Journal of Systems Architecture 120 (2021) 102279

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

PMSort: An adaptive sorting engine for persistent memory

Yifan Hua, Kaixin Huang, Shengan Zheng, Linpeng Huang [*]

Shanghai Jiao Tong University, China

ARTICLE INFO

Keywords:
Persistent memory
Sorting algorithm
Wear-leveling
Failure recovery

ABSTRACT

Emerging persistent memory (PM, also termed as non-volatile memory) technologies can promise large capacity, non-volatility, byte-addressability and DRAM-comparable access latency. A host of PM-based storage systems and applications that store and access data directly in PM have been inspired by such amazing features. Sorting is an important function for many systems, but how to optimize sorting for PM-based systems has not been systematically studied. In this paper, we conduct extensive experiments for many existing sorting methods, including both conventional sorting algorithms adapted for PM and recently-proposed PM-friendly sorting techniques, on a DRAM and real PM hybrid platform. The results indicate that these sorting methods all have drawbacks for various workloads. Some of the results are even counterintuitive compared to running on a DRAM-simulated platform in their papers. To the best of our knowledge, we are the first to perform a systematic study on the sorting issue for persistent memory. Based on our study, we summarize principles on selecting the optimal algorithms and propose an adaptive sorting engine called PMSort to perform the selecting operation and reduce failure recovery overhead in PM. We conduct extensive experiments and show that PMSort can select the best sorting algorithm in a variety of cases.

1. Introduction

With the emergence of a new type of non-volatile storage device called persistent memory (PM), many unique features including byte-addressability, DRAM-comparable read and write latency have been explored by a host of researches such as memory management systems [1–3], file systems [4–6], KV-Stores [7–9] and DBMSs [10–12] on redesigning persistent storage systems. Unlike traditional SSD, HDD or Flash, PM technologies such as STT-RAM [13], PCM [14] and 3DXPoint [15] enable applications to access data in PM with simple load/store instructions.

Sorting is one of the most commonly-used functions in many storage systems. For instance, the 'ORDER BY' SQL command will automatically call the embedded sorting engine in a DBMS. The sorting component will sort the records in a table by a specified key.[1] While many PM-optimized storage systems have been proposed, only a few works discuss how to optimize sorting for PM. An intuitive thinking is simply to apply conventional sorting algorithms in PM-based systems. However, such a naive migration has many limitations for traditional sorting methods.

For internal sorting algorithms, first, since PM has the limited write endurance [16–18], simply migrating traditional sorting algorithms from DRAM to PM causes heavy write traffic to PM, which will reduce the lifespan of PM. This is mainly caused by the allocated PM space for large-size records and their swap during sorting. Second, long time consumption exists during record swap for large values. For instance, fixing the keys to be 8-byte in size and record number to be one million, sorting the records with 4 KB values will spend 26.1 s using quick sort while the records with 8-byte values only cost 224 ms. Large values also make it hard to exploit cache locality. Third, employing sorting methods directly in PM costs more time than with the assistance of DRAM in some cases (see more details in Section 3).

For external sorting algorithms, data loaded from PM to DRAM for sorting as performed in DRAM-Disk architecture not only consumes large DRAM space, but also takes many runs in the merging phase. First, external sort is heavily dependent on DRAM resource. When the available DRAM space is scarce, it may suffer from frequent data migration between DRAM and PM, which leads to the read/write amplification problem. Second, since disk/SSD is block addressable, sorting records using external sort can only load block-size data from disk to DRAM, which induces heavy time overhead in both loading and sorting phases.

Since it is reported that PM should have much higher write latency than read latency, and PM may suffer from the limited write endurance issue (e.g., PCM is reported to be worn out after $10^6 - -10^8$ writes) [16–18], a few researchers have proposed PM-friendly sorting methods [19–21] to decrease PM writes. For example, segment sort [20] intends to

* Corresponding author.
E-mail address: huang-lp@sjtu.edu.cn (L. Huang).

[1] In this paper, we call the attribute for sorting in a record as `key` and the other attributes as `value`.

https://doi.org/10.1016/j.sysarc.2021.102279
Received 27 April 2021; Received in revised form 29 August 2021; Accepted 11 September 2021
Available online 20 September 2021
1383-7621/© 2021 Published by Elsevier B.V.

trade off fewer writes for additional reads and allows a tunable combination of external sort and selection sort. The reason is that although the read complexity of selection sort is $O(N^2)$, its write complexity is merely $O(N)$. Another work B*-sort [19] develops a binary tree-based structure for sorting records in PM, which has $O(N)$ complexity for writes and $O(NlogN)$ complexity for reads. Luo et al. [21] utilize a heap structure and observe that if a node is close to the heap root, it is more likely to be read and written frequently. Thus, in order to reduce the average writes to PM, nodes close to the root are placed into DRAM while those close to the leaves are placed into PM. All these methods are evaluated on a DRAM-simulated platform and show good experimental results. However, when we run them on the real PM hardware (i.e., Optane platform), their performance is far from the expected result (e.g., much worse than the simple quick sort, a conventional sorting algorithm; see more details in Section 3).

We believe that there are at least three reasons. First, these three PM-friendly sorting methods heavily rely on the assumption that the latency of PM read should be much better than PM write. However, this is not the case for Optane. A recently-published paper [22] shows that Optane's write latency is comparable to DRAM but its read latency is 2x-3x worse than DRAM. Second, they cannot exert the full potential of cache locality, which makes them worse than quick sort in actual execution. For instance, the selection sort used in segment sort has to scan the entire portion each turn. As for B*-sort, it links records with left-child and right-child pointers, and hence all reads and writes are made to be random. For NVMSort, the node swap in the heapify process has to search nodes in both PM and DRAM. Third, they introduce extra PM read and write overhead despite of the relatively lower time complexity. For example, B*-sort allocates PM for all the left-child and right-child pointers, extra tunnel lists and metadata, leading to heavy additional overhead.

The limitations that exist in both conventional sorting methods and PM-friendly sorting methods inspire us to rethink the design of sorting in persistent memory. We first notice that the byte-addressable feature of PM allows us to index records with simple pointers (i.e., data addresses), which is quite different from the DRAM-Disk storage architecture. Based on the unique byte-addressable characteristics of PM, we propose a pointer-indirect mechanism in this paper to speed up the sorting performance for large-size records.

In addition, we have conducted a careful analysis on the tradeoffs for possible applied algorithms from various aspects, including redundant metadata overhead, time complexity of algorithms, wear-leveling performance, cache locality utilization, asymmetry read/write latency characteristics of Optane, whether or not using pointer-indirect mechanisms, large or small value size, requirements of user, etc., as well as experiments to show their performance under these different conditions. Based on the specific study conducted for these commonly-used sorting algorithms, we find that no single sorting method can be the best-level fit (i.e., both time-efficient and memory space-efficient) for different workloads and situations. For instance, although quick sort in PM performs well for many cases, it is worse than external sort for large-size records when the DRAM space is sufficient in a DRAM-PM hybrid memory architecture.[2] External sort, on the contrary, has worse performance when the DRAM size is very small. We have verified these bottlenecks by conducting multiple experiments on a DRAM and the Intel Optane DC Persistent Memory (Optane) hybrid platform (see more details in Section 3).

Based on the analysis that no single sorting method is the best fit for different conditions, we then design and implement an adaptive sorting engine for persistent memory, namely PMSort. PMSort can get complicated users' demands and hardware settings, then automatically make decisions to pick the best-suited embedded sorting method in an ad-hoc style. Compared with single sorting method, PMSort can get benefit from both space and time efficiency. Moreover, we have applied the pointer-indirect mechanism in PMSort algorithm library, which can reduce the write traffic to PM as well as sorting latency. Notice that the pointer indirection in this paper is a user-transparent procedure for speeding up sorting and no matter using pointer-indirect or non-pointer-indirect methods, the result will be finally output as <key, value> (not <key, pointer>) to the user buffer in order, which does not change the sorting library syntax. Moreover, we have carefully analyzed the tradeoff between reshuffle and no-reshuffle when adopting pointer-indirect methods.

In order to eliminate the inconsistency problem and reduce recovery overhead for system crash or power failure, we have carefully designed appropriate recovery methods for all sorting algorithms in PMSort library. Note that the recovery mechanism we design can ensure data consistency for both key and value larger than 8B. The experiments prove the effectiveness of our recovery mechanism for existing sorting methods. Our contributions are summarized as follows.

• To the best of our knowledge, we are the first to make a systematic study for sorting methods in DRAM-PM hybrid architecture. We carefully analyze the tradeoffs of using different sorting algorithms from various aspects on the real PM platform and demonstrate that existing sorting methods have non-negligible limitations.

• Combined with the advantages of various sorting methods, we detail how to choose the best PM sorting algorithms. Based on our analysis on existing sorting algorithms, we summarize some principles on selecting the optimal algorithms and develop an adaptive sorting engine called PMSort to perform the selecting operation in PM for different conditions.

• We have devised proper recovery mechanisms for all sorting algorithms in PMSort algorithm library to ensure data consistency and reduce the reordering overhead for failure recovery.

• We conduct extensive experiments on the Optane platform and the results show that the pointer-indirect mechanism in PMSort reduces the execution time by 56% ∼ 92% compared to its non-pointer counterpart, with orders of magnitude less writes in PM as well; PMSort can always pick the most suitable algorithms for various workloads and situations; our proposed failure recovery method greatly mitigates the reordering overhead.

The rest of this paper is organized as follows. Sections 2 and 3 introduce the background and motivation of our work, respectively. Section 4 presents our proposed pointer-indirect sort mechanism and adaptive sorting engine, namely PMSort, in detail. We evaluate PMSort in Section 5 and discuss some extensions of PMSort in Section 6. In Section 7, we present related work and finally conclude this paper in Section 8.

2. Background

2.1. Persistent Memory

Persistent Memory (PM) such as PCM [14], STT-RAM [13] and 3DX-Point [15] is a new type of memory technology that has large capacity, non-volatility, byte-addressability, and limited write endurance [16, 18]. Attaching PM to the main memory bus provides a raw storage medium that can be orders of magnitude faster than modern persistent storage medium such as disk and SSD [4]. Intel Optane DC Persistent Memory (Optane for short) is the first commercially-available PM product [23]. The read and write bandwidth of Optane are 6.6 GB/s and 2.3 GB/s, respectively. Contrast to the bandwidth, the latency of read for Optane is quite larger than write, which is different from the commonly-believed read/write asymmetry feature (i.e., write latency is much higher than read latency). Read sequential and random latency reach 169 ns and 305 ns while write latency are merely 90 ns and 62 ns for ntstore and clwb operations. For parallel I/O, the performance of

[2] In this paper, we assume that PM is always large enough to accommodate all records and unsorted records are initially stored in PM while DRAM is not always sufficient relative to PM.

Table 1
Average time and space complexity of traditional sorting algorithms.

	Read time complexity	Write time complexity	Space complexity
Insertion Sort	$O(N^2)$	$O(N^2)$	$O(1)$
Selection Sort	$O(N^2)$	$O(N)$	$O(1)$
Quick Sort	$O(N log N)$	$O(N log N)$	$O(log N)$
Merge Sort	$O(N log N)$	$O(N log N)$	$O(N)$
Heap Sort	$O(N log N)$	$O(N log N)$	$O(1)$

Optane is non-monotonic with increasing thread count. For the non-interleaved (i.e., single-DIMM) cases, performance peaks at between one and four threads and then tails off. Interleaving pushes the peak to twelve threads for store+clwb. The WPQ (write pending queue) buffer queues up to 256 B data issued from a single thread [22]. The DWPD (Drive Writes Per Day) of Optane can reach 60 for 5 years lifetime, which outperforms other persistent memory. The emergence of PM has inspired a lot of researches for building persistent storage systems and applications [1–3,5].

