Redesigning the Sorting Engine for Persistent Memory

Yifan Hua[1], Kaixin Huang[1], Shengan Zheng[2], and Linpeng Huang[1](✉)

[1] Shanghai Jiao Tong University, Shanghai, China
{huahuahuahua,Kaixinhuang,huang-lp}@sjtu.edu.cn
[2] Tsinghua University, Beijing, China
venero@tsinghua.edu.cn

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. Such amazing features have inspired a host of PM-based storage systems and applications that store and access data directly in PM. Sorting is an important function for many systems, but how to optimize sorting for PM-based systems has not been systematically studied yet. 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 real PM 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 propose an adaptive sorting engine, namely SmartSort, to optimize the sorting performance for different conditions. The experimental results demonstrate that SmartSort remarkably outperforms existing sorting methods in a variety of cases.

Keywords: Persistent memory · Sorting algorithm · Pointer-indirect · Wear-leveling

1 Introduction

Emerging persistent memory (PM) is a new type of non-volatile storage device. Unlike traditional SSD, HDD or Flash, PM technologies such as STT-RAM [1], PCM [2] and 3DXPoint [3] can provide byte-addressability, DRAM-comparable read and write latency. Applications can access data in PM with simple load/store instructions. PM has inspired a host of researches on redesigning persistent storage systems, such as memory management systems [9,28,29], file systems [7,8,30], KV-Stores [31,34] and DBMSs [10,35].

For many storage systems, sorting is one of the most commonly-used functions. For instance, the 'ORDER BY' SQL command will automatically call

© Springer Nature Switzerland AG 2021
C. S. Jensen et al. (Eds.): DASFAA 2021, LNCS 12683, pp. 393–412, 2021.
https://doi.org/10.1007/978-3-030-73200-4_27

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 [5,6,32], 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 Sect. 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.

With a specific study 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 the Intel Optane DC Persistent Memory (Optane) platform (see more details in Sect. 3).

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) [5,6,32], a few researchers have proposed PM-friendly sorting methods [12,13,36] to decrease PM writes. For example, segment sort [12] intends to trade off fewer writes for additional reads and allows a tunable combination of external sort and selection sort. The

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

[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.

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 [13] 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. [36] 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, 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 Sect. 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 [14] 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-addressability 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. We propose a pointer-indirect mechanism in this paper to speed up the sorting performance for large-size records. 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, SmartSort. SmartSort can automatically pick the best-suited embedded sorting method in an ad-hoc style. 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 PM. We demonstrate that existing sorting methods have non-negligible limitations.
- Taking advantage of PM's byte-addressability, we propose a pointer-indirect sort mechanism, which not only reduces the PM read and write overhead, but also does good to PM wear-leveling.
- Combined with the advantages of various sorting methods, we develop an adaptive sorting engine, namely SmartSort, to minimize the sorting overhead in PM for different conditions.

- We conduct extensive experiments on the Optane platform and the results show that SmartSort remarkably outperforms existing sorting methods for various workloads and situations.

The rest of this paper is organized as follows. Section 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 SmartSort, in detail. We evaluate SmartSort in Sect. 5 and discuss related work in Sect. 6. In Sect. 7, we finally conclude this paper.

2 Background

2.1 Persistent Memory

Persistent Memory (PM) such as PCM [2], STT-RAM [1] and 3DXPoint [3] is a new type of memory technology that has large capacity, non-volatility, byte-addressability, and limited write endurance [5,6]. 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 [30]. Intel Optane DC Persistent Memory (Optane for short) is the first commercially-available PM product [4]. The emergence of PM has inspired a lot of researches for building persistent storage systems and applications [7,9,28,29].

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 [10,35], KV-Stores [31,34], and B+-Trees [19,20]. 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 $O(NlogN)$).

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

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	$O(NlogN)$	$O(NlogN)$	$O(logN)$
Merge sort	$O(NlogN)$	$O(NlogN)$	$O(N)$
Heap sort	$O(NlogN)$	$O(NlogN)$	$O(1)$

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 [22], selection sort [23], merge sort [24], and external sort [25] 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 [12,13,36] 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 [12] 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 [13] develops a binary tree-based structure for sorting records in PM, which has $O(N)$ complexity for writes and $O(NlogN)$ complexity for reads. B*-sort also uses extra tunnel lists and register metadata to optimize the worst-case read complexity. Luo et al. [36] 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.

