Algorithmic Building Blocks for Asymmetric Memories

Yan Gu
Carnegie Mellon University, Pittsburgh, PA, USA
yan.gu@cs.cmu.edu

Yihan Sun
Carnegie Mellon University, Pittsburgh, PA, USA
yihans@cs.cmu.edu

Guy E. Blelloch
Carnegie Mellon University, Pittsburgh, PA, USA
guyb@cs.cmu.edu

Abstract

The future of main memory appears to lie in the direction of new non-volatile memory technologies that provide strong capacity-to-performance ratios, but have write operations that are much more expensive than reads in terms of energy, bandwidth, and latency. This asymmetry can have a significant effect on algorithm design, and in many cases it is possible to reduce writes at the cost of more reads. This paper studies which algorithmic techniques are useful in designing practical write-efficient algorithms. We focus on several fundamental algorithmic building blocks including unordered set/map implemented using hash tables, comparison sort, and graph traversal algorithms including breadth-first search and Dijkstra's algorithm. We introduce new algorithms and implementations that can reduce writes, and analyze the performance experimentally using a software simulator. Finally, we summarize interesting lessons and directions in designing write-efficient algorithms that can be valuable to share.

2012 ACM Subject Classification Theory of computation → Models of computation, Theory of computation → Design and analysis of algorithms

Keywords and phrases Asymmetric Memory, I/O Cost, Write-Efficient Algorithms, Hash Tables, Graph-Traversal Algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2018.44

Related Version The full version is available at [20], `https://arxiv.org/abs/1806.10370`.

Acknowledgements This work was supported by NSF grants CCF-1408940, CCF-1533858, CCF-1629444 and CCF-1745331.

1 Introduction

The future of main memory appears to lie in the non-volatile memory technologies that promise persistence, significantly lower energy costs, and higher density than the DRAM technology used in today's main memories [21, 24, 33, 43]. However, despite the advantages, a key property of such memory technologies is their asymmetric read-write costs: compared to reads, writes can be much more expensive in terms of latency, bandwidth, and energy. Because bits are stored in these technologies as at rest "states" of the given material that can be quickly read but require physical change to update, this asymmetry appears fundamental.

This motivates the need for *write-efficient* algorithms that largely reduce the number of writes compared to existing algorithms.

In the related work section, we review the literature on studying this read-write asymmetry on NAND Flash chips [4, 17, 18, 35] and algorithms targeting database operators [12, 39, 40]. These works provide novel aspects on rethinking algorithm design. However, most of the papers either treat NVMs as external memories, or are based on hardware simulators for existing architecture, which may have many concerns that we will further discuss in the related work section.

Blelloch et al. [5, 7, 8] formally defined and analyzed several sequential and parallel computation models with good caching and scheduling guarantees. The models abstract such asymmetry between reads and writes, and can be used to analyze algorithms on future memory. The basic model, which is the Asymmetric RAM (ARAM), extends the well-known external-memory model [1] and parameterizes the asymmetry using ω, which corresponds to the cost of a write relative to a read to the non-volatile main memory. The cost of an algorithm on the ARAM, the **asymmetric I/O cost**, is the number of write transfers to the main memory multiplied by ω, plus the number of read transfers. This model captures different system consideration (latency, bandwidth, or energy) by simply plugging in different values of ω, and also allows algorithms to be analyzed theoretically. Based on this idea, many interesting algorithms (and lower bounds) are designed and analyzed by various recent papers [5, 6, 7, 8, 10, 25].

Unfortunately, all of the analyses of such write-efficient algorithms are asymptotic, showing the upper and lower bounds on the complexity of these problems. Also, to prove the bounds, the theoretical models simplify the real architecture (e.g., without considering blocking of cache-lines or cache policies). It still remains unknown what the performance of these algorithms are in practice. In this paper, our goal is to show such performance on a number of fundamental algorithmic building blocks. We believe the lessons in designing and implementing them are useful for our community to use new memory in the future.

Contribution of this paper

In this work, our goal is to bridge the gap between theory and practice. We try to study and understand which algorithmic techniques are useful in designing practical write-efficient algorithms. As the first paper of this kind, we focus on several of the most commonly-seen algorithmic building blocks in modern programming. Due to the page limit, in this paper we briefly discuss unordered set/map implemented using **hash tables**, and graph traversal algorithms: **breadth-first search** for unweighted graphs and **Dijkstra's algorithm** for weighted graphs. In the full version of this work, we discuss more details of these algorithms, as well as ordered set/map implemented using **binary search trees** and **comparison sort**.

Unfortunately, no non-volatile main memory is currently available, making it impossible to get real timings. Furthermore, details about latency and other parameters of the memory and how they will be incorporated into the architecture are also not available. This makes detailed cycle-level simulation (e.g., PTLsim [36], MARSSx86 [34] or ZSim [38]) of questionable utility. However, it is quite feasible to count the number of reads and write to main memory while simulating a variety of cache configurations. For I/O-bounded algorithms, these numbers can be used as reasonable proxies for both running time (especially when implemented in parallel) and energy consumption.[1] Moreover, conclusions drawn from these numbers can likely give insights into tradeoffs between reads and writes among different algorithms.

[1] The energy consumption of main memory is a key concern since it costs 25-50% energy on data centers and servers [28, 32, 30].

For these reasons, we propose a framework based on a software simulator that can efficiently and precisely measure the number of read and write transfers of an algorithm using different caching policies. We also consider variants in caching policies that might lead to improvements when read and write are not the same.

