# NVMSorting: Efficient Sorting on Non-Volatile Memory

Zhaole Chu, Yongping Luo, Peiquan Jin, Shouhong Wan
[1]*School of Computer Science and Technology, University of Science and Technology of China, Hefei, China*
[2]*Key Lab. of Electromagnetic Space Information, Chinese Academy of Sciences, Hefei, China*
jpq@ustc.edu.cn

***Abstract*—Non-volatile memory (NVM) as a new type of storage technology has many advantages such as non-volatility, byte addressability, high storage-density, and low energy consumption. Meanwhile, NVM has some limitations, e.g., asymmetric read and write latency, limited write endurance, and high price. Therefore, at present, it is not realistic to completely replace DRAM with NVM in computer systems. A more feasible scheme is to adopt the hybrid memory architecture composed of NVM and DRAM. Following the assumption of hybrid memory architecture, this paper proposes an NVM-friendly sorting algorithm called NVMSorting. Particularly, we introduce a new concept called *natural runs* to improve the existing MONTRES algorithm and present the cost analysis of the algorithm in the hybrid memory architecture. In order to verify the performance of our proposal, we implement six existing sorting algorithms as baselines, including the MONTRES algorithm, and conduct comparative experiments on an unsorted dataset and a partially sorted dataset. The experimental results suggest the efficiency of NVMsorting in terms of execution time and NVM writes. Especially, on the partially sorted dataset, NVMSorting has 6.2% improvement on time performance and 5.7% reduction on NVM writes compared to MONTRES, and 13.0% performance improvement and 27.1% NVM-write reduction compared to the traditional merge sorting algorithm.**

***Keywords*—Non-Volatile Memory, Hybrid Memory, Sorting Algorithm**

## I. Introduction

Non-volatile memory (NVM) such as Phase Change Memory (PCM) is one of the research hotspots in recent years, and is also considered as a hot candidate for the next generation of storage technology. NVM has some special properties [1]–[3]. First of all, differing from DRAM, it is non-volatile, meaning that all data written into NVM will not be lost when the host computer is shut down. Second, differing from magnetic disks or solid-state drives (SSD) that only support block-based data accesses, NVM is byte addressable, which is similar to DRAM. Third, the density of NVM is generally higher than that of DRAM and is comparable to that of SSD. To this end, NVM has the advantages of both disks and DRAM. However, NVM also has some limitations compared to DRAM and disks. Firstly, the read and write latency of NVM is not balanced. Particularly, NVM has the similar read latency as DRAM, but its write latency is higher than that of DRAM.

DOI reference number: 10.18293/SEKE2021-141

TABLE I
Comparison of SSD, DRAM, and NVM

| | SSD | DRAM | NVM |
|---|---|---|---|
| Read Latency | $25\mu s$ | $20-50ns$ | $50ns$ |
| Write Latency | $500\mu s$ | $20-50ns$ | $1\mu s$ |
| Endurance | $10^5$ | $\infty$ | $10^8$ |
| Density | High | Low | High |
| Non-Volatile | Yes | No | Yes |
| Byte-Addressable | No | Yes | Yes |

In addition, the endurance of NVM is limited, meaning that after a certain number of writes ( $10^8$ at present), NVM will become unstable. Thus, algorithms running on NVM have to be write-friendly. In summary, we list the main features of SSD, DRAM, and NVM in Table I.

Due to the limitations of NVM, currently it is not realistic to completely replace DRAM with NVM. A more feasible scheme is to adopt the hybrid memory architecture composed of NVM and DRAM.There are two kinds of hybrid memory architectures. The first type is the hierarchical architecture [2], which takes DRAM as the cache of NVM. In this architecture, only the DRAM space is recognized by the operating system as main memory and does not utilize the high density of NVM. In addition, it will introduce additional cost of data migration and consistency. The second type is parallel architecture [3], in which NVM and DRAM are both used as the main memory. In this way, we can make good use of the advantages of NVM, such as byte addressability and persistence. In addition, we can reduce the write operations to NVM by devising appropriate algorithms. Therefore, so far, the parallel architecture has received much attention in NVM-related research. In this study, we also adopt the parallel architecture.