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 Sect. 5.1.

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

	10K	100K	1M	10M	100M	1B
Selection sort	123	12195	1232723	Too long	Too long	Too long
Insertion sort	186	18376	1911156	Too long	Too long	Too long
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

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

	In-PM				In-DRAM				PM-DRAM			
	8B	64B	512B	4 KB	8B	64B	512B	4 KB	8B	64B	512B	,KB
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	Too long	18.4	27.6	127.9	Too long	18.4	27.6	127.9	Too long
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

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. 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; the In-DRAM mechanism indicates that the records

are loaded into and sorted in DRAM (but not stored back to PM); the PM-DRAM mechanism represents that the records are loaded into and sorted in DRAM, and finally stored back to PM.

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.

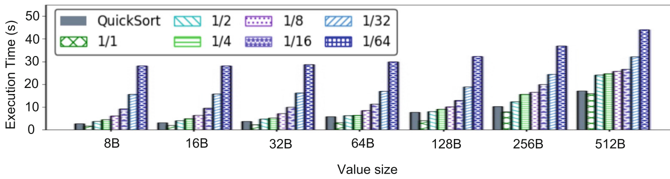


Fig. 1. The impact of available relative DRAM capacity on external sort.

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. Figure 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.

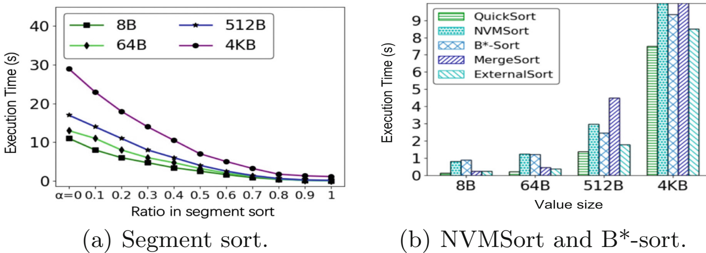
From Fig. 1, we can observe that the performance of external sort decreases with the relative DRAM capacity being smaller, but the sharp performance drop is mainly in the shift from full record capacity to 1/2 record capacity. For instance, with 8-byte value size, the time consumption climbs by 2.2x, when

changing the DRAM capacity from full record space to 1/2 record space. However, only 18% extra time is incurred when shifting the DRAM capacity form 1/2 record space to 1/4 record space. Second, external sort is better than (In-PM) quick sort in performance with sufficient DRAM capacity and worse than (In-PM) quick sort if the DRAM capacity 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 [12], B*-sort [13] and NVMSort [36] 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. [12] 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. Figure 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 complexity but lower read complexity than selection sort. The merging phase is kept as the previous design.



(a) Segment sort. (b) NVMSort and B*-sort.

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

In Fig. 2(a), $\alpha = 0$ means that segment sort only uses selection sort; by contrast, $\alpha = 1$ 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 [14] 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 10 million 16-byte records, quick sort takes 304.99 s while random write takes merely 4.32 s. 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 [13] 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 $O(NlogN)$. 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. Figure 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 has a lot of non-adjacent data swap. They not only fail to utilize cache locality but also incur severe time overhead [3,4]; 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 SmartSort

As Sect. 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 SmartSort, to address the challenges we mentioned. In this section, we first present the overall structure of SmartSort, then introduce the PM-enabled pointer-indirect sort mechanism, and finally provide the details of how SmartSort works.

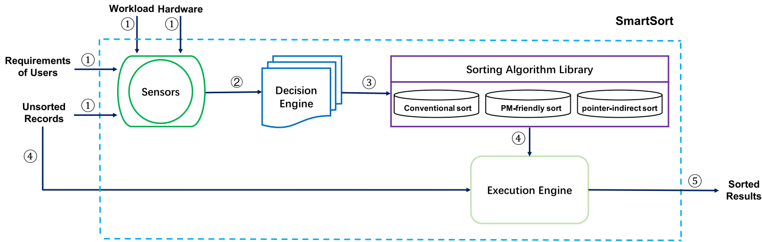


Fig. 3. Architecture of SmartSort.

4.1 Overview

SmartSort 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. Figure 3 shows the architecture of SmartSort. SmartSort 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 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 Sect. 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.