2.2. Review of conventional sorting methods

Sorting is one of the most important components in many storage systems and indexing structures, such as DBMSs [10,11], KV-Stores [7,8], and B+-Trees [24,25]. Traditional sorting algorithms can be divided into two types: internal sort and external sort. Internal sort executes the sorting procedure for all records directly in memory space. By contrast, external sort is used for large data size that does not fit in memory and depends on a two-phase sorting procedure: (1) divide all the records into several chunks and each time load a single chunk into memory from disk/SSD to perform an internal sort (e.g., quick sort) on the chunk, then write out the sorted chunk to disk/SSD; (2) use merge sort in memory to combine multiple sorted chunk records into globally-sorted records. Table 1 provides the average time and space complexity for some representative internal sorting algorithms. Clearly, selection sort has the lowest write complexity while it suffers from high read complexity. Insertion sort has both high read complexity and write complexity. Other sorting algorithms, such as quick sort, merge sort and heap sort, achieve more balanced read and write complexity (i.e., both are $O(N log N)$).

2.3. Sorting in Persistent Memory

Although there are a lot of researches on both PM-based system design and in-memory data sorting optimizations, few open discussions have been made on combining these two points together and redesigning the sorting engine in persistent memory. An intuitive idea is to apply conventional sorting algorithms, such as quick sort [26], selection sort [27], merge sort [28], and external sort [29] for PM-based systems. However, such a naive migration for sorting methods in PM can lead to huge writes, which could reduce the lifespan of PM. In addition, sorting records directly in PM will lead to heavy time overhead with an increasing record size.

A few researchers have proposed PM-friendly sorting methods [19–21] by exploiting the unique features of persistent memory device. Due to the commonly-believed read/write asymmetry feature (i.e., write latency is much higher than read latency), they seek to trade off fewer writes for additional reads or minimize the write complexity assisted with special data structures. Segment sort [20] allows a tunable combination of external sort and selection sort. That is, α $(0 \leq \alpha \leq 1)$ of all records are sorted by external sort and the remained $(1-\alpha)$ portion are sorted by selection sort. These two portions are then merged into the final sorted records. The reason is that although the read complexity of selection sort is $O(N^2)$, its write complexity is merely $O(N)$. Given that PM's read latency is much lower than write latency, segment sort is supposed to achieve better performance than simple external sort or selection sort with a proper α setting. B*-sort [19] develops a binary tree-based structure for sorting records in PM, which has $O(N)$ complexity for writes and $O(N log N)$ complexity for reads. B*-sort also uses extra tunnel lists and register metadata to optimize the worst-case read complexity. Luo et al. [21] improve traditional heap sort by placing nodes near the heap root in DRAM and those near the leaves in PM to reduce writes in PM based on the observation that nodes close to the root are more likely to be accessed.

2.4. Inconsistency issue in Persistent Memory

Inconsistency is a significant problem for power failure or system crash in persistent memory, mainly including partial update and re-ordering write. In case of the data inconsistency problem due to system failure, many mechanisms such as logging (undo/redo), copy-on-write, checksum and persistence primitives are widely applied in PM-based systems [5,30,31] and data structures [24,32,33]. NOVA [5] relies on three write ordering rules to ensure consistency. First, it commits data and log entries to PM before updating the log tail. After that, journal data is promised to be stored into PM before propagating the updates. Finally, new versions of data pages are committed to PM before recycling the stale versions. Besides, Nova also employs *clwb* and *PCOMMIT* to ensure entries committed to PM and *fence* to order subsequent *PCOMMIT* and store operations. To ensure consistency, ZoFS [30] issues atomic instructions with proactive cache line flushes to provide atomic metadata updates and each metadata update is divided into multiple steps. ZoFS follows a carefully-decided order of these steps, so that after crashes, partially processed metadata updates can be roll-backed by the recovery code. ZoFS's consistency relies on an offline recovery phase. For recovery, ZoFS first scans all coffers in the whole file system and then recovers in-coffer metadata for each coffer. Concretely, it starts with the coffer root page to traverse the whole coffer and records in-use PM pages as well as cross-coffer metadata. When a corrupted file or dentry is found during the traversal, ZoFS tries to recognize and recover it. If the recovery is not possible, ZoFS skips the corrupted content. At the end of traversal, ZoFS sends all in-use pages in the coffer to KernFS, which will reclaim the rest of the pages. After all in-coffer metadata are checked, ZoFS continues to validate cross-coffer metadata according to the information recorded during coffer traversals. To address the consistency issue from unanticipated cache line eviction without page cache and cache flush operations, SoupFS [31] adopts dual views, a latest view and a consistent view, which is used in soft updates for file system metadata. All metadata in the consistent view is always persisted and consistent, while metadata in the latest view is always up-to-date and might be volatile. Without caring about the cache line eviction, a syscall handler operates directly in the latest view and tracks the dependencies of modifications. Unanticipated cache line eviction in the latest view can never affect the persisted metadata in the consistent view by design. Background persisters are responsible for asynchronously persisting metadata from the latest view to the consistent view according to the tracked dependencies. By using *clflush/clflushopt+sfence* operations, background persisters enforce the update dependencies without affecting the syscall latency. In addition, to implement dual views efficiently, SoupFS also proposes pointer-based dual views, in which most structures are shared by both views and different views are observed by following different pointers. SoupFS can avoid almost all synchronous cache flushes in the critical path, and the consistent view can be immediately used without performing file system checking or recovery after crashes.

NV-Tree [32] only enforces consistency on leaf nodes without logging or versioning and reconstructs internal nodes during failure recovery. CDDS-Tree [33] uses FLUSH to enforce consistency on all the tree nodes. In order to keep entries sorted, all the entries on the right side of the insertion position need to be shifted when an entry is inserted to a node. The consistency cost in CDDS-Tree is significant

Table 2
Execution time (ms) of typical traditional sorting algorithms.

	10K	100K	1M	10M	100M	1B
Selection sort	123	12195	1232723	> 1 h	> 1 h	> 1 h
Insertion Sort	186	18376	1911156	> 1 h	> 1 h	> 1 h
Quick Sort	3.7	23	199.1	2581	24291	296782
Merge Sort	4.6	28.4	343.2	4271	43871	569731
Heap Sort	5.8	33.5	416.6	7975	78128	1526177

because it performs FLUSH for each entry shift. In addition, the entry-level versioning method is also applied to all tree operations to ensure consistency. Chen and Jin [24] propose the write atomic B+-Trees (wB+-Trees), a new type of main-memory B+-Trees, that aims to reduce overhead to promise consistency as much as possible. Each wB+-Tree node employs a small indirect slot array and a bitmap so that most insertions and deletions do not require the movement of index entries. In this way, wB+- Trees can achieve node consistency either through atomic writes in the nodes or by redo-only logging.

3. Motivation

Although the adapted conventional sorting algorithms and recently-proposed PM-friendly sorting techniques can be applied for persistent memory scenarios, we claim that both types of sorting methods have non-negligible limitations, such as high read/write complexity, severe write overhead due to large value size, and remarkable performance decrease caused by limited DRAM resource. To observe the bottlenecks of existing sorting methods for PM, we conduct a series of experiments on randomly-generated records, each of which only contains a key (fixed 8-byte in size) and a value (varied-size). The detailed configuration of our experimental platform is provided in Section 5.1.

3.1. Comparison of typical traditional sorting algorithms

Table 2 shows the time consumption of typical traditional sorting algorithms to sort records with 8-byte values from 10 thousand to 1 billion. We have two observations. First, selection sort has better performance among the algorithms with $O(N^2)$ time complexity. It has 1/3 less time consumption compared to insertion sort. Second, quick sort achieves the best time efficiency among all the compared sorting algorithms. Although merge sort and heap sort have the same read and write complexity as quick sort, they have lower performance in practical running. We infer that it is because quick sort can better utilize memory cache locality while merge sort always writes its temporary sorted results to new memory space and heap sort has more non-adjacent elements comparison. On one hand, for sorting an array that fits into cache, quick sort requires fewer memory accesses while merge sort needs to allocate an additional array in the memory space. On the other hand, for sorting an array that does not fit into cache, both quick sort and merge sort recurse to sort smaller sections, and quick sort still outperforms merge sort by the former argument. The drawback of quick sort, however, is that it has more writes than selection sort.

3.2. The impact of value size

Table 3 shows the execution time of three traditional sorting algorithms (i.e. selection sort, insertion sort and quick sort) for sorting 100 thousand records with the value size growing. The In-PM mechanism means that the sorting procedure is directly executed in PM, which is likely to be applied in a scenario with scarce DRAM; the In-DRAM mechanism indicates that the records are loaded into and sorted in DRAM (but not stored back to PM), which is often adopted by users when there is no need to save the sorted result; the PM-DRAM mechanism represents that the records are loaded into and sorted in DRAM, and finally stored back to PM. Note that the write-back overhead can be calculated by the PM-DRAM latency minus the In-DRAM latency.

From Table 3, we have three main observations. First, with the value size growing, the time consumption of sorting algorithms can increase remarkably. For instance, insertion sort and quick sort incur 7.1x and 4.7x time overhead in PM, respectively, when the value size increases from 8B to 512B. This is because more reads and writes are performed during the sorting procedure. Second, for both selection sort and insertion sort, the In-PM time consumption is similar to that of In-DRAM and PM-DRAM. This indicates that their time consumption for sorting is too heavy due to $O(N^2)$ time complexity. Third, for quick sort, the PM write overhead plays an important role in the final performance. It can be seen that In-DRAM and PM-DRAM quick sort outperform In-PM quick sort by 2.3x and 1.79x respectively, when the value size is 4 KB. Notice that compared to In-PM quick sort, PM-DRAM quick sort can also reduce writes to PM (i.e., merely N writes for storing sorted records), and hence alleviate the risk of PM wear out.

3.3. The impact of DRAM capacity

In conventional storage architecture, data is first loaded from disk/SSD to DRAM, and the actual sorting procedure is executed in DRAM. However, compared to the durable storage device, DRAM space can be relatively more scarce, and hence it cannot store all the records in some cases. To address this problem, external sort is employed. Records in disk/SSD are divided into multiple chunks, each of which can be fitted in DRAM space and sorted. Each sorted chunk will be written back to disk/SSD and they will be merged as a final sorted file via properly using the limited DRAM resource. In a DRAM-PM hybrid architecture, external sort can still work in the similar way. Fig. 1 shows the performance effect on external sort with different relative DRAM capacity. The number of records is ten million and the In-PM quick sort is considered as a baseline. The PM size taken up by all records can be calculated by the total number of records multiplying the record size (i.e., key size plus value size). Moreover, the DRAM size taken up can be calculated by the ratio shown in Fig. 1 multiplying the PM capacity used.

From Fig. 1, we can observe that the performance of external sort decreases with the relative DRAM size being smaller, but the sharp performance drop is mainly in the shift from full record size to 1/2 record size. For instance, with 8-byte value size, the time consumption climbs by 2.2x, when changing the DRAM size taken up by records from full record space to 1/2 record space. However, only 18% extra time is incurred when shifting the DRAM size from 1/2 record space to 1/4 record space. Second, external sort is better than (In-PM) quick sort in performance with sufficient DRAM space and worse than (In-PM) quick sort if the DRAM size is smaller than the total record size. Notice that compared to In-PM quick sort, external sort has much better wear-leveling capability because it requires only $O(N)$ PM writes.