We also note that designing write-efficient algorithms falls in a high dimensional parameter space since the asymmetries on latency, bandwidth, and energy consumption between reads and writes are different. Here we abstract this as a single value ω. This value together with the cache size M and cache-line size B (set to be 64 bytes in this paper) form the parameter space of an algorithm.

Our framework provides a simple, clean and hardware-independent method to analyze and experiment the performance on the asymmetric memory. We investigate the algorithmic techniques and learn lessons from the experiments that generally apply for a reasonably large parameter space of ω, M and B. This framework also allows monitoring, reasoning and debugging the code easily, so it can remain useful even after the new hardware is available.

With the framework, we design, implement and discuss many algorithms and data structures and their write-efficient implementations. Although some of the implementations are standard, like quicksort and hash tables, many others, including k-level hash tables, sample sort and phased Dijkstra, require careful algorithmic design, analysis, and coding. Under our measurement which is the asymmetric I/O cost and compared to the most commonly-used ones on symmetric memories, we provide better alternatives to all problems we studied in this paper.

With the algorithms and their experimental results, we draw many interesting algorithmic strategies and guidance in designing write-efficient algorithms. A common theme is to trade (more) reads for (fewer) writes (apparently it is hard to directly decrease the writes since this can improve the performance on symmetric memory as well and should have been investigated already). Some interesting lessons we learned and can be valuable to share are listed as follows, which can suggest some potential directions to design and engineer write-efficient algorithms in the future.

1. Indirect addressing is less problematic. In the classic setting, indirect addressing should be avoided if possible, since each addressing can be a random access to the memory. However, when writes are expensive, moving the entire data is costly, while indirect addressing only modifies the pointers (at the cost of a possible random access per lookup).
2. Multiple candidate positions for a single entry in a data structure can help. It can be a good option to use more reads per lookup but apply less frequent data movements, when the size of a data structure changes significantly. This is a common strategy we have applied in this paper to provide an algorithmic tradeoff between reads and writes.
3. It is usually worth to investigate existing algorithms that move or modify the data less. These algorithms can be less efficient in the symmetric setting due to various reasons (e.g., more random accesses, less balanced), but the property that they use fewer writes can be useful in the asymmetric setting (like samplesort vs. quicksort, treap vs. AVL or red-black tree).
4. In-cache data structures should draw more attention. Since the data structures are kept in the cache (or small symmetric memory), the algorithm requires significantly less writes to the large asymmetric memory, although may require extra reads to compensate for less information we can keep within the data structure. In this paper, we discuss Dijkstra's algorithm on shortest-paths as an example, and such idea can also be applied to computing minimum spanning tree, sorting, and many other problems.

2 Related Work

There exist a rich literature to show the read-write asymmetry on the new memories [2, 3, 7, 8, 11, 13, 15, 16, 22, 23, 26, 27, 31, 37, 41, 42, 44, 45]. Regarding adapting softwares for such read-write asymmetry, some work has studied the system aspect. For example, there exist many papers on how to balance the writes across the chip to avoid uneven wear-out of locations in the context of NAND Flash chips [4, 17, 18, 35].

The early and inspirational attempts to design algorithms with fewer writes targeting database operators: Chen et al. [12] and Viglas [39, 40] presented several write-efficient sequential algorithms for searching, hash joins and sorting. However, their results are mainly shown by assuming external memories rather than main memories, or on the cycle-based simulators for existing architecture. For the latter case however, the prototypes of the new memories are still under development, and yet nobody actually knows the exact parameters of the new memories, or how they are incorporated into the actual architecture. As a result, we believe that the results based on cycle-based simulator might not be very accurate. In the meantime, the asymmetries on latency, bandwidth, and energy consumption between reads and writes are different, and any of these constraints can be the bottleneck of an algorithm. Hence, designing algorithms on asymmetric memory are in a multiple-dimension parameter space, rather than just recording the running time from a simulator. Therefore, it is essential to develop theoretical models and tools that account for, and abstract this asymmetry and use them to analyze algorithms on future memory.

Blelloch et al. [5, 7, 8] formally defined several sequential and parallel computation models that take asymmetric read-write costs into account. Based on the computational models, many interesting algorithms (and lower bounds) are designed and analyzed in both sequential and parallel settings, which includes sorting, permuting, matrix multiplication, FFT, list/tree contraction, BFS/DFS and other graph algorithms, and many computational geometric and dynamic programming problems [5, 6, 7, 8, 10, 25, 9, 19]. Carson et al. [11] also presented write-efficient sequential algorithms for a similar model, as well as write-efficient parallel algorithms (and lower bounds) on a distributed memory model with asymmetric read-write costs, focusing on linear algebra problems and direct N-body methods. Although many problems under the asymmetric setting have been studied, all the analyses are asymptotic and only show the upper and lower bounds on the complexity of these problems.

3 Our Model and Simulator

To start with, we discuss how to measure the performance of algorithms on asymmetric memories. We begin with the computational model that estimates the cost of an algorithm. This model requires the numbers of read and write transfers between the non-volatile memory and the cache, so later we introduce how the numbers of an algorithm can be simulated.

The Cost Model for Asymmetric Memory. The most commonly-used cost measure of an algorithm is the time complexity based on the RAM model, which is the overall number of instructions and memory accesses executed in this algorithm. Nowadays, since the latency of an memory access is at least two orders of magnitudes more expensive than a CPU instruction, the *I/O cost* based on the external-memory model [1] is widely used to analyze the cost of an I/O-bounded algorithm. This model assumes a *small-memory* (cache) of size $M \geq 1$, and a unbounded-size *large-memory*. Both memories are organized in blocks (cache-lines) of B words. The CPU can only access the small-memory (with no cost), and it takes unit

cost to transfer one block between the small-memory and the large-memory. This cost measure estimates the running time reasonably well for I/O-bounded algorithms, especially in multi-core parallelism. An efficient algorithm in practice should achieve optimality in both time complexity and I/O cost.