Because of the emergence of NVM with high storage density, we can use NVM to replace the traditional disk as the persistent storage device and build a hybrid memory system without disk. Based on this inference, this paper studies the sorting algorithm in hybrid memory. At the same time, due to the read/write asymmetry of NVM, we need to reduce the writing operations of sorting as much as possible. Briefly, we make the following contributions in this study:

(1) We improve the existing MONTRES algorithm to make it suitable for NVM. MONTRES was designed as an external sorting algorithm on SSD. In this paper, we propose a new

concept of *natural runs* and devise a new NVM-friendly memory sorting algorithm called NVMSorting. Compared to the original MONTRES algorithm, the proposed NVMSorting can detect partially ordered runs and reduce the sorting cost.

(2) We present cost analysis of the NVMSorting algorithm to theoretically demonstrate that NVMSorting has a lower cost than existing sorting algorithms.

(3) We experimentally compare the time performance and the number of NVM writes of the NVMSorting algorithm with six existing sorting algorithms. The experimental results show that NVMSorting has better time performance and less NVM writes than existing sorting algorithms. In addition, NVMSorting achieves better performance on the partially ordered dataset, indicating that it can effectively detect the partially ordered runs.

## II. Related Work

To the best of our knowledge, few studies have been focused on the improvement of fundamental sorting algorithms on NVM.

The first study [4] presented a write-limited sorting algorithm within the context of database query processing. It proposed three write limited sorting algorithms, namely segment sort, hybrid sort and lazy sort. These algorithms are all based on trading writes for reads to achieve an optimal NVM memory cost. They either offer a knob to adjust the read and write ratio of an algorithm, or use a lazy mechanism to delay result materializing until the penalty outweighs the gains.

The second work [5] proposed a cost model for sorting on storage devices with asymmetric read and write latency. In this literature, the authors presented three sorting algorithms (merge sort, sample sort, and heap sort using buffer trees) for the asymmetric external memory model and gave a detailed cost analysis. However, this work is toward page-based storage devices, such as flash-memory-based SSDs. Although flash memory also has limited write endurance and low write latency, it is much different from NVM, because NVM can be used as main memory while flash memory can only be used as secondary storage.

In the literaure [6], the authors proposed a write-once sorting algorithm, named B*-sort. Differing from previous work, this work focused on pure NVM-based embedded system. Unlike the commonly used array-based sorting algorithms, the B*-sort adopts a new concept, tree-based sort which is inspired by the binary search tree that has the write-once property during tree construction. The algorithm can guarantee $O(n)$ writes. Because of the tunnel-list structure proposed by the author, B*-sort can also guarantee $O(n\sqrt{n})$ reads. Although the pure NVM memory system could be realistic in the future, many studies have reported that the hybrid memory architecture will be more realistic in next years. This is mainly because that the DRAM's speed is higher than that of NVM. Our study is also towards the hybrid memory architecture. Therefore, we will not compare the B*-sort algorithm in the experiment.

Recently, Luo et al. [7] proposed an optimal data placement model for solving the data placement issue on DRAM-NVM-based hybrid memory. They also developed a new sorting algorithm that adopted heap structures to accelerate the sorting process. However, this sorting algorithm was an in-memory sorting one that ran on the parallel memory architecture composed of DRAM and NVM, which is different from the memory architecture of this study.

## III. Design of NVMSorting

### A. Motivation

NVM and flash memory have similar defects, which is the read/write asymmetry and the limitation of endurance . Based on this observation, we can infer that a write efficient algorithm designed for SSD may have the same effect when it is transplanted to NVM. At present, the research of sorting algorithms for SSD has made great progress, and a state-of-art sorting algorithm for SSD is called MONTRES [8]. Therefore, this study aims to improve MONTRES to make it suitable for NVM-based hybrid memory architecture.