3.4. Performance study of PM-friendly sorting methods

Segment sort [20], B*-sort [19] and NVMSort [21] are three recently-proposed PM-friendly sorting methods that show good performance on a DRAM-simulated platform. Currently, since the real PM product is available, it should be interesting and useful to study their real performance.

Segment sort is a combination of external sort and selection sort, and its main idea is to trade off fewer writes for additional reads since PM is expected to have much higher write latency than read latency. Viglas et al. [20] believe that there will be an optimal ratio to make segment sort reach the best performance. However, we observe a different result on Optane-based platform. Fig. 2(a) shows the time consumption of segment sort with different ratio to sort 100 thousand records. For simplicity while maintaining the spirit of segment sort, we replace the external sort with In-PM quick sort, which has higher write

Table 3
Execution time (s) with different value size.

	In-PM				In-DRAM				PM-DRAM			
	8B	64B	512B	4KB	8B	64B	512B	4KB	8B	64B	512B	4KB
Selection Sort	12.2	13.5	18.3	30.6	12.2	13.5	18.2	30.2	12.2	13.5	18.2	30.3
Insertion Sort	18.4	27.9	130.2	> 1 h	18.4	27.6	127.9	> 1 h	18.4	27.6	127.9	> 1 h
Quick Sort	0.02	0.04	0.14	0.95	0.02	0.03	0.07	0.41	0.02	0.03	0.09	0.53

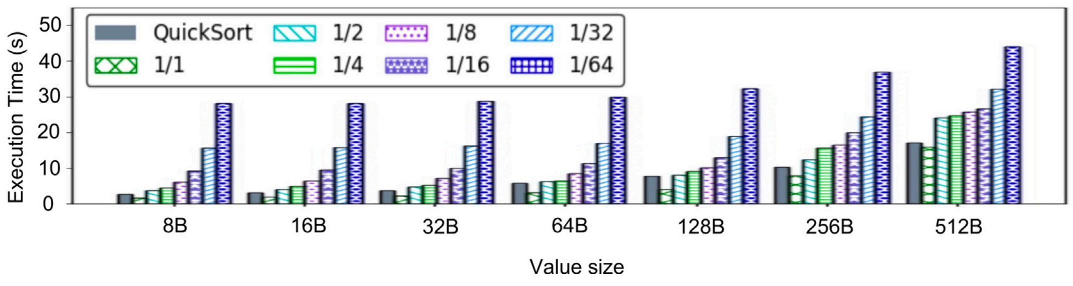


Fig. 1. The impact of relative DRAM size on external sort for sorting ten million records (In-PM quick sort as a baseline).

complexity but lower read complexity than selection sort to illustrate the performance of segment sort's main idea on Optane (IN-PM quick sort has more PM writes while does not have a negative impact than external sort on the overall performance as shown in Fig. 1). The merging phase is kept as the previous design.

In Fig. 2(a), α=0 means that segment sort only uses selection sort; by contrast, α=1 indicates that segment sort only utilizes quick sort. We can learn from Fig. 2(a) that as α decreases, the time overhead grows as well. In other words, the larger the ratio of quick sort is employed, the better the performance of segment sort is achieved. Segment sort can gain no time profits from selection sort by trading off fewer writes for additional reads. We believe that there are two reasons for it. First, the read latency is not better than write latency. A recent study on Optane's performance [22] shows that the random 8-byte read (i.e., load) latency can be 300 ns while the random 8-byte write (i.e., store) latency can be merely 100 ns. Optane's write can be faster than its read in terms of latency. While the read bandwidth of Optane is higher than write, we infer that the bottleneck for sorting is not bandwidth based on several experiments. For example, for sorting ten million records with 8B key and 4 KB value, quick sort takes 224.7s while random write takes merely 50.33s. Thus, for Optane, segment sort just trades off faster writes for slower reads. Second, selection sort is not as efficient as quick sort in utilizing cache locality, and the portion of records using selection sort becomes the bottleneck in the entire sorting procedure. In conclusion, segment sort is worse than the simple quick sort algorithm.

B*-sort [19] adopts a binary search tree structure to reduce the complexity of PM writes to $O(N)$ and limit the average complexity of PM reads to $O(N \log N)$. To avoid the worst case for reads, it also utilizes additional tunnel lists and register metadata. NVMSort trades off part of PM writes for DRAM writes. While the theoretic complexity of B*-sort and NVMSort is much better than quick sort, the performance results on Optane-based platform are much worse. Fig. 2(b) compares the time consumption among B*-sort, NVMSort, (In-PM) quick sort, (In-PM) merge sort and external sort for one million records when DRAM capacity is 1/2 the total record size. We can draw two takeaways from Fig. 2(b). First, for small-size values, B*-sort and NVMSort have much higher time overhead than quick sort. For instance, the time consumption of B*-sort is 6.1x and 5.5x higher than quick sort with the value size of 8B and 64B, respectively. There are two reasons for this: (1) B*-sort is a pointer-based data structure and hence incurs a lot of random reads and writes during sorting. NVMSort is a heap-based structure and the index range of data swap in the record sequence is larger than that of quick sort for recursion. They not only fail to utilize cache locality but also incur severe time overhead [15,23]. The impact of cache invalidation overhead vs write-once property benefit can be evaluated via the Valgrind tool [34]. Taking the 8B value size for B*-sort and quick sort in Fig. 2(b) as an example, the D1 and LLd miss rate are 3.3% and 0.4% for B*-sort and 1.2% and 0.2% for quick sort. Although B*-sort has the write-once property, the cache locality is worse than that of quick sort, which incurs heavier overhead; (2) the additional tunnel lists and register metadata in B*-sort add more PM-allocated overhead in the critical path. Second, for larger-size values, B*-sort can be comparable to quick sort. It is observed that with 4 KB value size, B*-sort is much better than NVMSort and merge sort while obtains merely less than 15% time consumption compared to quick sort. This is because each tree node access can benefit from sequential reads and writes. In conclusion, B*-sort is worse than the simple quick sort algorithm in performance but better for the wear-leveling goal.

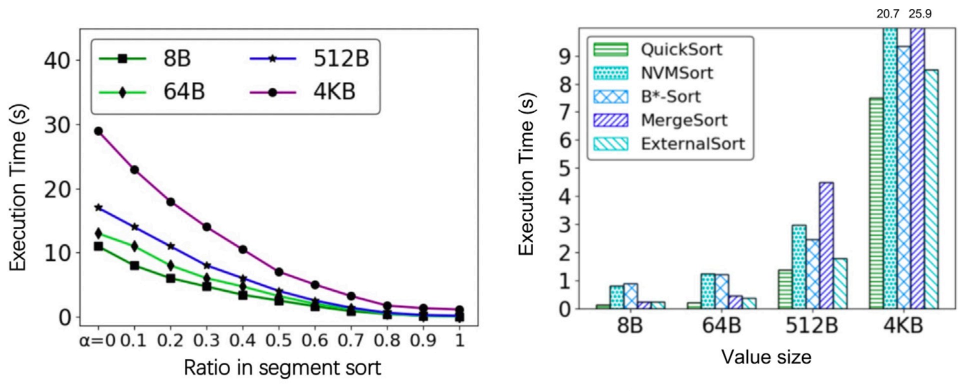
4. PMSort

As Section 3 demonstrates, both traditional sorting algorithms and recently-proposed PM-friendly sorting methods have limitations, which include (1) performance issue caused by large value size, limited DRAM capacity, and random PM read/write overhead; (2) wear-out concern caused by PM writes during sorting. Furthermore, no single sorting method can beat others in all cases. For instance, while quick sort is better in time efficiency than selection sort and B*-sort, it is not better in wear-leveling. While PM-DRAM quick sort can be better than In-PM quick sort, it heavily depends on the space consumption of DRAM and when the available DRAM capacity is limited, it will transform to external sort and the performance can drop sharply.

Based on these key observations and conclusions, we claim that it is necessary to redesign the sorting engine for persistent memory. In this paper, we propose an adaptive sorting engine, which is named PMSort, to address the challenges we mentioned. In this section, we first present the overall structure of PMSort, then introduce the PM-enabled pointer-indirect sort mechanism, and finally provide the details of how PMSort works.

4.1. Overview

PMSort is an adaptive sorting engine that targets at providing the appropriate sorting technique for each workload according to the workload features and other significant related conditions. Fig. 3 shows the architecture of PMSort. PMSort is composed of four core components: *Sensor (SS)*, *Decision Engine (DE)*, *Sorting Algorithm Library (SAL)* and *Execution Engine (EE)*. Among these components, *SS* is designed for extracting the useful information from unsorted records, requirements of users, workloads and the hardware in system (shown in ①), and



(a) Segment sort.

(b) NVMSort and B*-sort.

Fig. 2. Performance study for PM-friendly sorting methods.

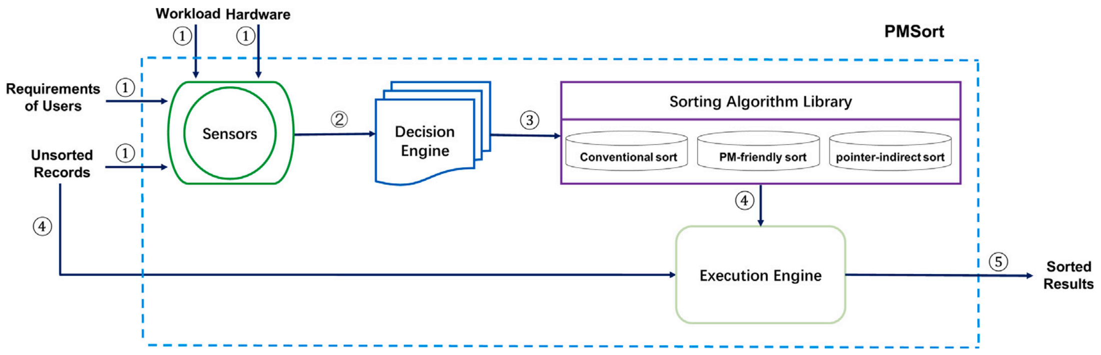


Fig. 3. Architecture of PMSort.

conveying the information to *DE* (shown in ②). The information includes four aspects: (1) information of unsorted records, such as the size of total records, the number of total records, and the value size of records; (2) requirement information of users, such as the wear-leveling requirement of application and the persistence requirement of the sorted results; (3) workload information, such as the limited write size of PM and limited sorting time; (4) hardware information, such as the available DRAM capacity and the PM write endurance. Based on the above collected information, *DE* is responsible for selecting the appropriate sorting technique from *SAL* (shown in ③). *SAL* is a suite of sorting techniques, including adapted conventional sorting algorithms, existing PM-friendly sorting techniques and pointer-indirect-optimized sorting methods (see Section 4.2 for more details). After that, the selected sorting method will be transmitted to *EE*, and *EE* performs it on the unsorted records (shown in ④). Finally, the sorted results are generated as output (shown in ⑤). They may be either persisted in PM or simply copied to the user buffer. Note that the sensors can collect more inputs if user demands and more future sorting methods can be integrated in the sorting algorithm library.