To account for more expensive writes on future memories, here we adopt the idea of an (M, ω)-Asymmetric RAM (ARAM) [8]: similar to the external-memory model, transferring a block from large-memory to small-memory takes unit cost; on the other direction, the cost is either 0 if this block is clean and never modified, or $\omega \gg 1$ otherwise. The **asymmetric I/O cost Q** of an algorithm is the overall costs for all memory transfers. We abbreviate such cost Q as the *I/O cost* throughout the paper, unless stated otherwise explicitly. Theoretical results on this new model have been studied in [5, 6, 7, 8, 10, 25, 9, 19].

Cache Policies. Either the classic external-memory model or the new ARAM assumes that we can explicitly manipulate the cache in the algorithm. This largely simplifies the analysis, and in many cases is provably within a constant factor of a more realistic cache's performance. For example, the standard least-recent used (LRU) policy is 2-competitive against the optimal offline cache-replacement sequence. However, the competitive ratio does not hold in the asymmetric setting in the worst case. The overhead is proportional to ω, which can be significant and problematic. In the full version of this paper [20], we discuss several alternative solutions with worst-case performance guarantees. In this conference version we show our experiment results based on the LRU policy, and the comparison to other policies are covered in the full version of this paper.

The Cache Simulator. To capture the number of reads and writes to the main memory, we developed a software simulator that can adapt to different cache policies. The cache simulator is composed of an ordered map that keeps tracks of the time stamp of the last visit to each cache-line in the current cache, and an unordered map that stores the mapping from each cache-line to the corresponding location in the ordered map if this cache-line is currently in the cache. Interestingly, the implementation of this cache simulator is a natural application of the techniques discussed in this paper.

The cache simulator encapsulates a new structure Array that is used in coding algorithms in this paper. It is like a regular array that can be dynamically allocated and freed, and supports two functions: Read and Write to a specific location in this array. The Arrays are responsible for reporting the memory accesses of the algorithm to the cache simulator, and the cache simulator will update the state of the cache accordingly. Therefore, coding using the Arrays is not different from regular programming much.

The memory accesses to loop variables and temporary variables are ignored, as well as the call stack. This is because the number of such variables is small in all of the algorithms in this paper (usually no more than 10). Meanwhile, the call stack of all algorithms in this paper has size $O(\log n)$. The overall amount of uncaptured space is orders of magnitudes smaller than the amount of fast memory in our experiments.

The cache simulator maintains two counters: the number of **read transfers**, and the number of **write transfers**. When testing each algorithm on a specific input instance, the cache is emptied at the beginning and flushed at the end. A read or write is free if the location is already in the cache; otherwise, the corresponding cache-line is loaded, the counter of read transfer increments by 1, and the least-recently-used cache-line in this pool is evicted. Also, a write will mark the dirty-bit of the cache-line to be `true`. When evicting a dirty cache-line, the counter of write transfer increments by 1. Notice that memory reads can cause write transfers, and memory writes can lead to read transfers.

When simulating the **Classic** policy (i.e., the standard one), we also verified our simulated results to ZSim (cycle-level simulator for current architecture), and the numbers always differ by no more than 10% when the parameters are set correctly.

4 Unordered Sets and Maps

Sets and maps are two of the most commonly-used data types in modern programming. Most programming languages either have them built in as basic types (e.g., python) or supply them as standard libraries (C++, C#, Java, Scala, Haskell, ML). In this section, we discuss efficient implementations of unordered sets and maps implemented using hash tables.

Our implementation of unordered sets and maps is based on hash tables that support **lookup**, **insertion**, and **deletion**. The hash tables discussed in this section use open addressing and linear probing, since the goal of the data structure is to try to minimize the I/O cost focusing on smaller entries (accessing and reading larger entries are costly anyway so different hash-table implementations make minor differences). For simplicity, we assume no duplicate keys, and it is straightforward to handle the duplicates with minor modifications. In this setting, each operation of the hash table reads a small number of cache-lines, and an insertion or deletion will modify exactly one cache-line that contains the location of the key and will be eventually written back to the large-memory.

The challenge emerges when the set size changes dynamically. For an efficient implementation, we hope the overall size of the hash table to be neither too large nor too small. If the load factor passes 80%, linear probing's performance drastically degrades. On the other hand, we want the hash table size to be reasonably small to better utilize the small-memory (cache), since each cache-line holds more entries in this case. In practice, some implementations keep the load factor up- and lower-bounded by some constant. For example, a typical implementation keeps the occupancy of the hash table between 1/8 and 1/2, and the size doubles or shrinks by half if the number of entries exceeds this range. Such resizing reinserts p entries after at least $p/2$ insertions and deletions (where p is the set/map size). When reads and writes have approximately the same cost, the extra cost for such resizing is small compared to the query and update costs (e.g., the queries read from lots of memory locations). In the asymmetric setting however, the reads cost much less, but the extra writes in resizing can be significant: the resizing can incur at most twice $(p/(p/2) = 2)$ the writes compared to the initial insertions ($3\times$ writes in total). Hence, our goal is to discuss an alternative approach that optimizes such extra writes.

4.1 The k-level Hash Table

Instead of keeping one hash table, our main idea is to maintain a small number k of hash tables simultaneously, where k is a pre-determined parameter. In particular, the k-level hash table *HashTable* is initialized with k arrays $HashTable_{1,\cdots,k}$ with size $2^{c'+i}$ for $1 \leq i \leq k$ (or smaller in specific applications) and a constant c'. In practice we set c' to be 5.