MONTRES was designed as an external sorting algorithm on SSD. Its main idea is to persist data to SSD as early as possible so that we need not partially write data to intermediate runs. To achieve this, they proposed the merge-on-the-fly mechanism and the run-expansion mechanism to generate runs as large as possible. MONTRES has been demonstrated to be helpful for accelerating the phase of merging runs. However, MONTRES was designed for paginated SSDs and is not suitable for byte-addressable NVM.

In the literature [9], researchers proposed the MONTRES-NVM algorithm to optimize MONTRES for NVM, which offers to detect all the sorted pieces in the original data set. Each sorted piece is treated as a sorted run and it is ignored in the run generation phase, to reduce useless NVM writes. However, in real world applications, sorted pieces tend to be short at length, which will lead to many tiny sorted runs and then end up with poor performance.

### B. Natural run

To address the problem mentioned above, we introduce a new concept of *natural run*, which is defined in definition 1. A natural run is composed of several blocks (items in a block need not to be sorted , while any two items in consecutive blocks are ordered). Figure 1 shows an example of a natural run. In this figure, 3 blocks of items which have no overlap value range are grouped together to be a natural run. A natural run can be treated as a sorted run in the merge phase but the actual sort procedure of each block is delayed until the last block is drained at merging. The concept of natural run exploits much longer pieces that can be ignored in run generation phase, so it can significantly reduce the NVM writes and improve overall performance.

Following this idea, we propose the NVMSorting algorithm leveraging natural runs. We will show that our NVMSorting

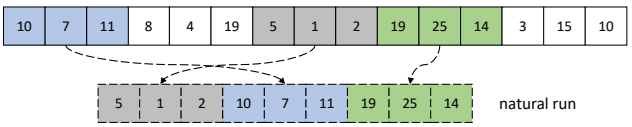


Fig. 1. Example of a natural run

can achieve better sorting performance than the MONTRES-NVM algorithm theoretically.

*Definition 1:* (*Natural Run*) A natural run is a sequence of blocks $S = \{b^1, b^2, ...b^x\}$. Each block $b$ contains identical number of items in value range [$b_{min}$, $b_{max}$]. For $\forall i, j$ that have $i < j$, it must hold that $b^i_{max} < b^j_{min}$.

### C. The NVMSorting Algorithm

The basic idea of NVMSorting algorithm is to detect natural runs first and then perform merge sort for the natural runs and normal runs. NVMSorting consists of three phases: (1) natural run detection, (2) run generation, and (3) run merge. Below, we will detail these phases.

(1) *Natural Run Detection*. This phase is designed to find the natural runs in the input data. This can avoid loading all input data into DRAM for sorting and then writing them back to NVM. As writes to NVM are much costly, the detection of natural runs help to reduce loading and writing back blocks into NVM. The definition of natural run indicates that an actual run tends to be longer than already sorted pieces used in MONTRES-NVM. So we can infer that natural runs are more common and useful than MONTRES-NVM.

Differing from the MONTRES algorithm that scans the data to build a Min-Index before generating runs, we record both the maximum and minimum values of each block when scanning the data to build a MinMax-Index. Each element of the MinMax-Index consists of the minimum and maximum value of one data block, and elements are sorted in ascending order. This index is then used to detect natural runs. The algorithm of detecting natural runs is shown in Algorithm 1. Specially, we detect the natural run with the first *n* elements as the starting point, and select the longest run as the result.

(2) *Run Generation*. In the run generation phase, we also use the merge-on-the-fly mechanism and the run expansion mechanism. Due to the existence of natural run, this phase is different from that of MONTRES. Here, we set block size to *M*, the size of the whole memory work space of $N \cdot M$, and divide the DRAM memory into two areas, namely $W_n$ and $W_s$, where $W_n$ is the work space for loading the natural run whose size is *M* and $W_s$ is the work space for loading other data whose size is ($N$-1) $\cdot$ *M*. Algorithm 2 shows the process of run generation.