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) 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 Sect. 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 [37], 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.

The key step of the the pointer-indirect sort mechanism is building the mapping <key, pointer> records, based on the original <key, value> records. Concretely, a new region (either in DRAM or PM) 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.

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 Sect. 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. 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

Table 4. Complicated sorting conditions and corresponding sorting methods.

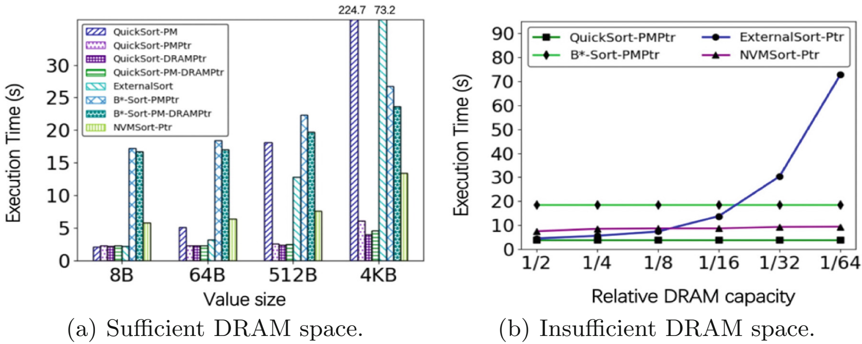
Value size	Sufficient DRAM	Wear-leveling	Persistence	Best-suited sorting method
small (large)	Yes	Yes	Yes	(pointer-indirect) PM-DRAM quick sort
	Yes	Yes	No	(pointer-indirect) In-DRAM quick sort
	Yes	No	Yes	(pointer-indirect) In-PM quick sort
	Yes	No	No	(pointer-indirect) In-DRAM quick sort
	No	Yes	Yes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

quick sort, external sort and B*-sort. As shown in Fig. 3, pointer-indirect sort is a core mechanism of the *SAL* component.

4.3 Adaptive Sorting

To achieve the best sorting performance with SmartSort, 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 than 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.



(a) Sufficient DRAM space. (b) Insufficient DRAM space.

Fig. 4. Comparison of the execution time for different sorting methods.

The corresponding best-suited sorting method is carefully selected based on our experimental observations. As we discuss in Sect. 4.2, the pointer-indirect sort mechanism should gain many performance benefits when the value size is large. Figure 4(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 4KB (e.g., 2.2x, 6.9x and 36.8x for 64B, 512B and 4KB 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. 4(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) outperforms QuickSort-PMPtr (i.e., pointer-indirect In-PM quick sort) by up to 1.4x when the value size is 4KB. In addition,

sorting in DRAM will reduce writes in PM a lot. Therefore, when wear-leveling is required, SmartSort 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, SmartSort can simply choose In-PM quick sort and In-DRAM quick sort according to different persistence requirements.

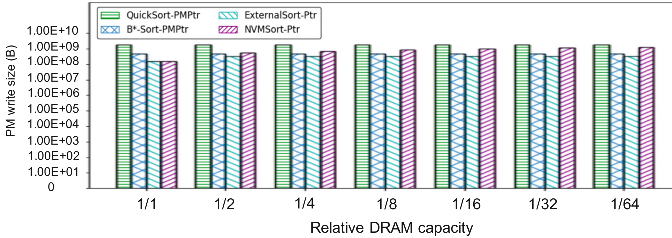


Fig. 5. Write size in PM for different pointer-indirect sorting algorithms.

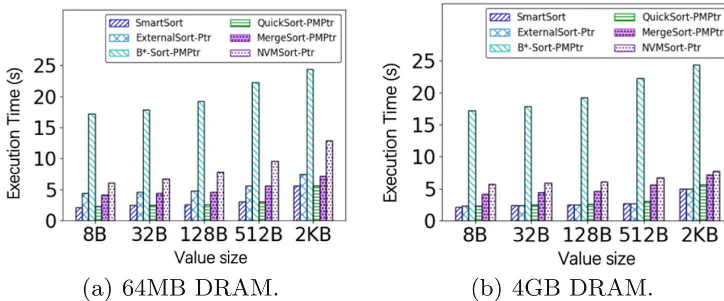
When DRAM capacity is not sufficient, and if there is no restriction on wear-leveling, then it could be very straightforward to select the In-PM quick sort due to its efficiency. However, if wear-leveling is set as a target, then SmartSort should shift the sorting method. Notice that both external sort 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. Figure 4(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. 4(b), its PM write size is comparable to that of QuickSort-PMPtr in that case as Fig. 5 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, SmartSort 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 SmartSort is the best-suited one for the corresponding condition. Notice that the selection for a sorting method in SmartSort each time is not manually-configured. That is why we call SmartSort an adaptive sorting engine.