For insertions, when the overall load factor exceeds some threshold r, we allocate a new chunk of memory with the double size of the largest current array, and the smallest hash table is discarded after all elements in it have been reinserted back. Similarly for deletions, if the occupancy of the hash tables drops below a threshold l, a small array with half size of the current smallest hash table is allocated, and the largest table is freed after the entries in it being reinserted. For instance, a valid k-level hash table may contain two arrays of size $2^{15} = 32768$ and $2^{16} = 65536$, when $k = 2$ and 30000 entries in the current configuration. We show the pseudocode of the k-level hash table in Algorithm 1. The occupancy range

Algorithm 1: The k-level hash table.

Input: Parameter k, occupancy range l and r

```
1  function Lookup(x)
2      for i ← 1 to k do
3          p ← HashTable_i.Lookup(x)
4          if p ≠ null then return (i, p)
5      return null
6  function Insert(x) // x is not in HashTable
7      for i ← 1 to k do
8          if HashTable_i.occupancy < r then
9              HashTable_i.Insert(x)
10             return
11     Allocate HashTable_{k+1} of size 2 · HashTable_k.size
12     Relabel the hash tables with indices from 0 to k
13     foreach y ∈ HashTable_0 do
14         Insert(y)
15     Free HashTable_0
16 function Delete(x; i, p) // x is located p-th in HashTable_i
17     HashTable_i.Delete(x, p)
18     if Overall occupancy is less than l (and HashTable_1.size > 1) then
19         Allocate HashTable_0 of size HashTable_1.size/2
20         Relabel the hash tables with indices between 1 to k + 1
21         foreach y ∈ HashTable_{k+1} do
22             Insert(y)
23         Free HashTable_{k+1}
```

$0 < l < r < 1$ indicates when the resizing happens (an example of l and r can be 1/8 and 1/2). A classic implementation can be viewed as the special case of the k-level hash table when $k = 1$.

We now analyze the I/O cost Q of the k-level hash table. Here we assume that the size of the k-level hash table is larger than the small-memory and $1 - r < 1/B$, so on average, one lookup, insertion or deletion in a single level in the hash table requires no more than $c < 2$ cache-line loads to locate the position.

Lookup. In a k-level hash table, a lookup requires ck instead of c read transfers (c is the constant just defined) in the worst case (can quit earlier once the entry is found). The cost increases by a factor of k at most.

Insert. There are two definitions of insertions: an insertion that the key is known to be not in the set/map, or an insertion that it is unknown whether the key is in this set/map. Both cases are commonly-used. In this paper, we take the first definition and analyze the cost of this type of insertions. The second type of insertion can be viewed as a lookup first, then an insert if the lookup fails.

When inserting an element in a k-level hash table, we always try the smaller tables first. Once all tables are full, we resize it. More details can be found in Algorithm 1.

The I/O cost Q of an insertion comes in two parts: the cost of the initial insertion to the hash table, and the cost of this entry in future hash-table resizings. The cost of the initial insertion is no more than $c + \omega$, where c is the number of cache-line reads to find the position to insert, plus ω, one cache-line write for the actual insertion. The cost of resizing is more complicated to analyze.

We note that although a specific entry can be reinserted multiple times during different resizing processes, the overall number of element reinsertion is bounded, and thus we can a amortize the work. A resizing occurs when an insertion comes in and the hash table contains exactly $r \cdot 2^p(2^k - 1)$ elements for some positive integer p. In this case, at most $r \cdot 2^p$ entries (the size of the smallest hash table), are reinserted during the resizing. The total number of insertions from the last resizing is at least $r \cdot 2^{p-1}(2^k - 1)$ (assuming $4l \leq r$), so the amortized I/O cost Q of reinsertion for each insertion is upper bounded by $\frac{(c+\omega)r \cdot 2^p}{r \cdot 2^{p-1}(2^k-1)} = (c+\omega) \cdot 2/(2^k - 1)$.

In the asymmetric setting when $\omega \gg 1$, the I/O cost of each insertion is approximately $\omega \cdot (1 + 2/(2^k - 1))$, indicating that compared to the classic implementation where $k = 1$, in the worst-case the improvement when $k = 2, 3, 4$ is about 44%, 57% and 62% respectively. The asymptotic improvement when $k \to +\infty$ is 67% ($\frac{2}{3}$).

Delete. A deletion in the k-level hash table is similar to an insertion except that a lookup for the location is required (details in Algorithm 1). The cost of the initial deletion is $ck + \omega$. A resizing of the hash table can occur after at least $l \cdot 2^p(2^k - 1)$ deletions for some positive integer p, and the current hash table keeps $l \cdot 2^p(2^k - 1)$ entries. However, it is possible that all of these entries are in the last hash table so they are all reinserted. We note that when reinserting the elements from the discarded array, we always try smaller arrays first. This means that a reinserted entry, if not being deleted in the future, will not be reinserted again in the next $\min(k - 1, \log_2 r/2l)$ shrinking resizings. Namely, the amortized extra cost of a deletion in future resizings is about ω/k if l is set to be about $2^{-k}r$. The overall I/O cost for a deletion is $Q = ck + \omega(1 + 1/k)$.

We have bounded of the I/O cost of each lookup, insertion or deletion, and the overall cost Q can be estimated by summing the amount of each operation multiplied by the cost of this operation. In practice, insertions and deletions can interleave. For example, when a deletion comes after an insertion, the number of entries remains the same, which leads to no further cost for these two updates afterward. The exact cost is also affected by the pattern of the sequence of the operations, and we will show by experiments.