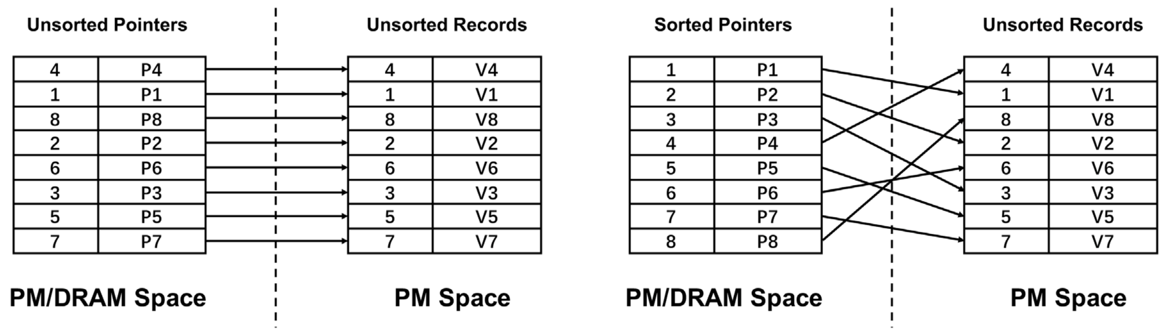
4.2. PM-enabled pointer-indirect sort

For traditional DRAM-Disk storage architecture, records should be first loaded into DRAM for sorting, with internal or external sorting algorithms. The loading procedure is performed in a block-based style. That is, blocks containing the full record information (i.e., keys and values stored compactly) are loaded into DRAM. The main bottleneck is the I/O overhead. Now, with PM, the records can be stored in PM directly for storage systems and applications, and the sorting procedure can be executed in PM as well. But as we point out in Section 3, the sorting performance drops remarkably with the growth of value size, which incurs more PM reads and writes. Some main memory database systems enable to use pointers for data indexing [35], which reduces the movement for the actual large-size value during scan or some other operations. Inspired by this, we devise the pointer-indirect sort mechanism to speed up the sorting performance for PM-resided records which have large-size values. Note that one more hop to access data and pointer mapping overhead caused by the pointer-indirect mechanism is quite small compared to normal sorting process, as shown in Sections 5.3 and 5.4. In addition, the pointer-indirect mechanism is very suitable for software such as TikTok and YouTube [36]. These software has range query functions where sort is the major operation. For example, we can search the most popular videos in order in a week. With the rapid growth of data, PM is gradually applied in these systems.

The key step of the pointer-indirect sort mechanism is building the mapping <key, pointer> records, based on the original <key, value > records, as Fig. 4(a) illustrates. Concretely, a new region (in DRAM for IN-DRAM and PM-DRAM sorting methods, and in PM for IN-PM sorting methods) is created, and each <key, value > record is transformed to a much smaller <key, pointer> record, where *the pointer is an indirection (i.e., address) to the original <key, value > record*. Suppose that the key is fixed-size with 8B, then we can limit the <key, pointer> record to be merely 16B. Due to the space-efficiency of <key, pointer> records, with a given DRAM capacity, a much larger number of records may be loaded into DRAM for sorting when it is compared to traditional full-record loading mechanism. Therefore, it is possible to transform the heavyweight external sort to the much simpler In-DRAM quick sort.

As provided in Fig. 4(b), instead of directly conducting a sorting algorithm on large-size records, the pointer-indirect sort mechanism enables to sort the much smaller-size <key, pointer> records, and hence reduces a lot of PM reads and writes. It is also easy to read out sorted records via the sorted pointers. We have studied the result reading overhead in Section 5, which is very lightweight compared to the actual sorting overhead.

In conclusion, the pointer-indirect sort mechanism is beneficial to both sorting performance and PM wear-leveling for large value size.



(a) Build mapping <key, pointer> records.   (b) Sort <key, pointer> records.

Fig. 4. An example of pointer-indirect sort mechanism.

Notice that the pointer-indirect mechanism is not limited to the use of a single sorting method. It can be combined with all existing sorting algorithms and techniques, such as quick sort, external sort and B*-sort. As shown in Fig. 3, pointer-indirect sort is a core mechanism of the *SAL* component. Moreover, notice that the pointer indirection in this paper is a user-transparent procedure for speeding up sorting and no matter PMSort adopting pointer-indirect or non-pointer-indirect methods, the result will be finally output as <key, value> (not <key, pointer>) to the user buffer in order, which does not change the sorting library syntax.

4.3. PMSort design and sorting selection principles

To achieve the best sorting performance with PMSort (i.e., both I/O performance and weal-leveling according to different users' demands and hardware settings), we should (1) clearly distinguish different workloads and situations as different conditions, and (2) select the most-suitable sorting method for the corresponding condition.

We study a variety of sorting conditions and provide the most appropriate sorting method for each condition in Table 4. We consider the value size to be small if it is smaller or equal to 8 bytes, otherwise it is marked as large. The DRAM capacity is considered to be sufficient if DRAM can store all the <key, value > records or the corresponding <key, pointer> records using the pointer-indirect mechanism. The wear-leveling requirement indicates if the writes should be shorten to $O(N)$ times and the persistence requirement tells if the sorted results should be stored as a persistent object.

As we point in Section 4.2, adopting pointer-indirect mechanism can reduce great overhead for large size value while bring extra little pointer mapping overhead and one more hop to read results for small size value (evaluations are shown in Section 5.3 and Section 5.4). Thus, pointer-indirect mechanism is more suitable for records with large value size and there is no need to adopt it when the value size is small. Moreover, sufficient DRAM enables sorting to be performed in DRAM, which leads to both better performance and fewer PM writes. Therefore, DRAM utilization should be considered for sufficient DRAM conditions while sorting ought to be performed in PM or external sort should be used when DRAM resource is scarce. Wear-leveling is a common issue for PM, which may reduce PM lifespan. If users require for better wear-leveling performance, sorting can be performed in DRAM directly under sufficient DRAM condition, and B*-sort or external sort are more appropriate under insufficient DRAM condition compared with quick sort since B*-sort has write-once characteristic and external sort uses DRAM to reduce PM writes. If wear-leveling is not very concerned, quick sort is more suitable for less sorting time consumption. In addition, whether or not to persist sorting results in PM also affects the selection decision. For sorting process in PM, sorted results are stored in PM as well. For sorting process in DRAM, if there is no need to persist sorted results, results will be read to the user buffer directly, which can save both write-back time cost and write traffic to PM.

The corresponding best-suited sorting method is carefully selected based on our analysis on existing sorting algorithms under different settings and evaluated by experiments. Fig. 5(a) shows the time consumption for eight candidate sorting methods to sort ten million records when DRAM capacity is sufficient. We can observe that the performance of QuickSort-PMPtr (i.e., pointer-indirect In-PM quick sort) is much higher than QuickSort-PM (i.e., pure In-PM quick sort) when the value size varies from 64B to 4 KB (e.g., 2.2x, 6.9x and 36.8x for 64B, 512B and 4 KB values, respectively). However, when the value size is small (i.e., 8B), there is no need to employ the pointer-indirect mechanism because their performance is nearly equal.

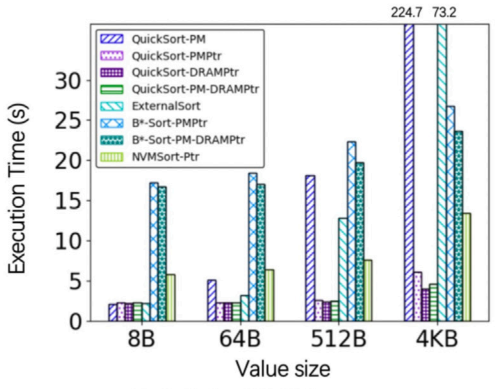
From Fig. 5(a), we can also observe that using DRAM can bring benefits to the sorting performance. For instance, QuickSort-PM-DRAMPtr (i.e., pointer-indirect PM-DRAM quick sort, whose key-pointer map is in DRAM) outperforms QuickSort-PMPtr (i.e., pointer-indirect In-PM quick sort, whose key-pointer map is in PM) by up to 1.4x when the value size is 4 KB. In addition, sorting in DRAM will reduce writes in PM a lot. Therefore, when wear-leveling is required, PMSort should utilize DRAM for sorting. Concretely, when persistence for sorted results is required, it should utilize PM-DRAM quick sort; otherwise, it should use In-DRAM quick sort. By contrast, if wear-leveling is not a critical concern, PMSort can simply choose In-PM quick sort and In-DRAM quick sort according to different persistence requirements.

When DRAM capacity is not sufficient, and if there is no restriction on wear-leveling, then it could be very straightforward to select the In-PM quick sort due to its efficiency. However, if wear-leveling is set as a target, then PMSort should shift the sorting method. Notice that both external sort and B*-sort have the PM write complexity of $O(N)$, which should be the candidate sorting methods for this case. But which one is better? To answer this question, we conduct another experiment to compare the performance of B*-sort with external sort for different relative DRAM capacity. Fig. 5(b) shows the results for sorting ten million records and QuickSort-PMPtr (i.e., pointer-indirect In-PM quick sort) is used as performance baseline. It can be observed that when the relative DRAM space is larger than or equal to 1/16, ExternalSort-Ptr (i.e., pointer-indirect external sort) is better than B*-Sort-PMPtr (i.e., pointer-indirect In-PM B*-sort). When the relative DRAM capacity gets even smaller, B*-Sort-PMPtr starts to outperform ExternalSort-Ptr. Although NVMSort-Ptr (i.e., pointer-indirect NVMSort) performs better than ExternalSort-Ptr and B*-Sort-PMPtr when the relative DRAM space is smaller than or equal to 1/16 in Fig. 5(b), its PM write size is comparable to that of QuickSort-PMPtr in that case as Fig. 6 shows. Thus, NVMSort-Ptr is not a suitable choice when wear-leveling is required. In our current implementation, we set the (relative DRAM capacity) switch boundary between external sort and B*-sort as 1/16. Compared with the baseline, PMSort has a limited boundary of extra time overhead (i.e., 4.5x) to guarantee wear-leveling, which should be acceptable in practical use.

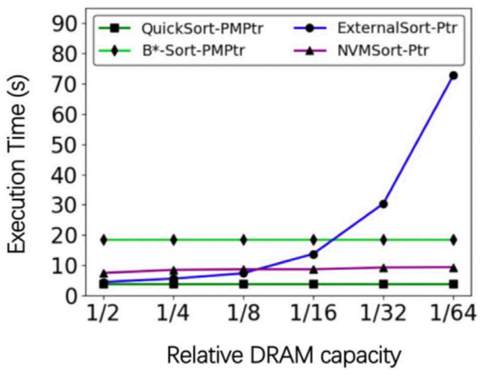
Based on the experiments and analysis above, we have demonstrated that each employed sorting technique in PMSort is the best-suited one for the corresponding condition. We outline evaluation

Table 4
Complicated sorting conditions and corresponding sorting methods.

Value size	Sufficient DRAM	Wear-leveling	Persistence	Best-suited sorting method
Small (large)	Yes	Yes	Yes	(pointer-indirect) PM-DRAM quick sort
	Yes	Yes	No	(pointer-indirect) In-DRAM quick sort
	Yes	No	Yes	(pointer-indirect) In-PM quick sort
	Yes	No	No	(pointer-indirect) In-DRAM quick sort
	No	Yes	Yes	(pointer-indirect) external sort
	No	Yes	No	(pointer-indirect) In-PM B*-sort
	No	No	Yes	(pointer-indirect) In-PM quick sort
	No	No	No	



(a) Sufficient DRAM space.



(b) Insufficient DRAM space.

Fig. 5. Comparison of the execution time for different sorting methods.