Before run generation phase, we have scanned the data to get the MinMax-Index and natural run index. The algorithm loads the data into the work space in the order of the smallest elements until the data block in the MinMax-Index is exhausted (line 1). Only one block will be loaded to $W_n$

**Algorithm 1:** FindNaturalRun

**input :** MinMax-Index
**output:** NaturalRun-Index

```
1  for i ← 1 to n do
2      s=MinMax-Index[i];
3      TempResult={};
4      while s ≠ NULL do
5          TempResult.insert(s);
6          Locate the first t holds t.min > s.max;
7          s ← t;
8      end
9      if TempResult.size > NaturalRun-Index.size then
10         NaturalRun-Index ← TempResult;
11     end
12 end
13 delete elements which were inserted to
     NaturalRun-Index from MinMax-Index;
```

**Algorithm 2:** Run Generation

**input :** original data *S*, MinMax-Index, NaturalRun-Index

```
1  while Not Empty(MinMax-Index) do
2      if W_n is empty or the element in W_n is exhausted
         then
3          Load data from natural run to W_n and sort;
4      end
5      Load data to W_s and sort;
6      next-min ←next min value from MinMax-Index;
7      Merge on fly with W_n, W_s and already generated
         runs;
8      Write the rest element in W_s to the current run;
9      Expand the current run;
10 end
```

at a time for sorting. When the element is exhausted, the next block will be loaded(line 2). Figure 2 illustrates the process of the merge on-the-fly mechanism. During the process, $W_n$, $W_s$, and all the previously generated runs are involved in the merge process (line 7). After merging, we will expand the current run. Because of the existence of natural run, we will use one block of DRAM memory as natural run work space, which leads to the shorter length of the generated run, while the run expansion will reduce this effect and avoid generating too many runs.

(3) *Merging Runs*. In the run merge phase, we will exploit the byte addressable feature of the NVM to merge the runs. Unlike MONTRES and the traditional external sorting algorithm, which require loading a block of data into memory each time, we can load only one value from each run.

We assume that there are *k* generated runs. Due to the existence of natural run, in the merge phase, we use a min-heap containing each run's minimum element with size of *k*+1.

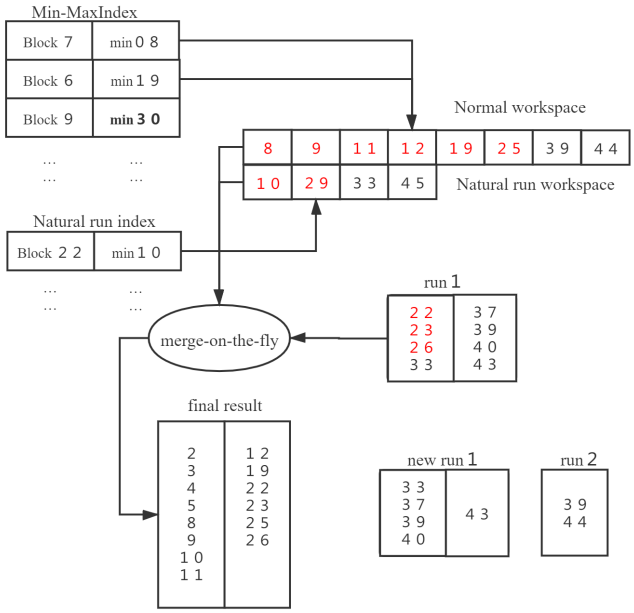


Fig. 2. Example of the merge-on-the-fly mechanism

TABLE II
SYMBOLS USED IN COST ANALYSIS

| Symbol | Definition |
|---|---|
| $N$ | Number of the elements to be sorted |
| $N_n$ | Size of the natural run |
| $N_{nl}$ | Number of the elements that are loaded into the work space from natural run during run generation phase |
| $N_s$ | Number of the elements that are directly written into the final result |
| $r$ | Cost of one read operation |
| $w$ | Cost of one write operation |
| $\lambda$ | $r/s$ |
| $P_r$ | Additional reads during the merge on-the-fly |
| $P_w$ | Additional writes during the merge on-the-fly |

On each iteration, we write the minimum value in the min-heap into the final result and delete it. Then, we extract the next value from the run containing the lowest value. When a run is exhausted, we will reduce the size of the min-heap by 1 and continue the merge until all the runs are exhausted.