5 Experimental Evaluation

5.1 Experimental Setup

We implement SmartSort 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 DIMM space in the maximum to ensure all records accommodated in this paper. DRAM size in Sect. 3 and 4 varies according to the record number, value size and relative ratio to PM as detailed in each picture. In Sect. 5, DRAM size is fixed as 4GB (large enough to store all records or pointers), 64 MB (nearly 1/3 of the total pointer size) and 4 MB (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 4KB. To guarantee data persistence and consistency in PM, similar to many prior works [7,28], we properly utilize *clwb+sfence* instructions to force flushing out the records from caches. We use the standard benchmarks [27] to evaluate the sorting performance of SmartSort. 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 SmartSort, we compare SmartSort 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).



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

Fig. 6. The execution time for different sorting algorithms.

5.2 Sorting Performance for Different Workloads

Figure 6 compares the execution time for sorting ten million records between SmartSort and other sorting methods. The DRAM capacity in Fig. 6(a) and Fig. 6(b) is 64 MB (i.e., insufficient DRAM space) and 4 GB (sufficient DRAM space), respectively. We can observe that SmartSort is remarkably better than the other sorting methods and has good scalability with the value size growing.

For 64 MB DRAM capacity, SmartSort will adaptively select quick sort for 8-byte value size and pointer-indirect quick sort in PM for larger value size when wear-leveling is not a restricted factor. For 4 GB DRAM capacity, SmartSort 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.

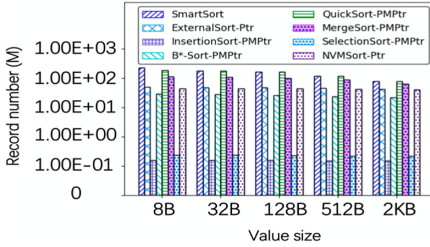


Fig. 7. Sorted record number in 1 min.

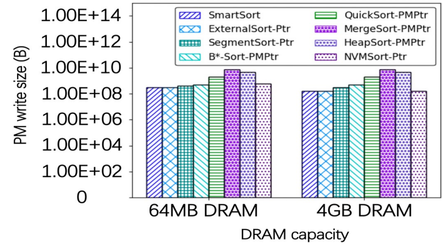
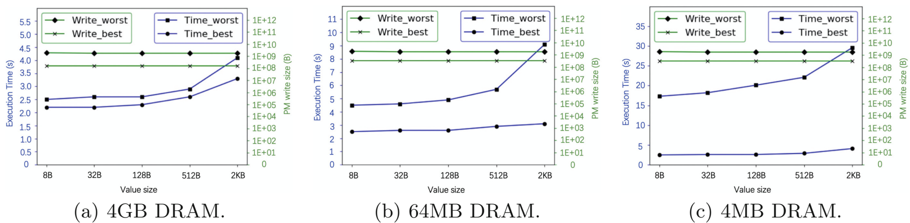


Fig. 8. Write size in PM.

Figure 7 shows the maximum number of records that can be sorted in one minute by different sorting methods. We set the DRAM capacity as 64 MB, which is insufficient to contain all records. In this case, SmartSort 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. Figure 8 shows the total PM write size (in bytes) of sorting ten million records for the compared methods when DRAM is 64 MB and 4 GB. For 64 MB DRAM capacity, SmartSort will adaptively select pointer-indirect external sort. For 4 GB DRAM capacity, SmartSort will adaptively select pointer-indirect PM-DRAM quick sort (or In-DRAM quick sort if there is no need to persist the sorted results).