4.2 Experiments

We provide the full experiment of our k-level hash table in the full version of this paper [20]. We test the performance on various update/query patterns, and report the numbers of read transfers and write transfers, as well as I/O costs. We also justify our result by comparing to the wall-clock running time (in the full paper [20]). Due to the space limit, in this conference version we only show one of the experiments here that contains insertions and queries.

In all experiments, we insert 1 million elements to an empty hash table. Each of the element is a 4-byte integer, and we vary the number of queries. The simulated cache contains 10,000 cache-lines. The occupancy rate is set to be $l = 0.2$ and $r = 0.8$. We have tried other parameters (r between 0.6 and 0.8 and $l = r/4$). The results slightly vary, but all general conclusions in this section still hold.

Many applications, like webpage caching or the breadth-first searches, only insert but never delete elements in a hash table. Our experiment starts with this simpler case. We first show the relationship between k (the number of hash tables) and the numbers of read transfers and write transfers for a variety of insertion/query ratios, and the results are shown in Table 1. We fix the number of insertions to be one million, and query α times after each insertion. We vary α from 0, 1/8, to 8 ($\alpha < 1$ indicates one query per $1/\alpha$ insertions). About

Table 1 Numbers of read and write transfers of the k-level hash tables with different query/insert ratios. Numbers of read and write transfers are divided by 10^6 (i.e., per insertion).

10^6 insertions, $\alpha \times 10^6$ queries where α is from 0 to 8, the cache contains 10,000 cache-lines.

α	0		1/8		1/4		1/2		1		2		4		8	
	RT	WT	RT	WT	RT	WT	RT	WT	RT	WT	RT	WT	RT	WT	RT	WT
k=1	1.35	1.17	1.44	1.18	1.52	1.19	1.69	1.21	2.02	1.24	2.68	1.27	4.00	1.31	6.64	1.34
k=2	0.85	0.79	1.06	0.84	1.23	0.87	1.54	0.91	2.09	0.96	3.11	1.00	5.07	1.03	8.94	1.05
k=3	0.76	0.72	1.08	0.80	1.32	0.85	1.73	0.90	2.44	0.95	3.76	0.99	6.31	1.02	11.32	1.05
k=4	0.70	0.67	1.11	0.78	1.40	0.82	1.89	0.88	2.74	0.93	4.30	0.97	7.33	1.00	13.31	1.03

Table 2 The I/O costs of the k-level hash tables with different query/insert ratios. The write-read ratio ω are selected to be typical projected values 10 (latency, bandwidth) and 100 (energy). Results are based on the numbers in Table 1. The numbers in red with underlines indicate the best choice of k that minimizes the I/O cost in this setting, and numbers in blue indicate better I/O costs compared to the classic hash table implementation (i.e., $k = 1$).

The I/O costs of the k-level hash tables with the same configurations in Table 1.

	$\omega = 10$								$\omega = 100$							
α	0	1/8	1/4	1/2	1	2	4	8	0	1/8	1/4	1/2	1	2	4	8
k=1	13.0	13.2	13.4	13.8	14.4	15.4	17.1	20.0	117.9	119.3	120.5	122.7	125.8	129.9	134.8	140.5
k=2	8.8	9.5	10.0	10.7	11.7	13.1	15.4	19.5	79.9	85.1	88.4	92.8	97.7	102.9	108.2	114.4
k=3	8.0	9.1	9.8	10.7	11.9	13.7	16.5	21.8	73.1	81.4	85.8	91.3	97.0	102.7	108.7	116.3
k=4	7.4	8.9	9.6	10.7	12.0	14.0	17.4	23.6	67.9	78.6	83.8	89.6	95.5	101.3	107.7	116.1

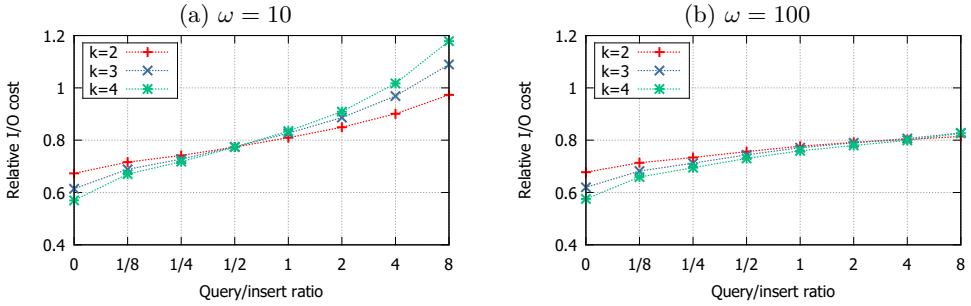

Figure 1 Relative I/O cost of k-level hash table with different values of k. The I/O costs are divided by the $k = 1$ case, so every data point below 1 indicates an improvement in such case. Numbers are from Table 2.

50% query keys are in the hash table (this ratio affects the I/O cost since a successful query can terminate earlier). The number of levels k varies from 1 to 4. In Table 2, we show the overall I/O costs, which are the weighted sums assuming two typical values of the write-read ratio ω, 10 and 100.

We first look at the number of write transfers. When there is no query (i.e., the first column, just inserting 1 million entries), the numbers of writes are consistent with our analysis for insertions in Section 4.1. The only exception here is that cache can hold a constant fraction of the elements, which batches the writes and reduces the number of memory transfers. However, the relative trend in each column remains unchanged. Namely, the number of writes always decreases with the increase of k regardless of the ratio between queries and updates. The number of writes is reduced by 33%, 40% and 43% when $k = 2, 3, 4$ respectively. Such improvement also shows up in the overall I/O cost in Table 2.