### D. Cost Analysis

Table II gives the symbols used in cost analysis. In the run generation phase, first we need scan the input data to build the MinMax-Index, so the read/write cost of the scanning process is: $N \cdot r$.

After the scanning, we begin to generate runs. All data that is not in the natural run will be loaded into DRAM memory, sorted, and written back to NVM. Some of the data will be written back directly to the result. We assume that the total amount of this part of data is $N_n$. The read and write cost of this section is $(N - N_n) \cdot (r + w)$. While a portion of the data in the natural run is loaded into DRAM memory, we assume that the total amount of data loaded into DRAM memory at this stage is $N_{nl}$, resulting in a read-write cost of $N_{nl} \cdot r$. Due to the existence of merge on-the-fly mechanism, during the run generation phase, we will also read the runs that have been written back before and the qualified data will be written back to the final result. The data that participates in the merge process in the natural run and the qualified data will also be written back to NVM. We assume that the additional reading operation generated in this process is $P_r$, resulting in an additional write operation of $P_w$. For the convenience of later calculation, $P_w$ is divided into two parts: The first part is created by writing back natural run data, set to $P_{w1}$; the second part is created by writing back other data, set to $P_{w2}$, $P_w = P_{w1} + P_{w2}$.The resulting read and write cost is $(P_w \cdot w + P_r \cdot r)$. To sum up, we can get the total cost of reading and writing in the run generation phase by Eq.1:

$$\begin{aligned} C_{rg} &= N \cdot r + (N - N_n) \cdot (r + w) + N_{nl} \cdot r \\ &\quad + P_w \cdot w + P_r \cdot r \\ &= (2 \cdot N + N_{nl} + P_r - N_n) \cdot r + (N + P_w - N_n) \cdot w \\ &= ((2 + \lambda) \cdot N + N_{nl} + \lambda \cdot P_w + P_r - (1 + \lambda) \cdot N_n) \\ &\quad \cdot r \end{aligned} \tag{1}$$

During the run merge phase, all the data not loaded in the natural run will be loaded into DRAM for merging and then written back to NVM. The resulting read-write cost is $(N_n - N_{nl}) \cdot (r + w)$. The elements in the natural run that have been loaded into DRAM memory but have not been written back in the process of run generation also need to be written back to NVM. The total number of data in this part is $N_{nl} - P_{w1}$. Thus, the read and write cost of this part is $(N_{nl} - P_{w1}) \cdot w$. At the same time, the remaining elements in the generated runs will also be loaded into DRAM memory for merging and then written back to NVM. The total number of remaining elements is $N - N_n - N_s - P_{w2}$. Therefore, the read-write cost of this part is $(N - N_n - N_s - P_{w2}) \cdot (r + w)$. To sum up, we can get the total read-write cost in the run merge phase by Eq. 2.

$$\begin{aligned} C_{rm} &= (N_n - N_{nl}) \cdot (r + w) + (N_{nl} - P_{w1}) \cdot w \\ &\quad + (N - N_n - N_s - P_{w2}) \cdot (r + w) \\ &= (N - N_{n1} - N_s - P_{w2}) \cdot r \\ &\quad + (N - P_{w1} - N_s - P_{w2}) \cdot w \\ &= ((1 + \lambda) \cdot N - N_{n1} - (1 + \lambda) \cdot N_s - P_{w2} - \lambda \cdot P_w) \\ &\quad \cdot r \end{aligned} \tag{2}$$

Based on Eq. 1 and Eq. 2, we can get the total cost of the algorithm by Eq.3. In Eq. 3, $P_{w2}$ approximately equals $P_r$. Eq. 3 shows that the cost of NVMSorting is associated with the size of natural run and the data that can be written into the result directly.