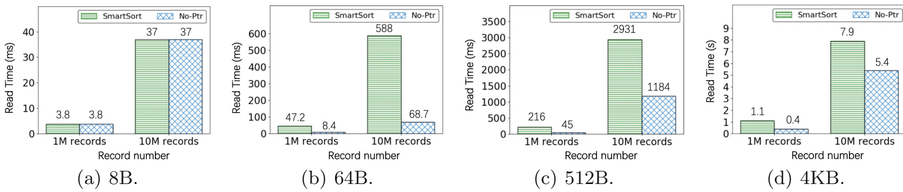


(a) 4GB DRAM. (b) 64MB DRAM. (c) 4MB DRAM.

Fig. 9. The best and worst performance to persist sorted result.

Given ten million records, Fig. 9 provides the upper bound and lower bound of SmartSort 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 SmartSort'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 $O(NlogN)/O(N)$.



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

Fig. 10. Time overhead of reading sorted records for different value size.

5.3 Comparison of Time Overhead on Reading Sorted Records

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). Figure 10 compares the time overhead between SmartSort 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 SmartSort. There are two reasons. First, the pointer-indirect mechanism employed by SmartSort 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 SmartSort 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 4KB values), and the sorting performance is improved by 12.3x. Therefore, the overall request overhead can be remarkably reduced by SmartSort.

6 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 [12] 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 [13] 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 $O(NlogN)$. It also develops a tunnel list structure and adds register metadata to optimize the worst-case read complexity to $O(NlogN)$ 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. [36] 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, SmartSort 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 [18–20] have developed efficient techniques to minimize the sorting overhead. For instance, wB+-Tree [19] 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 [18] 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 trade-off 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, SmartSort 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).

7 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, SmartSort, 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 SmartSort. We hope that SmartSort can inspire further researches in the area of PM sorting and we believe that more intelligent decisions on proper sorting techniques should be explored.

Acknowledgment. This work is supported by National Key Research & Development Program of China (No. 2018YFB1003302). Linpeng Huang is the corresponding author of this paper and also supported by SJTU-Huawei Innovation Research Lab Funding (No. FA2018091021-202004). Shengan Zheng is supported by China Postdoctoral Science Foundation (No. 2020M680570).

References

1. Kültürsay, E., Kandemir, M., Sivasubramaniam, A., Mutlu, O.: Evaluating STT-RAM as an energy-efficient main memory alternative. In: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Austin, TX, pp. 256–267 (2013)
2. Wong, H.-S.P., Raoux, S., et al.: Phase change memory. Proc. IEEE **98**(12), 2201–2227 (2010)
3. Hady, F.T., Foong, A., Veal, B., Williams, D.: Platform storage performance With 3D XPoint technology. Proc. IEEE **105**(9), 1822–1833 (2017)
4. Peng, I.B., Gokhale, M.B., Green, E.W.: System evaluation of the Intel optane byte-addressable NVM. In: Proceedings of the International Symposium on Memory Systems, pp. 304–315 (2019)
5. Qureshi, M.K., 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)
6. Huang, K., Mei, Y., Huang, L.: Quail: using NVM write monitor to enable transparent wear-leveling. J. Syst. Archit. **102**, 101658 (2020)
7. Xu, J., Swanson, S.: 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, pp. 323–338 (2016)
8. Zheng, S., Hoseinzadeh, M., Swanson, S.: Ziggurat: a tiered file system for non-volatile main memories and disks. In: Proceedings of the 17th USENIX Conference on File and Storage Technologies, pp. 207–219 (2019)
9. Coburn, J., Caulfield, A., Akel, A., 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, pp. 105–118 (2011)
10. Kaiyrakhmet, O., Lee, S., Nam, B., Noh, S.H., Choi, Y.: SLM-DB: single-level key-value store with persistent memory. In: Proceedings of the 17th USENIX Conference on File and Storage Technologies, pp. 191–205 (2019)
11. Seo, J., Kim, W.-H., Baek, W., Nam, B., Noh, S.H.: Failure-atomic slotted paging for persistent memory. SIGARCH Comput. Archit. News **45**(1), 91–104 (2017)
12. Viglas, S.D.: Write-limited sorts and joins for persistent memory. Proc. VLDB Endow. **7**(5), 413–424 (2014)
13. Liang, Y.-P., et al.: B*-sort: enabling Write-once Sorting for persistent memory. IEEE Trans. Comput.-Aided Design Integr. Circ. Syst. **PP**(99), 1 (2020)
14. Yang, J., Kim, J., Hoseinzadeh, M., 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, pp. 168–182 (2020)
15. Hwang, D., Kim, W., Won, Y., Nam, B.: Endurable transient inconsistency in byte-addressable persistent B+-tree. In: Proceedings of the 16th USENIX Conference on File and Storage Technologies, pp. 187–200 (2018)