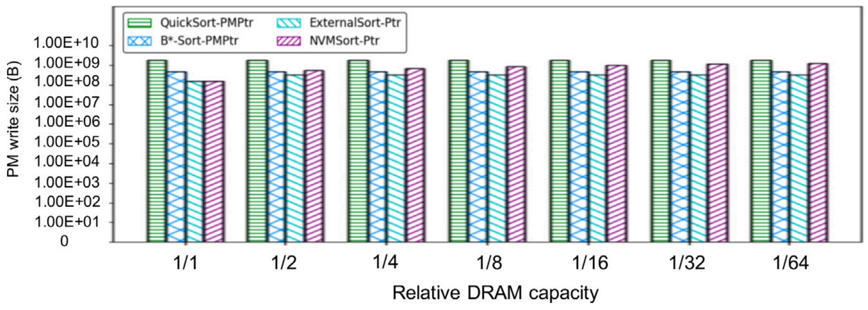


Fig. 6. Write size in PM for different pointer-indirect sorting algorithms.

results and summarize principles on selecting the optimal algorithms here. The pointer-indirect mechanism is quite effective for large size value and the suitable utilization of DRAM can both improve sorting performance and reduce write traffic to PM. Different sorting algorithms have different characteristics such as time complexity, cache locality utilization, write traffic to PM, memory capacity occupation, etc. For different settings and user requirement, all algorithms should be studied carefully. For storage with wear-leveling concerns such as PM, it is more suitable to choose algorithms producing fewer writes to storage or better utilize other storage such as DRAM which has longer writing lifespan. Algorithms with lower time complexity do not mean that they consume less execution time. Some factors such as cache miss rate and metadata overhead of an algorithm should be evaluated and experiments should be performed on a real PM platform to study the real performance of an algorithm. For designing a better PM-based sorting algorithm, the read and write characteristics of PM also should be taken into consideration, which is quite different from those of DRAM.

4.4. Recovery methods in PMSort for power/system failure

Sorting a large number of records always consumes huge amount of time, which accounts for non-negligible overhead in database systems. Since power failure or system crash may interrupt the sorting process, some inconsistent states may occur and still exist in PM after recovery. Fig. 7 illustrates three possible inconsistent states for data swap in PM. The initial state shows that two <key, pointer> records (i.e., <key1, pointer1> and <key2, pointer2>) need to be swapped with the assistance of a temporal <key, pointer> record space (i.e., <Tkey, Tpointer>). <key2, pointer2> record is moved to the temporal area first. After that, <key1, pointer1> record is copied to the original <key2, pointer2> record area and finally the <key2, pointer2> record is sent back to its target space. State 1 and state 2 occur when the power offs or system crashes in step 2, which means the <key1, pointer1> record only has been copied partially and the <key2, pointer2> record has not been moved respectively. The state 3 mainly attributes to the system failure in step 3. The 8-byte key2 has been duplicated atomically while the pointer2 still exists in the temporal space. Similar with inconsistency problems in pointer-indirect sorting methods, three inconsistent cases as Fig. 7 shows may also occur in non-pointer sorting algorithms performed directly on <key, value > records. From the same start state, the above three cases lead to three different inconsistent states. Similarly, it is easy to show that the same inconsistent state can be derived from multiple start states with different failure cases. Consequently, using the information of an inconsistent state alone is impossible to recover from the failure because there is no way to tell which start state is the correct one.

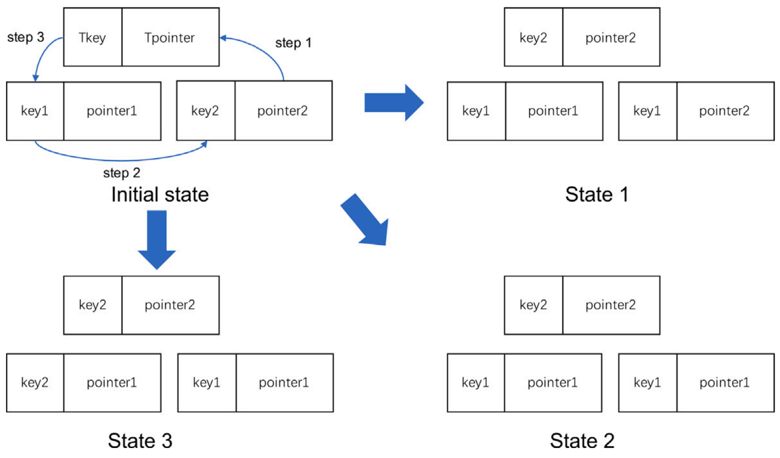


Fig. 7. Probable inconsistent states for sorting in PM.

In case of the data inconsistency problem during failure recovery, many file systems [5,30] and KV-Stores [24,32] are designed for persistent memory. For sorting in PM, consistency is non-trivial as well. However, few open discussions have been made on how to eliminate inconsistency for sorting. To avoid these inconsistent states mentioned above, more specific mechanisms are designed for sorting algorithms in SAL in our PMSort engine.

B*-sort is a non-swap sorting method. Keys are inserted into a tree structure in sequence. There is no inconsistency problems since the B-tree can be rebuilt based on the original <key, value > records in PM. However, the reordering overhead sometimes becomes intolerable for database users. In order to reduce the reordering overhead, an intuitive thinking is simply to enable the sorting process recover from the breakpoint and introduce extra overhead as low as possible. Since all keys are inserted in the modified B-tree in order, we can simply use a variable to record the current inserted key, which is considered as the beginning for failure recovery.

Pointer-indirect, In-DRAM and PM-DRAM sorting methods can be recovered from the original <key, value > record sequence in PM. Pointer-indirect sorting process is performed on the pointers, not real <key, value > records. As a result, PMSort can simply create new pointers by mapping original <key, value > records and reorder these new pointers. In-DRAM and PM-DRAM sorting algorithms first load keys into DRAM and then perform sort in DRAM. After failure occurs, since DRAM is volatile, inconsistent cases will disappear and PMSort can simply reload records to DRAM and carry out reordering in DRAM.

In-PM quick sort has to swap keys in persistent memory during sorting process and inconsistent states in Fig. 7 may occur. Sorting needs to be performed on a replication of the original record sequence and during the failure recovery, PMSort makes another copy of the record sequence and reorders these copied records once more.

External sort first divides records into chunks, then sorts each chunk and finally merges all sorted chunks when DRAM is insufficient. Inconsistency may occur when PMSort only copy part of a record from DRAM to PM. Thus, consistency still needs to be ensured and the recovery overhead ought to be reduced as much as possible for both external sort and pointer-indirect external sort. Concretely, if failure interrupts the dividing procedure, PMSort can simply record the chunk number at the breakpoint and perform quick sort for each unsorted chunk during recovery process. Similarly, PMSort may label the to-be-merged record in each chunk before power/system failure and continue the merging process after power/system restoration. If DRAM is sufficient, external sort can be transformed to In-DRAM quick sort and PMSort can simply adopt the recovery mechanism for In-DRAM sorting methods.

In addition, if the key size is larger than 8B, the sorting process may also lead to an inconsistent state, namely partial write inconsistency. That means the key may be written incompletely when the power fails. In this case, our recovery mechanism mentioned above can still address this problem to ensure data consistency, including B*-sort, external sort, pointer-indirect, In-DRAM, PM-DRAM and In-PM sorting methods.

To sum up, based on our designs, PMSort can ensure the consistency of all sorting algorithms in SAL and greatly reduce the reordering overhead in the event of power or system failure. Experiments for the effectiveness of our recovery designs are shown in Section 5.5.

4.5. *Reshuffle and no-reshuffle tradeoff*

Since we have applied the pointer-indirect mechanism in PMSort, on one hand, we can simply read values through sorted <key, pointer> records after sorting process. On the other hand, reshuffling the original <key, value> record sequence in PM and then accessing the sorted values is another appropriate choice. It seems that sequential access will need one more PM read to get the next value if PMSort adopts the former method. Worse still, the sequential access to the key–value becomes random access into the value map. However, the reshuffling process and corresponding redundant PM write exert extra overhead to the system. In this section, we conduct several experiments to show these tradeoffs.

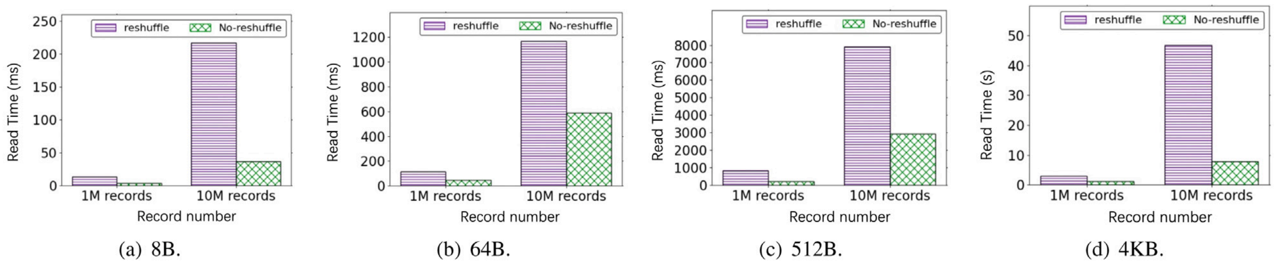
Fig. 8 illustrates the time overhead for reshuffle and no-reshuffle methods sorting one million and ten million records. Reshuffle takes 2x ∼ 5x more time than no-reshuffle method for sorting one million and ten million records. Furthermore, the reshuffle process will allocate a new space in PM to copy all <key, value> records and make them ordered according to the sorted pointers. The extra write overhead caused to PM is the total size of all <key, value> records.

The sorted result may be used as an intermediate result in database systems, not the final result according to the database demand, and the sorted result will not be used again. Moreover, the user may not require to persist the sorted results and PMSort will not write them back to PM for a better wear-leveling and space efficiency goal. In these cases, the sorted result will not be read multiple times and PMSort prefer not to reshuffle the original <key, value> records. In addition, if the extra write overhead caused by the reshuffle process cannot be accepted by users for a very large size of dataset, PMSort will not perform reshuffle either. On the contrary, the original <key, value> record sequence in PM ought to be reshuffled for future read.

5. Experimental evaluation

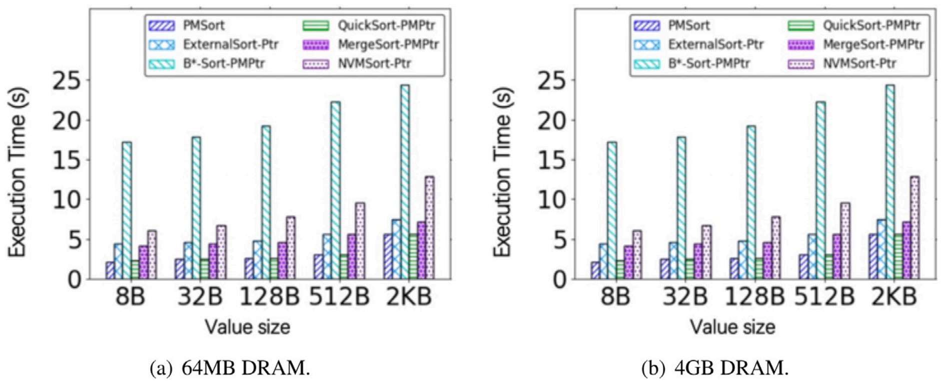
5.1. *Experimental setup*

We implement PMSort using C++ on a Linux server (CentOS 7.8) with 2.60 GHz Intel(R) Xeon(R) Gold 6240 CPU. This CPU has 36 physical cores, with a 24 MB L3 cache. We use 600 GB of overall Optane



(a) 8B. (b) 64B. (c) 512B. (d) 4KB.

Fig. 8. Reshuffle and no-reshuffle tradeoff.



