, these algorithms might still have practical benefit. We also presented a cache-oblivious matrix-multiply that asymptotically reduces the writes by a factor of k while not asymptotically increasing reads. Future work includes proving lower bounds for the asymmetric external memory models, devising write-efficient algorithms for additional problems, and performing experimental studies.

Acknowledgments

This research was supported in part by NSF grants CCF-1218188, CCF-1314633, and CCF-1314590, and by the Intel Science and Technology Center for Cloud Computing.

7. REFERENCES

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

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *CACM*, 31(9), 1988.

[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.

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

[5] L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *SPAA*, 2008.

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

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

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

[9] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low-depth cache oblivious algorithms. In *SPAA*, 2010.

[10] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *SPAA*, 1991.

[11] 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.

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

[13] S. Cho and H. Lee. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *MICRO*, 2009.

[14] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4), 1988.

[15] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19, 1965.

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

[17] 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.

[18] 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.

[19] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM*, 17(3), 1970.

[20] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, 1999.

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

[22] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17, 1982.

[23] www.slideshare.net/IBMZRL/theseus-pss-nvmw2014, 2014.

[24] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[25] 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.

[26] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *ISCA*, 2009.

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

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

[29] T. Ottmann and D. Wood. How to update a balanced binary tree with a constant number of rotations. In *SWAT*, 1990.

[30] H. Park and K. Shim. FAST: flash-aware external sorting for mobile database systems. *Journal of Systems and Software*, 82(8), 2009.

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

[32] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3), 1989.

[33] J. H. Reif and S. Sen. Parallel computational geometry: An approach using randomization. In J. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, chapter 18. Elsevier Science, 1999.

[34] N. Sitchinava and N. Zeh. A parallel buffer tree. In *SPAA*, 2012.

[35] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2), 1985.

[36] S. Viglas. Write-limited sorts and joins for persistent memory. *PVLDB*, 7(5), 2014.

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

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

[39] B.-D. Yang, J.-E. Lee, J.-S. Kim, J. Cho, S.-Y. Lee, and B.-G. Yu. A low power phase-change random access memory using a data-comparison write scheme. In *ISCAS*, 2007.

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

[41] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *ISCA*, 2009.

[42] O. Zilberberg, S. Weiss, and S. Toledo. Phase-change memory: An architectural perspective. *ACM Computing Surveys*, 45(3), 2013.