We note that more queries cause more reads, and larger k also leads to more reads. Since these reads flush the cache-lines, the numbers of writes in these cases also marginally increase. The optimal choice of k is decided by the update/query distribution as well as the write-read ratio ω. In general, more queries lead to worse performance with larger k, and larger ω prefers larger k. In Table 2, we underline the numbers indicating the best choice of k in that specific setting. The experiment results indicate that picking k to be 2 or 3 is always a good choice when $\omega = 10$, and 3 or 4 when $\omega = 100$.

4.3 Conclusions

We proposed a new data structure, the k-level hash table, to implement unordered set and map that has the same space utilization compared to the classic open-addressing hash tables. The key idea is to keep multiple instead of one level of hash tables. As a result, the algorithm uses fewer writes during resizings, at the cost of more reads in other operations.

The best choice of k is decided by the ratio of updates and queries. Our experiment shows that $k = 2$ always leads to a lower or similar I/O cost when the query/insert ratio is no more than 8, compared to the classic $k = 1$ setting. For the ratio of write/read cost is larger (like 100), larger values of k, like 3 or 4, are even more preferable than the $k = 2$ case.

5 Graph Traversal Algorithms

In this paper, we discuss two of the most commonly-used graph traversal algorithms: breadth-first search (BFS), and Dijkstra's algorithm. We show that using the new implementations discussed in this paper, these algorithms use much fewer writes in most cases, compared to the classic ones. Due to the page limit, we abstract our approaches and conclusions here in this section, and provide the full details in the full version of this paper [20].

Given a graph $G = (V, E)$, we assume $n = |V|$ is the number of vertices, and $m = |E|$ is the number of edges.

5.1 Breadth-First Search

We discuss our implementations and experiment results on breadth-first searches (BFS) on undirected graph traversing or searching. Our algorithms compute the single-source shortest paths (SSSP) or pairwise shortest-paths (given the specific source and target) on unweighted graphs, which can further apply to graph radii estimation, eccentricity estimation and betweenness centrality, and act as a basic building block for other graph algorithms like graph connectivity, reachability, biconnected components, and strongly connected components.

Implementations. The classic implementation of BFS keeps a vertex queue of size n, and an array of boolean flags of size n indicating whether each vertex is visited or not during the search. This implementation requires at most 2 writes per vertex, and the overall I/O cost of BFS $Q(n, m) = O(\omega n + m)$ [8]. This bound is asymptotically optimal for arbitrary graphs since the output size of BFS is $\Theta(n)$. However, a number of applications (e.g., s-t shortest-path or connectivity, graph radii estimation or eccentricity estimation) have output size $O(1)$, which allows utilizing the small-memory and reducing the number of writes.

The key observation to improve the write-efficiency is that, at any time, we only need the information of three consecutive frontiers (a frontier is the set of vertices with the same distance to the source node). We hence use the k-level hash table discussed in Section 4 to implement the frontiers. This avoids the writes to mark the visited flag of each vertex. We

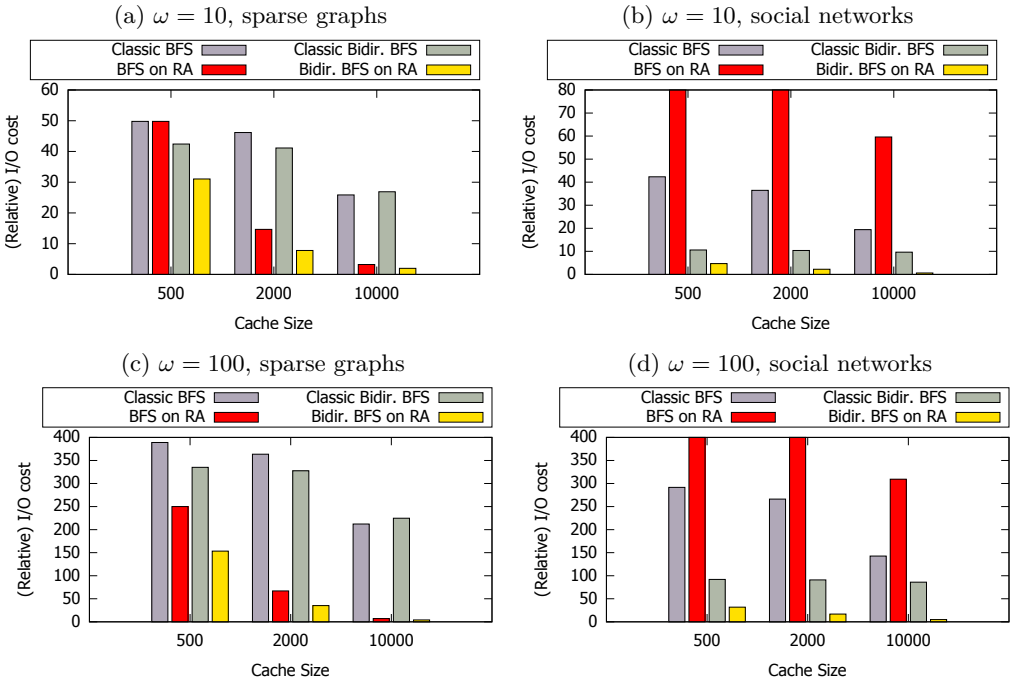

Figure 2 The trends of the I/O costs of four different implementations of BFS. The new implementations shown in this paper are the BFS and bidirectional BFS based on rotating arrays (red and yellow bars). The graphs used in the experiment are shown in full paper [20] and categorized into sparse (almost planar) graphs and social networks. We show the relative I/O cost based on varied cache sizes, and each number is geometric mean of the four graphs in that category. We can see the consistent advantages of the new BFS implementation on sparse graphs, and the improvement of the new bidirectional version in all cases. Notice that in (b) and (d) some values exceed the ranges of vertical axis.