16. Chen, Y., Lu, Y., Fang, K., Wang, Q., Shu, J.: uTree: a persistent B+-tree with low tail latency. Proc. VLDB Endow. **13**(12), 2634–2648 (2020)
17. Liu, M., Xing, J., Chen, K., Wu, Y.: Building scalable NVM-based B+tree with HTM. In: Proceedings of the 48th International Conference on Parallel Processing, pp. 1–10 (2019)
18. Yang, J., Wei, Q., Chen, C., Wang, C., Yong, K.L., He, B.: NV-Tree: reducing consistency cost for NVM-based single level systems. In: Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, pp. 167–181 (2015)
19. Chen, S., Jin, Q.: Persistent B+-trees in non-volatile main memory. PVLDB **8**(7), 786–797 (2015)
20. Oukid, I., Lasperas, J., Nica, A., Willhalm, T., Lehner, W.: 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, pp. 371–386 (2016)
21. Andrei, M., Lemke, C., et al.: Sorting with asymmetric read and write costs guy. In: Annual ACM Symposium on Parallelism in Algorithms and Architectures, vol. 2015, no. 6, pp. 1–12 (2015)
22. Woźniak, M., Marszałek, Z., Gabryel, M., Nowicki, R.K.: Preprocessing large data sets by the use of quick sort algorithm. In: Skulimowski, A.M.J., Kacprzyk, J. (eds.) Knowledge, Information and Creativity Support Systems: Recent Trends, Advances and Solutions. AISC, vol. 364, pp. 111–121. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-19090-7_9
23. Hayfron-Acquah, J.B., Appiah, O., Riverson, K.: Improved selection sort algorithm. Int. J. Comput. Appl. (0975–8887) **110**(5), 29–33 (2015)
24. Marszałek, Z.: Parallelization of modified merge sort algorithm. Symmetry **9**(9), 176 (2017)
25. Khorasani, E., Paulovicks, B.D., Sheinin, V., Yeo, H.: 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. (eds.) ICA3PP 2011. LNCS, vol. 7016, pp. 318–325. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24650-0_27
26. Quartz. https://github.com/HewlettPackard/quartz. Accessed 27 Oct 2020
27. Sort Benchmark Home Page. http://sortbenchmark.org/. Accessed 27 Oct 2020
28. Huang, K., Li, S., Huang, L., Tan, K., Mei, H.: Lewat: a lightweight, efficient, and wear-aware transactional persistent memory system. IEEE Trans. Parallel Distrib. Syst. **32**(03), 649–664 (2021)
29. Volos, H., Tack, A.J., Swift, M.M.: Mnemosyne: lightweight persistent memory. ACM SIGARCH Comput. Archit. News **39**(1), 91–104 (2011)
30. Dulloor, S.R., et al.: System software for persistent memory. In: Proceedings of the Ninth European Conference on Computer Systems (2014)
31. Xia, F., et al.: HiKV: a hybrid index key-value store for DRAM-NVM memory systems. In: 2017 USENIX Annual Technical Conference (2017)
32. Psaropoulos, G., 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)
33. Wan, H., et al.: Empirical study of redo and undo logging in persistent memory. In: 2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA) (2016)
34. Huang, Y., et al.: Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In: 2018 USENIX Annual Technical Conference (2018)

35. DeBrabant, J., Arulraj, J., et al.: A prolegomenon on OLTP database systems for non-volatile memory. Proc. VLDB Endow. **7**(14), 57–63 (2014)
36. Luo, Y., Chu, Z., Jin, P., Wan, S.: Efficient sorting and join on NVM-based hybrid memory. In: Qiu, M. (ed.) ICA3PP 2020, Part I. LNCS, vol. 12452, pp. 15–30. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-60245-1_2
37. Garcia-Molina, H., Salem, K.: Main memory database systems: an overview. IEEE Trans. Knowl. Data Eng. **4**(6), 509–516 (1992)