$$\begin{aligned} C_m &= C_{rg} + C_{rm} \\ &= ((3 + 2 \cdot \lambda) \cdot N - (1 + \lambda) \cdot (N_n + N_s) - P_{w2} + P_r) \\ &\quad \cdot r \end{aligned} \tag{3}$$

For the traditional external merge sort algorithm, it is easy to get the total read-write cost by Eq.4.

$$\begin{aligned} C_e &= 2 \cdot N \cdot (r + w) \\ &= (2 + 2 \cdot \lambda) \cdot N \cdot r \end{aligned} \quad (4)$$

Therefore, we can get the cost reduction of our NVMSorting compared with traditional external merge sort by Eq.5.

$$\begin{aligned} C_{dec} &= C_e - C_m \\ &= ((1 + \lambda) \cdot (N_n + N_s) + P_{w2} - P_r - N) \cdot r \end{aligned} \quad (5)$$

Eq. 5 shows that NVMSorting has lower cost than the external sort when the value of $N_n$ and $N_s$ is larger, which is consistent with the experimental results that will be discussed in Section IV.

## IV. Performance Evaluation

In this section, we report the experimental results of NVM-Sorting. As sorting algorithms are fundamental in computer science and there are a number of existing sorting algorithms, we will compare NVMSorting with several representatives of sorting algorithms. Below, we first introduce the experimental settings in Section IV-A, then we present the results in Section IV-B.

### A. Settings

So far, there is one industrial NVM module supplied by Intel in 2019, which is called the Intel Optane DC Persistent Memory [10]. However, in this paper, we still use a simulation way to simulate the hybrid memory using DRAM. There are two reasons for the simulation. First, it is hard to use the Intel DC Persistent Memory to construct various kinds of hybrid memory architecture. Thus, we will not be able to conduct experiments on different configurations of DRAM and NVM. Although it is possible to build multiple servers with different NVM and DRAM capacities, it is too costly because of the high price of the Intel Optane DC Persistent Memory . Second, we mainly focus on the count of NVM writing operations in the experiments. Such a metric can be measured correctly in the simulation environment. In other words, the count of NVM writes of a sorting algorithm will not be impacted by the underlying hardware.

In order to simulate the hybrid memory, we use the same method as [4] to simulate NVM. In particular, We insert delays after cacheline reads and writes. In the experiment, we add 20ns latency for a cacheline read operation and 500ns for a cacheline write to simulate NVM reads and writes.

All algorithms are run on a PC with an Intel CPU i5 8265U@1.6GHz. The CPU has 6MB of L3 cache associated with 12-way groups, and each core has 256KB of L2 cache associated with 4-way groups. The cacheline size is 64 bytes. The memory device is 4 GB LPDDR Sumsung memory. We use C++ on Ubuntu 18.04 to implement all sorting algorithms.

The Intel Optane DC Persistent Memory of 512GB costs over ten thousand U.S. dollars.

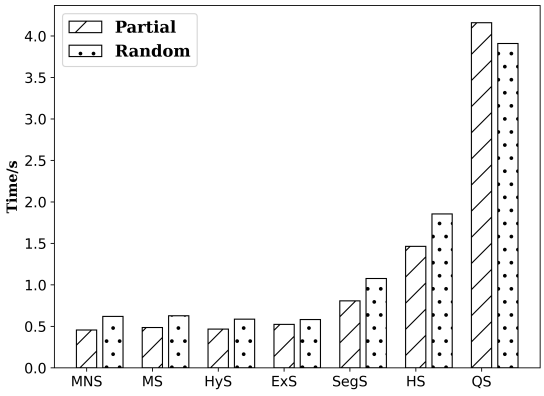


Fig. 3. Time performance (MNS is our NVMSorting algorithm, the others are MONTRES (MS), Hybrid Sort (HyS), External Sort (ExS), Segment Sort (SegS), Heap Sort (HS) and Quick Sort (QS)).