note that once the frontier size fits into the small-memory, the algorithm does not require any writes to traverse the newly visited vertices. In the full paper [20] we show the average and maximum frontier sizes of the experiment graphs, which will help to understand the performance on these graph instances. To the best of our knowledge, we are unaware of any graph invariant to capture and predict the average and maximum frontier sizes, and we believe that it can be an interesting topic for further study. This algorithm is referred to as the *BFS on RA* (rotating arrays) in Figure 2, and more details of the implementation are given in the full paper [20].

The previous algorithm works well on graphs with larger diameters, but not on small-diameter graphs like social networks. We then introduce the bidirectional version (*Bidir. BFS on RA*) when the queries are s-t (pairwise) shortest paths, that overcomes the disadvantages of the previous algorithm. More analysis and details of this version are given in full version.

Experiment. Our experiment is based on eight graphs that are synthesized or from SNAP [29]. We show a significant improvement on all eight graphs with various cache sizes, compared to the classic queue-based implementations. Figure 2 is the bar charts that show the trends of the four implementations. When $\omega = 10$, our implementation shows an up-to 8-fold improvement on SSSP, and an up-to 43-fold improvement on s-t shortest-paths. For $\omega = 100$, the improvement is more stable, which is 69 on SSSP and 71 on s-t shortest-paths.

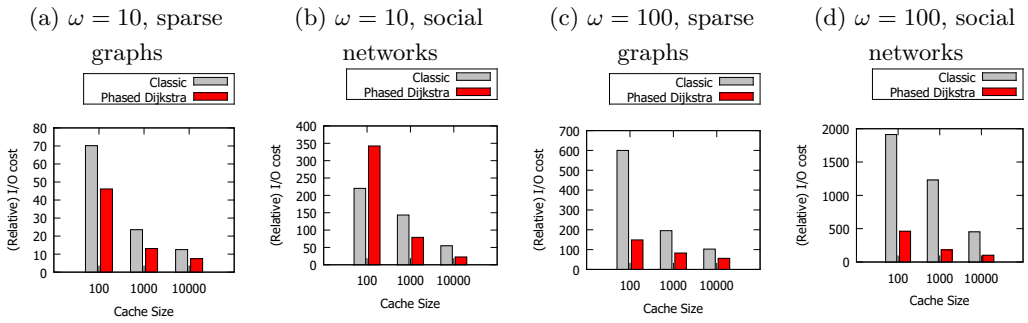

Figure 3 The trends of the I/O costs of classic Dijkstra (grey) and phased Dijkstra (red) on different graphs with varied cache sizes. The graphs used in the experiment are shown in the full paper [20] and categorized into sparse (almost planar) graphs and social networks. Each number of the I/O cost is geometric mean of the four graphs in that category. Phased Dijkstra performs consistently better in all cases except when both the cache size and the asymmetry ω are small.

Conclusions. We discuss how to efficiently implement BFS in the asymmetric setting and experiment the I/O performance for four implementations on a variety of undirected graphs. We show that for s-t (pairwise) distance queries, our bidirectional BFS using rotating arrays shows a significant advantage in all cases we tested. For single-source shortest-paths, the unidirectional BFS using rotating arrays has a significant improvement when the cache can hold every single frontier during the search.

5.2 Dijkstra's Algorithm

Dijkstra's Algorithm [14] computes single-source shortest paths on non-negative weighted graphs. The classic heap-based implementation requires $O(m \log(nB/M))$ reads and writes.

For the sake of write-efficiency, Blelloch et al. [8] discussed a variant called *Phased Dijkstra*. This algorithm only requires linear writes, but the algorithm is just explained at a high level and without much details. In this paper, we supplement the pseudocode, data structure design, and implementation details (in the full paper [20]).

Based on our implementation, we conduct various experiment to show the asymmetric I/O efficiency. In Figure 3, we show the trend of the relative I/O costs of the classic Dijkstra and phased Dijkstra with various cache sizes on different graphs. We show that phased Dijkstra outperforms classic Dijkstra in most cases except for the only case on social networks with very small cache size, and the improvement on I/O cost in all cases is up to 3 and 7.6 when ω is 10 and 100. We also consider various cache policies and sets of parameters and show that the improvement is consistent among all settings.

Summaries. We discuss phased Dijkstra and test its performance on a variety of graphs. The high-level idea is to fit the heap of Dijkstra's algorithm within the small-memory (i.e., the cache) and thus the algorithm applies no intermediate writes to the large asymmetry memory to maintain the heap. The extra cost is that, the algorithm is run in multiple phases each requiring $O(m)$ reads, but we show that such price is worthwhile in most cases. The experiments show that phased Dijkstra consistently outperforms the binary-heap version on I/O costs, except for the combination of small ω ($= 10$), small cache size, and on social networks. Although phased Dijkstra contains several parameters, we also show that they can be chosen from a reasonably wide range and do not affect the superiority of phased Dijkstra. The same conclusion also holds for different cache policies.

We note that the idea of fitting the data structure or the computation in the small-memory can also be applied to computing minimum spanning tree, sorting, and many other problems.