(a) 64MB DRAM. (b) 4GB DRAM.

Fig. 9. The execution time for different sorting algorithms.

DIMM space in the maximum to ensure all records accommodated in this paper. The total DRAM capacity in our machine is 192GB and DRAM size used in Sections 3 and 4 varies according to the record number, value size and relative ratio to PM, as detailed in each figure. In Section 5, DRAM size is fixed as 4GB (large enough to store all records or pointers), 64MB (nearly 1/3 of the total pointer size) and 4MB (1/40 of the total pointer size). Throughout our experiments, the key size is fixed as 8B by default (i.e., keys are randomly-generated integers), and the value size is allowed to vary from 8B to 4 KB. To guarantee data persistence and consistency in PM, similar to many prior works [2,5], we properly utilize *clwb+sfence* instructions to force flushing out the records from caches. We use the standard benchmarks [37] to evaluate the sorting performance of PMSort. Since the source code of B*-sort and NVMSort is not available in public, we implement them faithfully according to their papers. To demonstrate the benefit of PMSort, we compare PMSort with six traditional sorting algorithms (i.e., selection sort, insertion sort, external sort, quick sort, merge sort, heap sort) and three PM-friendly sorting techniques (i.e., segment sort, B*-sort, NVMSort).

5.2. Sorting performance for different workloads

Fig. 9 compares the execution time for sorting ten million records between PMSort and other sorting methods. The DRAM capacity in Fig. 9(a) and Fig. 9(b) is 64MB (i.e., insufficient DRAM space) and 4GB (sufficient DRAM space), respectively. For 64MB DRAM capacity, PMSort will adaptively select quick sort for 8-byte value size (about 0.1s less than pointer-indirect methods) and pointer-indirect quick sort in PM for larger value size when wear-leveling is not a restricted factor. For 4GB DRAM capacity, since 4GB DRAM can accommodate all pointer mapping, sorting process can be performed in DRAM rather than PM. PMSort will adaptively select PM-DRAM quick sort and pointer-indirect PM-DRAM quick sort (or In-DRAM quick sort and pointer-indirect In-DRAM quick sort if there is no need to persist the sorted results) for 8-byte and larger value size respectively.

Fig. 10 shows the maximum number of records that can be sorted in one minute by different sorting methods. We set the DRAM capacity as

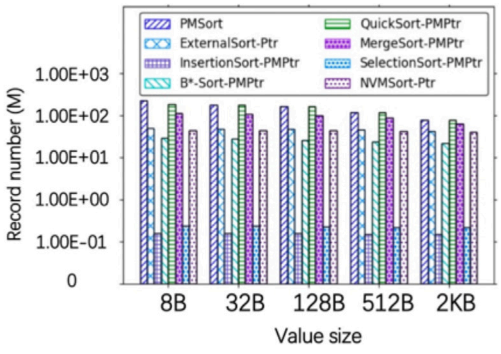


Fig. 10. Sorted record number in 1 min.

64MB, which is insufficient to contain all records. In this case, PMSort prefers In-PM quick sort for a small value size and pointer-indirect In-PM quick sort for a large value size. Insertion sort and selection sort complete the fewest records due to their $O(N^2)$ time complexity.

PMSort also aims to achieve good wear-leveling performance. For sufficient DRAM capacity, sorting process can be performed in DRAM, which contributes to good wear-leveling performance. For insufficient DRAM capacity, external sort and B*-sort show better wear-leveling performance than quick sort in PM since external sort utilizes DRAM and B*-sort has write-once characteristic. Fig. 11 shows the total PM write size (in bytes) of sorting ten million records for the compared methods when DRAM is 64MB and 4GB. For 64MB DRAM capacity, PMSort will adaptively select pointer-indirect external sort. For 4GB DRAM capacity, PMSort will adaptively select pointer-indirect PM-DRAM quick sort (or In-DRAM quick sort if there is no need to persist the sorted results). The extra overhead introduced by PMSort is the information collecting overhead and decision making overhead. Both PM write size and latency of these two types of overhead are less than 1% compared with those of the sorting overhead, and the measurement for these two types of overhead has been included in Figs. 9, 10 and 11.

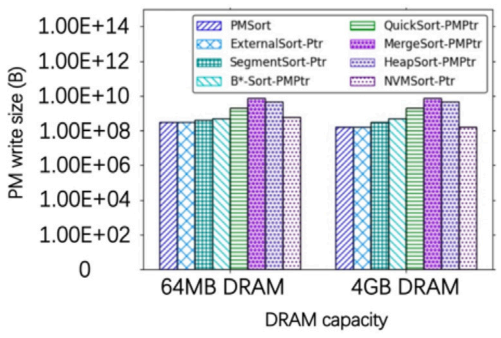


Fig. 11. Write size in PM.

Given ten million records, Fig. 12 provides the upper bound and lower bound of PMSort for both time consumption and PM writes with different DRAM configurations when sorted results need to be persisted. With the decrease of DRAM capacity, the upper bound of PMSort's time consumption gradually increases due to the use of (pointer-indirect) external sort and B*-sort. While the gap between the worst-case time consumption and the best-case time consumption varies remarkably under different conditions, the gap between the worst-case PM writes and the best-case PM writes is stable, which can be represented by $O(N\log N)/O(N)$.

There will be extra metadata overhead if PMSort selects B*-sort (i.e., tunnel lists and register metadata, which is one of the reasons that B*-sort consumes more time than quick sort as shown in Fig. 9). The longest length of a tunnel list is $\lceil\sqrt{n}\rceil$ (n is the total number of records to be sorted) because the maximum length of a sub-BST (i.e., sub binary search tree) structure is $\lfloor\sqrt{n}\rfloor+1$. Since the longest length of a tunnel list is $\lceil\sqrt{n}\rceil$, the amount of write traffic produced by tunnel list creation can be bounded by $O(\sqrt{n})$. The read performance of the tunnel list in B*-sort can be bounded by $O(n\sqrt{n})$ since each record reads at most $\lceil\sqrt{n}\rceil$ entries for insertion. The registers only record four variables (length, ceiling, floor and root, each is 8B) for helping create a tunnel entry.

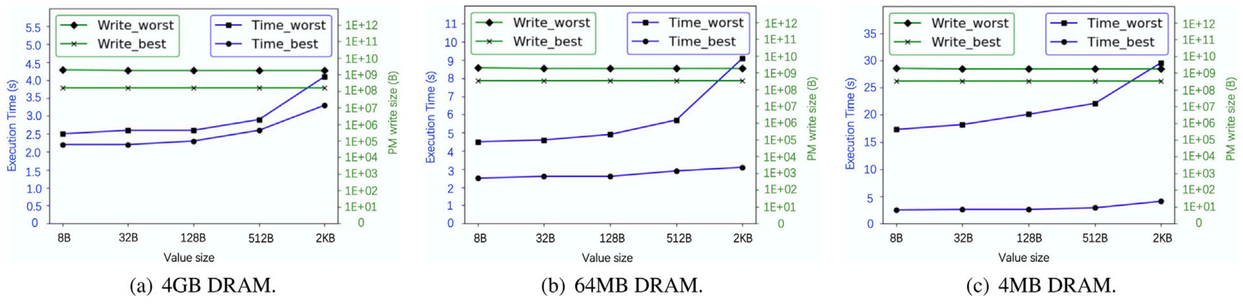
5.3. Reading sorted records impact on total time overhead

In addition to the sorting time cost, the overhead of reading sorted records (i.e., load all the sorted records to the user buffer) should be also studied since it may be in the critical path of an application request (e.g., SELECT command in DBMSs). Fig. 13 compares the time overhead between PMSort and the non-pointer-indirect sorting algorithm.

It can be observed that when the value size is larger than 8 bytes, it is slower to read out the records that are sorted by PMSort. There are two reasons. First, the pointer-indirect mechanism employed by PMSort requires an additional PM load operation for each record read. Second, the reads into the actual records cannot exploit the cache locality since only the pointer records are sorted. It can also be observed that with the value size getting larger, the performance gap becomes smaller. Concretely, when the value size is 512B, the pointer-indirect sort mechanism generates 2.48x time overhead compared with normal quick sort, for reading ten million sorted records. When the value size increases to 4 KB, the relative time overhead caused by pointer-indirect sort is merely 1.46x. Although PMSort requires more time to read sorted results from PM to the user buffer for large-size records, the reading overhead is much smaller than that of sorting (i.e., only 10.7% of quick sort for ten million records with 4 KB values), and the sorting performance is improved by 12.3x. Since the reshuffle overhead for PMSort is no more than 10x that of no-reshuffle (see more details in Section 4.5), the sorting performance can still be improved by orders of magnitude. Therefore, the overall request overhead can be remarkably reduced by PMSort.

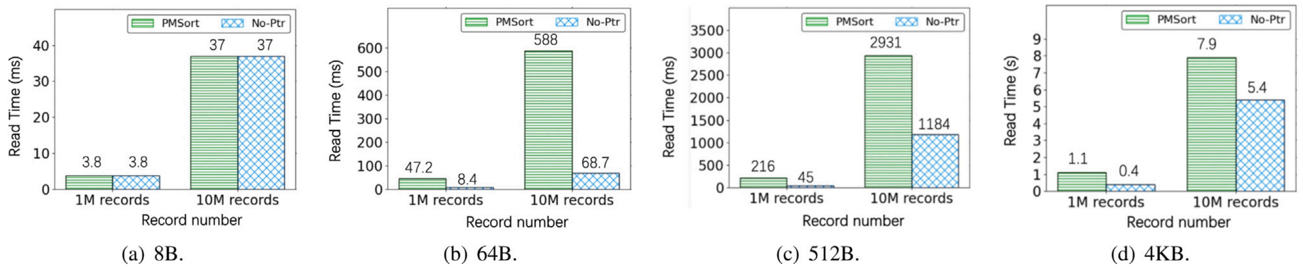
5.4. Building <key, pointer> mapping impact on total time overhead

Since many pointer-indirect sorting algorithms proposed in this paper transform <key, value> into <key, pointer> in the hybrid architecture of PM and DRAM, the time overhead of converting <key, value> into <key, pointer> (i.e., the mapping process) as an extra cost should be studied. Fig. 14 compares the mapping overhead for 10 million records with PMSort_SP (i.e., sorting process in PMSort) overhead and QuickSort-PM_SP (i.e., sorting process in In-PM quick sort) overhead. The DRAM capacity in Fig. 14(a) and Fig. 14(b) are 64MB (i.e., insufficient DRAM space) and 4GB (sufficient DRAM space), respectively. We can get three conclusions. First, with the value size



(a) 4GB DRAM. (b) 64MB DRAM. (c) 4MB DRAM.

Fig. 12. The best and worst performance to persist sorted result.



(a) 8B. (b) 64B. (c) 512B. (d) 4KB.

Fig. 13. Time overhead of reading sorted records for different value size.

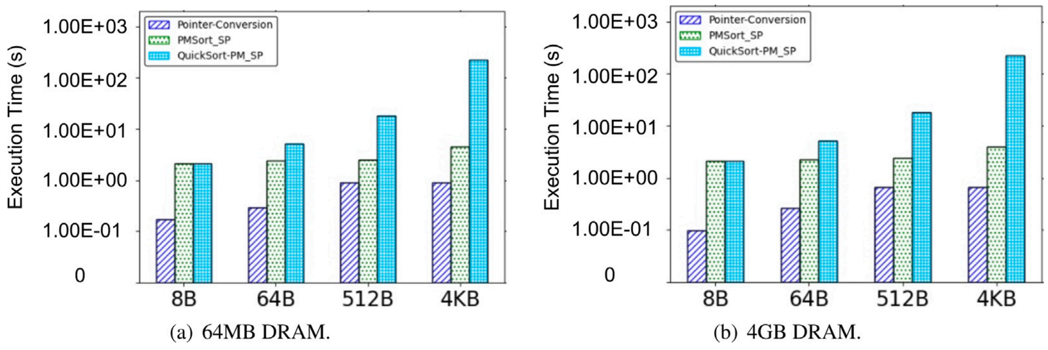


Fig. 14. Mapping overhead for different value size.