TABLE III
Time-performance improvement of NVMSorting

| | MS | HyS | ExS | SegS | HS | QS |
|---|---|---|---|---|---|---|
| Random | 1.1% | N/A | N/A | 42.4% | 66.6% | 84.1% |
| Partially sorted | 6.2% | 2.5% | 13.0% | 43.5% | 68.9% | 89.1% |

The source code was compiled using g++ version 8.3.0 with the -O3 optimization.

We use two datasets, including an unsorted random dataset (denoted as *random* in the results) and a partially sorted dataset (denoted as *partial*). In general, we expect that our NVMSorting algorithm will perform better on the *partial* dataset than on the *random* dataset.

The performance metrics include execution time *t* and the number of NVM writes *w*. Also, We compare NVMSorting it with six existing sorting algorithms, including quick sort, heap sort, external merge sort, hybrid sort [4], segment sort [4], and MONTRES [8]. And we focus on the comparison of execution time and the number of NVM writes.

### B. Results

In the experiment, we set the available DRAM memory size to 10% of the total size of the data, and six sorting algorithms are compared with NVMSorting.

Figure 3 shows the execution time of all sorting algorithms , where MNS, MS, HyS, ExS, SegS, HS, and QS represent NVMSorting, MONTRES, Hybrid sort, External sort, Segment sort, Heap sort, and Quick sort, respectively. We can see that the time performance of NVMSorting is far better than that of SegS, HS and QS in both case. Table III shows the time performance improvement ratio of the NVMSorting algorithm compared with other algorithms. It can be seen that in the case of completely random data, our algorithm has little improvement in time performance compared with MONTRES, even weaker than Hybrid sort and External sort. While in the case of partially sorted data, the time performance

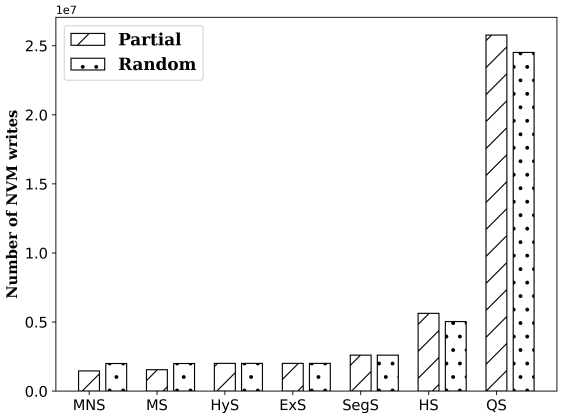


Fig. 4. NVM writes (MNS is our NVMSorting algorithm)

TABLE IV
REDUCTION OF NVM WRITES OF NVMSORTING

| | MS | HyS | ExS | SegS | HS | QS |
|---|---|---|---|---|---|---|
| Random | 0.1% | 0.3% | 0.4% | 23.4% | 60.4% | 91.9% |
| Partially sorted | 5.7% | 27.1% | 27.1% | 43.9% | 74.1% | 94.3% |

improvement of the NVMSorting algorithm compared with the MONTRES and the External sort is relatively obvious, with the improvement ratio reaching 6.2% and 13.0% respectively.

Figure 3 shows the number of NVM writes for all sorting algorithms. We can see that NVMSorting has the least number of NVM writes. Table IV shows the reduction ratio of NVM writes of the NVMSorting algorithm compared with other algorithms. As shown in Table IV, for the completely random data, NVMSorting has slight improvement in terms of NVM writes, but when running on the partially sorted data, NVM-Sorting has reduced 27.1% more NVM writes than Hybrid sort and External sort, and 5.7% more NVM writes than MONTRES. This indicates that NVMSorting is particularly suitable for partially sorted data.