References

1. Alok Aggarwal and Jeffrey S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9), 1988. doi:10.1145/48529.48535.
2. Ameen Akel, Adrian M. Caulfield, Todor I. Mollov, Rajech K. Gupta, and Steven Swanson. Onyx: A prototype phase change memory storage array. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2011.
3. Manos Athanassoulis, Bishwaranjan Bhattacharjee, Mustafa Canim, and Kenneth A. Ross. Path processing using solid state storage. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, 2012.
4. Avraham Ben-Aroya and Sivan Toledo. Competitive analysis of flash-memory algorithms. In *European Symposium on Algorithms (ESA)*, 2006.
5. Naama Ben-David, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. Parallel algorithms for asymmetric read-write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.
6. Naama Ben-David, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. Implicit decomposition for write-efficient connectivity algorithms. In *Proc. IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2018.
7. 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.
8. Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. Efficient algorithms with asymmetric read and write costs. In *European Symposium on Algorithms (ESA)*, pages 14:1–14:18, 2016.
9. Guy E Blelloch, Phillip B Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. The parallel persistent memory model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.
10. Guy E Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallel write-efficient algorithms and data structures for computational geometry. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2018.
11. Erin Carson, James Demmel, Laura Grigori, Nicholas Knight, Penporn Koanantakool, Oded Schwartz, and Harsha Vardhan Simhadri. Write-avoiding algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 648–658, 2016.
12. Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *Conference on Innovative Data Systems Research (CIDR)*, 2011.
13. Sangyeun Cho and Hyunjin Lee. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.
14. Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1), 1959.
15. Xiangyu Dong, Norman P. Jouupi, and Yuan Xie. PCRAMsim: System-level performance, energy, and area modeling for phase-change RAM. In *ACM International Conference on Computer-Aided Design (ICCAD)*, 2009.
16. Xiangyu Dong, Xiaoxia Wu, Guangyu Sun, Yuan Xie, Hai H. Li, and Yiran Chen. Circuit and microarchitecture evaluation of 3D stacking magnetic RAM (MRAM) as a universal memory replacement. In *ACM Design Automation Conference (DAC)*, 2008.

17 David Eppstein, Michael T. Goodrich, Michael Mitzenmacher, and Pawel Pszona. Wear minimization for cuckoo hashing: How not to throw a lot of eggs into one basket. In *ACM International Symposium on Experimental Algorithms (SEA)*, 2014.

18 Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2), 2005.

19 Yan Gu. *Write-Efficient Algorithms (draft)*. PhD Thesis, 2018.

20 Yan Gu, Yihan Sun, and Guy E. Blelloch. Algorithmic building blocks for asymmetric memories (full version). In *arXiv preprint:1806.10370*, 2018.

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

22 Jingtong Hu, Qingfeng Zhuge, Chun Jason Xue, Wei-Che Tseng, Shouzhen Gu, and Edwin Sha. Scheduling to optimize cache utilization for non-volatile main memories. *IEEE Transactions on Computers*, 63(8), 2014.

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

24 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, 2015.

25 Riko Jacob and Nodari Sitchinava. Lower bounds in the asymmetric external memory model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 247–254, 2017.

26 Hyojun Kim, Sangeetha Seshadri, Clement L. Dickey, and Lawrence Chu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *USENIX Conference on File and Storage Technologies (FAST)*, 2014.

27 Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *ACM International Symposium on Computer Architecture (ISCA)*, 2009.

28 Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W Keller. Energy management for commercial servers. *Computer*, 36(12):39–48, 2003.

29 Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection, 2014.

30 Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *European Conference on Computer Systems*, page 4. ACM, 2014.

31 Jasmina Malicevic, Subramanya Dulloor, Narayanan Sundaram, Nadathur Satish, Jeff Jackson, and Willy Zwaenepoel. Exploiting NVM in large-scale graph analytics. In *Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*. ACM, 2015.

32 Krishna T Malladi, Ian Shaeffer, Liji Gopalakrishnan, David Lo, Benjamin C Lee, and Mark Horowitz. Rethinking DRAM power modes for energy proportionality. In *IEEE/ACM International Symposium on Microarchitecture*, pages 131–142, 2012.

33 Jagan S. Meena, Simon M. Sze, Umesh Chand, and Tseung-Yuan Tseng. Overview of emerging nonvolatile memory technologies. *Nanoscale Research Letters*, 9, 2014.

34 MARSSx86. http://marss86.org.

35 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`.

36 PTLsim. http://www.ptlsim.org.

37 Moinuddin K. Qureshi, Sudhanva Gurumurthi, and Bipin Rajendran. *Phase Change Memory: From Devices to Systems*. Morgan & Claypool, 2011.

38 Daniel Sanchez and Christos Kozyrakis. Zsim: fast and accurate microarchitectural simulation of thousand-core systems. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 475–486. ACM, 2013.

39 Stratis D. Viglas. Adapting the B^+-tree for asymmetric I/O. In *East European Conference on Advances in Databases and Information Systems (ADBIS)*, 2012.

40 Stratis D. Viglas. Write-limited sorts and joins for persistent memory. *VLDB Endowment*, 7(5), 2014.

41 Cong Xu, Xiangyu Dong, Norman P. Jouppi, and Yuan Xie. Design implications of memristor-based RRAM cross-point structures. In *IEEE Design, Automation and Test in Europe (DATE)*, 2011.

42 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 *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2007.

43 Yole Developpement. Emerging non-volatile memory technologies, 2013.

44 Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *ACM International Symposium on Computer Architecture (ISCA)*, 2009.

45 Omer Zilberberg, Shlomo Weiss, and Sivan Toledo. Phase-change memory: An architectural perspective. *ACM Computing Surveys*, 45(3), 2013.