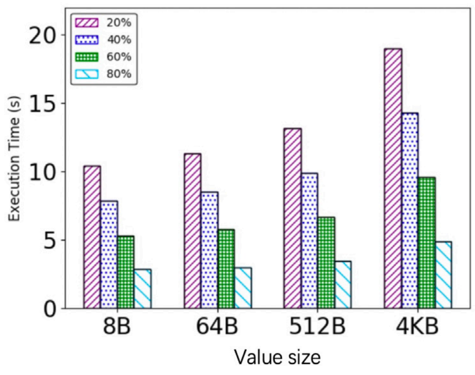


Fig. 15. Recovery for B*-sort.

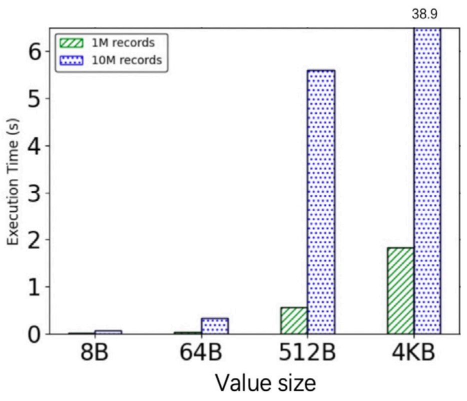


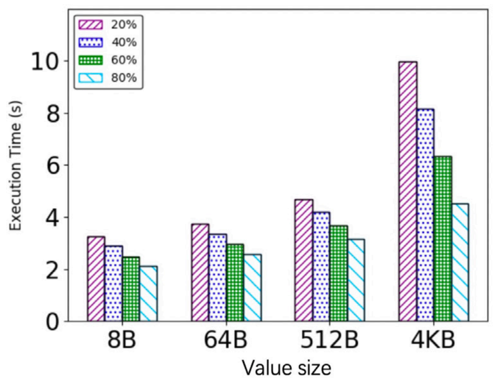
Fig. 16. Overhead for copying records.

growing, the mapping overhead increases when the value size is small while saturates when the value size is large. The mapping overhead includes two part. The first part is to produce pointers for the records. The second part is the overhead to copy the keys since we need to use keys for sorting. The former overhead caused by addressing the memory address is stable with the value size growing while the latter part increases when the value size is smaller than 256B. This is because the internal block size in Optane is 256B and if the 256B buffer contains multiple keys, the load overhead to the buffer can be amortized by these keys. Second, the time cost for pointer conversion with 4GB DRAM (i.e., sufficient DRAM) is lower than that with 64 MB DRAM (i.e., insufficient DRAM), which mainly attributes to the performance gap between PM and DRAM. Third, the mapping time to generate <key, pointer> is far less than the execution time of non-pointer-indirect methods, taking QuickSort-PM (i.e., In-PM quick sort) as an example. Thus, pointer-indirect sorting methods in PMSort can still get benefits from using pointers.

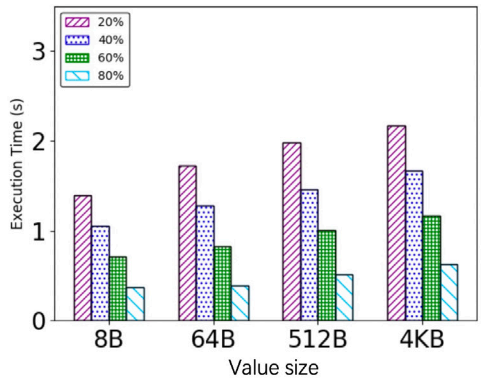
5.5. Effectiveness of failure recovery methods in PMSort

In this section, we will show the effectiveness of our recovery methods designed for persistent memory in PMSort while maintaining consistency during sorting procedure. Fig. 15 illustrates the time consumption for B*-sort (non-pointer for 8-byte value size and pointer-indirect for other value size) in PMSort algorithm library to sort 10 million records (value size varying from 8B to 4 KB) when B*-sort completes different ratio (i.e., 20%, 40%, 60%, 80%, respectively) for sorting records before power/system failure. Since B*-sort is a tree structure sorting method, recovery from the breakpoint consumes approximately the sorting time cost by the left portion of records, merely with extra overhead for identifying the breakpoint. For non-pointer IN-PM sorting algorithms, to ensure consistency in sorting process, making a replication of original record sequence is indispensable, as Section 4.4 elaborates. Fig. 16 gives the execution time for copying 1 million and 10 million records, with value size varying from 8B to 4 KB. Time cost is acceptable when the value size is small while it increases sharply with the value size growing, as Fig. 16 shows. Thus, for larger value size, PMSort takes pointer-indirect IN-PM sorting methods to avoid the tremendous overhead caused by sorting and producing a copy of original records. For 8-byte value size, replicating overhead is acceptable, which is less than 0.2 s.

Extensive experiments are also conducted to show the benefits of our recovery design for external sort. We give the time cost for breakpoints occurring both in dividing and merging process, with different ratio of fulfillment before power/system failure, as Fig. 17 (64MB DRAM capacity) and Fig. 18 (4MB DRAM capacity) illustrate. PMSort selects non-pointer method for 8-byte value size and pointer-indirect for larger value size adaptively. Since Fig. 18 restricts DRAM capacity more severely, records will be divided into more chunks and the record number in each chunk is smaller than that in Fig. 17. Due to more records in a chunk, dividing and sorting each chunk in dividing process take up more time for a larger DRAM capacity. By contrast, merging becomes the critical overhead for a more scarce DRAM capacity since more chunks need to be merged in external sort. Thus, dividing process in Fig. 17(a) has a sharper drop than merging phase in Fig. 17(b) with a growing completed ratio while merging procedure in Fig. 18(b) declines dramatically compared with the dividing portion in external sort as Fig. 18(a) shows. Although the time cost for reordering the remaining records after failure recovery can be reduced in various degree, Figs. 17 and 18 prove that our mechanism designed for crash recovery is efficient employed by (pointer-indirect) external sort.

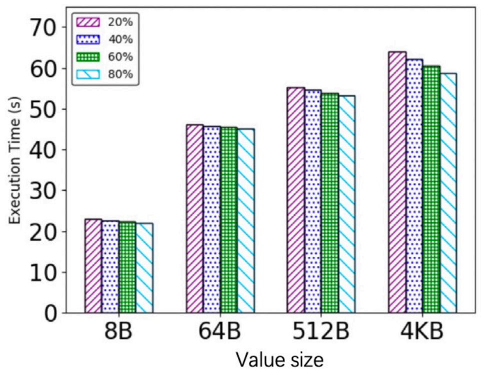


(a) Failure occurs during dividing process.

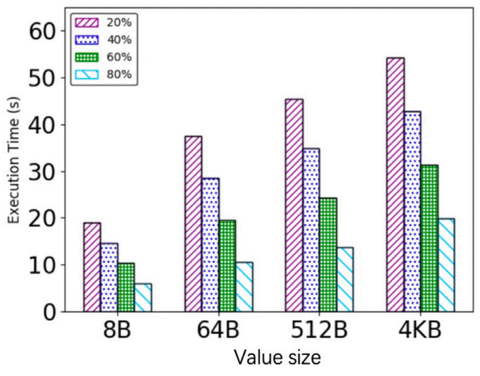


(b) Failure occurs during merging process.

Fig. 17. Failure occurs with 64MB DRAM for external sort.



(a) Failure occurs during dividing process.



(b) Failure occurs during merging process.

Fig. 18. Failure occurs with 4MB DRAM for external sort.

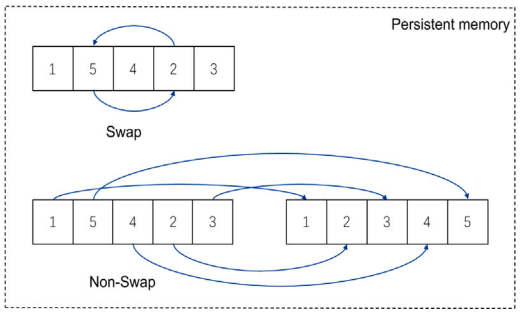


Fig. 19. Swap and non-swap selection sort.

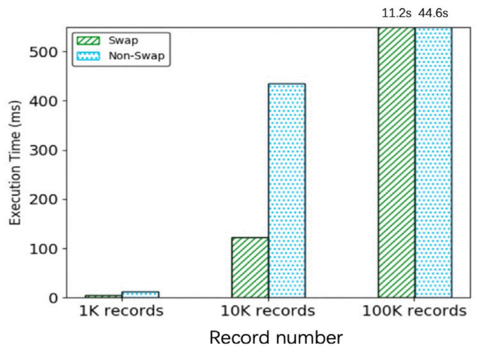


Fig. 20. Performance gap between swap and non-swap.

6. Discussion

6.1. Comparison of swap and non-swap sorting methods

Swap and non-swap are two common categories in sorting algorithms and their impacts in PM are interesting to be discussed. As Fig. 19 shows, in-place update is a common method for data swap in a sequence while non-swap mechanism do not perform data exchange on the original records, but copy sorted records to an extra place. Some traditional sorting algorithms such as selection sort, generally choose in-place update to exchange the positions of two records. The in-place update method leads to a better data locality, but swapping a pair of records requires two writes in PM while inserting one selected record into the newly allocated PM space only costs one write in PM. In addition, in-place update contributes to more local writes, leading to bad wear-leveling and reducing the lifetime for PM. Therefore, in addition to some traditional sorting algorithms in swap, performance of sorts without swap are also need to be studied. Here, we use selection sort as an example to compare the difference between swap and non-swap. Fig. 19 shows the difference between swap and non-swap selection sort and Fig. 20 illustrates the performance gap between these two methods. Obviously, for the same sequence to be sorted, the execution time of non-swap is about 3–4 times that of swap. For an adaptive sorting engine, swap sorting method is more preferred when wear-leveling is not crucial. By contrast, PMSort can simply take non-swap approaches.

6.2. Multi-thread sorting

With the rapid growth of data set size, multi-thread processing method is widely applied into many systems and applications. Different from some state-of-the-art KV-Stores designed for persistent memory, there is no delete operation in these sorting methods since all records are inserted in order. Thus, multithreading seems simpler for sorting engine design. For tree-based structure sorting methods such as B*-sort and B+-tree (B+-tree internally sorts the records within one B+-Tree node for each insert operation), multiple threads can cooperate to construct the same tree. For B+-tree, lock should be employed when internal nodes or leaf nodes split and for B*-sort, PMSort only need to lock the parent of the inserted node, to avoid insertion in the same address concurrently.

For traditional sorting methods such as quick sort and external sort, similar to MapReduce in spirit, partition and merging is an appropriate solution for multiple threads to achieve concurrency and improve the effectiveness of PMSort. Specifically, PMSort divide records into several part and each thread is responsible for sorting one part. After that, PMSort merges all records and outputs a sorted sequence in the end. Note that the store latency would be higher for multiple threads than that for single thread, which may change the choices of PMSort. We leave detailed multi-thread design for future work.