We can see from the experimental results that when running on the partially sorted dataset, NVMSorting achieves significant improvement over other sorting algorithms in terms of time performance and NVM writes. When the dataset is completely randomly unsorted, NVMSorting has comparable performance with MONTRES, hybrid sort, and external sort. This implies that NVMSorting is more efficient for partially sorted datasets. For partially sorted datasets, there is high probability of the occurrence of natural run, meaning that $N_n$ in Eq. 3 is large. At the same time, there are a large proportion of elements in one data block that are less than the minimum value in the next data block. Thus, there will be more elements that can be written into the final result directly, i.e., $N_s$ in Eq. 3 is large. When the data is completely random, the probability of the occurrence of natural run becomes low, and the merge-on-the-fly mechanism does not work effectively, resulting in little performance improvement. To sum up, NVMSorting is more suitable for partially sorted datasets.

## V. CONCLUSIONS

In this paper, we studied the optimization of sorting algorithms for NVM-based hybrid memory architecture and presented a new NVM-friendly sorting algorithm called NVM-Sorting. NVMSorting is motivated by the MONTRES algorithm that was designed for flash memory. Differing from the original MONTRES algorithm, NVMSorting proposed a new technique called natural run. We developed efficient algorithms for detecting the natural runs in a dataset and sorting data items according to natural runs. We theoretically analzyed the sorting cost of NVMSorting and demonstrated its superiority over the external sorting algorithm. Finally, we verified the performance of NVMSorting on two kinds of datasets and compared NVMSorting to six existing sorting algorithms. The experimental results showed that NVMSorting had higher time performance and fewer NVM writes than MONTRES and other sorting algorithms. In particular, it achieved better performance when running on the partially sorted dataset than on the randomly unsorted dataset.

In the future, we will integrate NVMSorting into database join algorithms [11] to develop efficient sort-join algorithms for NVM-based DBMSs.

## ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation of China (No. 62072419). Peiquan Jin is the corresponding author.

## REFERENCES

[1] Z. Wu, P. Jin, C. Yang, and L. Yue, "APP-LRU: A new page replacement method for pcm/dram-based hybrid memory systems," in *Proc. of NPC*, 2014, pp. 84–95.

[2] K. Chen, P. Jin, and L. Yue, "A novel page replacement algorithm for the hybrid memory architecture involving PCM and DRAM," in *Proc. of NPC*, 2014.

[3] R. Liu, P. Jin, Z. Wu, X. Wang, S. Wan, and B. Hua, "Efficient wear leveling for pcm/dram-based hybrid memory," in *Proc. of HPCC*, 2019, pp. 1979–1986.

[4] S. D. Viglas, "Write-limited sorts and joins for persistent memory," *Proceedings of the VLDB Endowment*, vol. 7, no. 5, pp. 413–424, 2014.

[5] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, and J. Shun, "Sorting with asymmetric read and write costs," in *Proc. of SPAA*, 2015, pp. 1–12.

[6] Y.-P. Liang, T.-Y. Chen, Y.-H. Chang, S.-H. Chen, H.-W. Wei, and W.-K. Shih, "B*-sort: Enabling write-once sorting for nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4549–4562, 2020.

[7] Y. Luo, Z. Chu, P. Jin, and S. Wan, "Efficient sorting and join on nvm-based hybrid memory," in *Proc. of ICA3PP*, 2020, pp. 15–30.

[8] A. Laga, J. Boukhobza, F. Singhoff, and M. Koskas, "Montres: merge on-the-run external sorting algorithm for large data volumes on ssd based storage systems," *IEEE Transactions on Computers*, vol. 66, no. 10, pp. 1689–1702, 2017.

[9] M. B. A. Khernache, A. Laga, and J. Boukhobza, "Montres-nvm: An external sorting algorithm for hybrid memory," in *Proc. of NVMSA*. IEEE, 2018, pp. 49–54.

[10] H. Bu, M. Dong, J. Yi, B. Zang, and H. Chen, "Revisiting persistent indexing structures on intel optane DC persistent memory," *Journal of Computer Science and Technology*, vol. 36, no. 1, pp. 140–157, 2021.

[11] L. Yang, P. Jin, and S. Wan, "Bf-join: An efficient hash join algorithm for dram-nvm-based hybrid memory systems," in *Proc. of ISPA*, 2019, pp. 875–882.