7. Related work

Due to the interesting features of emerging persistent memory technologies, a few researches have been proposed to optimize the sorting performance for PM. For example, segment sort [20] assumes that a proper ratio between selection sort and external sort will lead to a better performance in PM, by trading off slower write operations for much faster read operations in PM. However, we have demonstrated that segment sort is consistently worse than quick sort on the Optane-based platform. B*-sort [19] utilizes the binary search tree structure to restrict the write complexity to $O(N)$ (i.e., write-once property) and maintains the average read complexity to $O(N\,log\,N)$. It also develops a tunnel list structure and adds register metadata to optimize the worst-case read complexity to $O(N\,log\,N)$ as well. Unfortunately, it is observed that although B*-sort has better wear-leveling effect, it is slower than the simple quick sort algorithm on the Optane-based platform. Based on a heap structure, Luo et al. [21] place nodes near the heap root in DRAM and those near the leaves in PM to reduce PM writes according to the observation that nodes near the root are more likely to be read or written. However, it runs on a DRAM-simulated platform and the write latency it sets is far longer than real PM. On the Optane-based platform, it performs worse than quick sort in our experiments. Compared to these PM-friendly sorting technique proposals, PMSort provides a more comprehensive solution for the best-level sorting performance under different conditions.

Sorting is a significant function in many storage systems and index structures. The representative is B+-Tree, which internally sorts the records within one B+-Tree node for each insert operation. Some PM-optimized B+-Trees [24,25,32] have developed efficient techniques to minimize the sorting overhead. For instance, wB+-Tree [24] utilizes the indirection slot array, which is similar to our pointer-indirect mechanism in spirit, to avoid the actual sorting for records, and hence reduces a lot of PM write overhead. The limitation of the indirection array, however, is that the indirection number is limited (e.g., 8 or 16 in wB+-Tree). NV-Tree [32] only sorts records for In-DRAM inner nodes but leaves the records in In-PM leaf nodes out of order, thus totally avoiding the sorting overhead in PM. Each insert operation in NV-Tree just appends a new log to the last record (i.e., a log). The tradeoff is the extra overhead of probing the entire node for each single read and the garbage collection overhead for invalid log records. Compared to the sorting techniques proposed in these B+-Trees, PMSort is a more universal sorting engine, rather than being limited to sort only a small number of records (e.g., only in a B+-Tree node).

8. Conclusion

In this paper, we make a systematic study on sorting in PM and point out that existing sorting methods have limitations when using the real PM product. We propose an adaptive sorting engine, PMSort, which can dynamically adjust its internal sorting technique to the corresponding condition to achieve the best-level performance. The experimental evaluation demonstrates the merit of PMSort. We hope that PMSort can inspire further researches in the area of PM sorting and we believe that more intelligent decisions on proper sorting techniques should be explored.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work is supported by National Key Research & Development Program of China (No. 2018YFB1003302) and Natural Science Foundation of Shanghai (No. 21ZR1433600).

References

[1] Volos Haris, Andres Jaan Tack, Michael M. Swift, Mnemosyne: Lightweight persistent memory, ACM SIGARCH Comput. Archit. News 39 (1) (2011) 91–104.

[2] K. Huang, S. Li, L. Huang, K. Tan, H. Mei, Lewat: A lightweight, efficient, and wear-aware transactional persistent memory system, IEEE Trans. Parallel Distrib. Syst. 32 (03) (2021) 649–664.

[3] J. Coburn, A. Caulfield, A. Akel, et al., NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories, in: Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, 2011, pp. 105-118.

[4] R. Dulloor Subramanya, et al., System software for persistent memory, in: Proceedings of the Ninth European Conference on Computer Systems, 2014.

[5] J. Xu, S. Swanson, NOVA: A log-structured file system for hybrid volatile / non-volatile main memories, in: Proceedings of the 14th USENIX Conference on File and Storage Technologies, 2016, pp. 323-338.

[6] S. Zheng, M. Hoseinzadeh, S. Swanson, Ziggurat: A tiered file system for non-volatile main memories and disks, in: Proceedings of the 17th USENIX Conference on File and Storage Technologies, 2019, pp. 207-219.

[7] Xia Fei, et al., Hikv: A hybrid index key–value store for dram-nvm memory systems, in: 2017 USENIX Annual Technical Conference, 2017.

[8] Y. Huang, et al., Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs, in: 2018 USENIX Annual Technical Conference, 2018.

[9] J. Kim, S. Lee, J.S. Vetter, PapyrusKV: a high-performance parallel key–value store for distributed NVM architectures, in: International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '17, 2017, pp. 1–14.

[10] O. Kaiyrakhmet, S. Lee, B. Nam, S.H. Noh, Y. Choi, SLM-DB: Single-level key-value store with persistent memory, in: Proceedings of the 17th USENIX Conference on File and Storage Technologies, 2019, pp. 191–205.

[11] J. DeBrabant, J. Arulraj, et al., A prolegomenon on OLTP database systems for non-volatile memory, in: Proceedings of the VLDB Endowment, vol. 7(14), 2014, pp. 57-63.

[12] S. Kuznetsov, Towards a native architecture of In-NVM DBMS, on: 2019 Actual Problems of Systems and Software Engineering (APSSE), Moscow, Russia, 2019, pp. 77–89, 10.1109/APSSE47353.2019.00017.

[13] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, O. Mutlu, Evaluating STT-RAM as an energy-efficient main memory alternative, in: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Austin, TX, 2013, pp. 256–267, http://dx.doi.org/10.1109/ISPASS.2013.6557176.

[14] H.-S.P. Wong, S. Raoux, et al., Phase change memory, Proc. IEEE 98 (12) (2010) 2201–2227, http://dx.doi.org/10.1109/JPROC.2010.2070050.

[15] F.T. Hady, A. Foong, B. Veal, D. Williams, Platform storage performance with 3D xpoint technology, Proc. IEEE 105 (9) (2017) 1822–1833, http://dx.doi.org/10.1109/JPROC.2017.2731776.

[16] K. Qureshi Moinuddin, et al., Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling, in: 2009 42nd Annual IEEE/ACM international symposium on microarchitecture (MICRO), 2009.

[17] Psaropoulos Georgios, et al., Bridging the latency gap between NVM and DRAM for latency-bound operations, in: Proceedings of the 15th International Workshop on Data Management on New Hardware, 2019.
[18] Huang Kaixin, Yijie Mei, Linpeng Huang, Quail: Using NVM write monitor to enable transparent wear-leveling, J. Syst. Archit. 102 (2020) 101658.
[19] Yu-Pei Liang, et al., B*-sort: Enabling write-once sorting for persistent memory, IEEE Trans. Computer-Aided Design Integrated Circ. Syst. (99) (2020) 1.
[20] Stratis D. Viglas, Write-limited sorts and joins for persistent memory, in: Proceedings of the VLDB Endowment, vol. 7(5), 2014, pp. 413-424.
[21] Y. Luo, Z. Chu, P. Jin, S. Wan, Efficient sorting and join on NVM-based hybrid memory, in: Algorithms and Architectures for Parallel Processing, 20th International Conference, ICA3PP 2020, New York City, NY, USA, October 2–4, Proceedings, Part I, 2020, pp. 15–30.
[22] J. Yang, J. Kim, M. Hoseinzadeh, et al., An empirical guide to the behavior and use of scalable persistent memory, in: Proceedings of the 18th USENIX Conference on File and Storage Technologies, 2020, pp. 168-182.
[23] I.B. Peng, M.B. Gokhale, E.W. Green, System evaluation of the Intel optane byte-addressable NVM, in: Proceedings of the International Symposium on Memory Systems, 2019, pp. 304-315.
[24] S. Chen, Q. Jin, Persistent b+-trees in non-volatile main memory, PVLDB 8 (7) (2015) 786–797.
[25] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, W. Lehner, Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory, in: Proceedings of the 2016 International Conference on Management of Data, 2016, pp. 371–386.
[26] M. Woniak, Z. Marszałek, M. Gabryel, R.K. Nowicki, Preprocessing large data sets by the use of quick sort algorithm, in: Skulimowski A. Kacprzyk J. (Ed.), Knowledge, Information and Creativity Support Systems: Recent Trends, Advances and Solutions, in: Advances in Intelligent Systems and Computing, vol. 364, Springer, Cham, 2016.
[27] J.B. Hayfron-Acquah, Obed. Appiah, K. Riverson, Improved selection sort algorithm, Int. J. Comput. Appl. (0975–8887) 110 (5) (2015) 29–33.
[28] Z. Marszałek, Parallelization of modified merge sort algorithm, Symmetry 9 (9) (2017) 176.
[29] E. Khorasani, B.D. Paulovicks, et al., Parallel implementation of external sort and join operations on a multi-core network-optimized system on a chip, in: Xiang Y. Cuzzocrea A. Hobbs M. Zhou W. (Ed.), Algorithms and Architectures for Parallel Processing. ICA3PP 2011, in: Lecture Notes in Computer Science, vol. 7016, Springer, Berlin, Heidelberg, 2011.
[30] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, Haibo Chen, Performance and protection in the ZoFS user-space NVM file system, in: ACM SIGOPS 27th Symposium on Operating Sys- Tems Principles (SOSP '19), October (2019) 27–30, Huntsville, on, Canada, ACM, New York, NY, USA, 2019, p. 16, http://dx.doi.org/10.1145/3341301.3359637.
[31] Mingkai Dong, Haibo Chen, Soft updates made simple and fast on non-volatile memory, in: Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17), USENIX Association, Berkeley, CA, USA, 2017, pp. 719–731.
[32] J. Yang, Q. Wei, C. Chen, C. Wang, K.L. Yong, B. He, NV-Tree: Reducing consistency cost for NVM-based single level systems, in: Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST '15, 2015, pp. 167–181.
[33] S. Venkataraman, N. Tolia, et al., Consistent and durable data structures for non-volatile byte-addressable memory, in: Proceedings of the 9th USENIX Conference on File and Storage Technologies, FAST '11, 2011, p. 5.
[34] Valgrind, https://www.valgrind.org.
[35] H. Garcia-Molina, K. Salem, Main memory database systems: an overview, IEEE Trans. Knowl. Data Eng. 4 (6) (1992) 509–516, http://dx.doi.org/10.1109/69.180602.
[36] Xu Cheng, Jiangchuan Liu, Haiyang Wang, Accelerating YouTube with video correlation, in: Proceedings of the first SIGMM workshop on Social media (WSM '09). Association for Computing Machinery, New York, NY, USA, 2009, pp. 49–56, 10.1145/1631144.1631156.
[37] Sort benchmark home page, 2020, http://sortbenchmark.org/ (Accessed 27 2020).

Yifan Hua is currently a Ph.D. student at Shanghai Jiao Tong University, China. His research interests include non-volatile memory and high bandwidth memory management.

Kaixin Huang is currently a Ph.D. student at Shanghai Jiao Tong University, China. He received his bachelor degree from University of Electronic Science and Technology of China, in 2016. His research interests include non-volatile memory management and distributed systems.

Shengan Zheng received his Ph.D. and B.S. degrees from Shanghai Jiao Tong University in 2019 and 2014, respectively. He is currently a postdoctoral researcher in the Storage Research Group at Tsinghua University. His research interests lie in the area of non-volatile memory and file systems.

Linpeng Huang received his M.S. and Ph.D. degrees in computer science from Shanghai Jiao Tong University in 1989 and 1992, respectively. He is a professor of computer science in the department of computer science and engineering, Shanghai Jiao Tong University. His research interests include distributed systems and service oriented computing. He is a senior member of the IEEE.