



Asynchronous Compaction Acceleration Scheme for Near-data Processing-enabled LSM-tree-based KV Stores

HUI SUN, BENDONG LOU, CHAO ZHAO, and DEYAN KONG, Anhui University, China
 CHAOWEI ZHANG, Yangzhou University, China
 JIANZHONG HUANG, Huazhong University of Science and Technology, China
 YINLIANG YUE, Zhongguancun Laboratory, China
 XIAO QIN, Auburn University, USA

LSM-tree-based key-value stores (KV stores) convert random-write requests to sequence-write ones to achieve high I/O performance. Meanwhile, compaction operations in KV stores update SSTables in forms of reorganizing low-level data components to high-level ones, thereby guaranteeing an orderly data layout in each component. Repeated writes caused by compaction (a.k.a. write amplification) impacts I/O bandwidth and overall system performance. Near-data processing (NDP) is one of the effective approaches to addressing this write-amplification issue. Most NDP-based techniques adopt synchronous parallel schemes to perform a compaction task on both the host and its NDP-enabled device. In synchronous parallel compaction schemes, the execution time of compaction is determined by a subsystem that has lower compaction performance coupled by under-utilized computing resources in a NDP framework. To solve this problem, we propose an asynchronous parallel scheme named *PStore* to improve the compaction performance in KV stores. In *PStore*, we designed a multi-tasks queue and three priority-based scheduling methods. *PStore* elects proper compaction tasks to be offloaded in host- and device-side compaction modules. Our proposed cross-leveled compaction mechanism mitigates write amplification induced by asynchronous compaction. *PStore* featured with the asynchronous compaction mechanism fully utilizes computing resources in both host- and device-side subsystems. Compared with the two popular synchronous compaction modes based on KV stores (TStore and LevelDB), our *PStore* immensely improves the throughput by up to a factor of 14 and 10.52 with an average of a factor of 2.09 and 1.73, respectively.

CCS Concepts: • **Information systems** → **Flash memory**; • **Computer systems organization** → **Embedded software**; **Secondary storage organization**; • **Hardware** → **Emerging architectures**; **External storage**

Additional Key Words and Phrases: Near-data processing, key-value store, asynchronous compaction, write amplification

This work is supported in part by National Natural Science Foundation of China under Grants No. 61702004 and No. 62072001, State Key Laboratory of Computer Architecture (ICT, CAS) under Grant No. CARCH201915, and Natural Science Research Projects at Higher Institutions in Anhui Province (Grant No. KJ2017A015). Xiao Qin's work is supported by the U.S. National Science Foundation under Grants No. IIS-1618669, No. OAC-1642133, and No. CCF-0845257.

Authors' addresses: H. Sun, B. Lou, C. Zhao, D. Kong, and X. Qin, Anhui University, 111 Jiulong Road, Hefei, Anhui, 230601, China; e-mails: {sunhui, lbd, chaozhao, anddeyankong}@ahu.edu.cn, xqin@auburn.edu; C. Zhang, Yangzhou University, 88 South Daxue Road, Yangzhou, Jiangsu, 225127, China; e-mail: cwzhang@yzu.edu.cn; J. Huang, Huazhong University of Science and Technology, 1037 Luoyu Road, Wuhan, Hubei, China; e-mail: hjzh@hust.edu.cn; Y. Yue, Zhongguancun Laboratory, Cuihu North Road 2, Beijing, Beijing, China; e-mail: yylhust@qq.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](https://permissions.acm.org).

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1539-9087/2024/09-ART93 \$15.00

<https://doi.org/10.1145/3626097>

ACM Reference format:

Hui Sun, Bendong Lou, Chao Zhao, Deyan Kong, Chaowei Zhang, Jianzhong Huang, Yinliang Yue, and Xiao Qin. 2024. Asynchronous Compaction Acceleration Scheme for Near-data Processing-enabled LSM-tree-based KV Stores. *ACM Trans. Embedd. Comput. Syst.* 23, 6, Article 93 (September 2024), 33 pages. <https://doi.org/10.1145/3626097>

1 INTRODUCTION

LSM-tree [25] is widely adopted in key-value stores to manage large-scale unstructured data. Converting random write requests to sequential ones achieves improved write performance. A multiple-leveled LSM-tree contains a memory-resident component and multiple disk-level components. More times than not, compaction is deployed in LSM-tree to balance data placement among the components. More specifically, the compaction reorganizes data from lower-level components to higher-level components, followed by updating SSTable to guarantee that data are sequentially stored in each high-level component. LevelDB is a fast key-value storage library [20], which manipulates compaction in two caps—minor compaction and major compaction. A minor compaction usually takes place when data are transferred from main memory to disks. In case data volume in memory reaches a pre-defined threshold, the minor compaction is invoked to write memory data to its adjacent disk component. A major compaction is mainly responsible for moving data among disks. When the amount of data stored in a disk component reaches a given threshold, the major compaction is triggered to write data into a low level of the storage hierarchy. Frequent read/write operations while applying a major compaction may spark write-amplification issues leading to drained storage bandwidth and downgraded system performance.

Most of the existing methods adopt a CPU central scheme on the host side to relieve write amplification and to optimize system performance [41, 44]. These methods, however, merely rely on the host-side CPU and I/Os transferring large amounts of data that affects applications performance and resource utilization. A novel computing paradigm known as **near-data processing (NDP)** or **in-storage computing (ISC)** [7, 8, 11, 16] has been becoming attractive to the academia and industry.

The near-data computing model aims to curtail data movement costs through placing computation to the nearby location where data are stored. The NDP model incorporates data locations and insightful information to be extracted from data. There are growing pieces of research that adopt the NDP model to improve the performance of KV stores [32–34]. These works deploy both host and its NDP-enabled devices to perform compaction in a synchronous manner, thereby alleviating write amplification and improving compaction performance.

A compaction task is divided into multiple phases, including data partitioning, compaction execution, two-sided interaction, and results merging. The host- and device-side compaction managers simultaneously partition SSTable across multiple compaction tasks. Finally, the system is notified when the collaborative tasks are accomplished. The execution time of a compaction task is determined by one side that has a longer time to conduct compaction subtask between host- and device sides. The side having the long compaction execution time is a performance bottleneck of NDP-enabled KV stores. It is challenging for the host and the device to balance workload between the two sides.

In this work, we propose a novel NDP-enabled KV store system named *PStore*, where the host and device asynchronously perform compaction to achieve task parallelism. Minor and major compaction operations are carried out in parallel rather than suspend major compaction. Unlike the existing NDP-based KV stores, *PStore* simultaneously undertakes compaction tasks on the host and its NDP-enabled device. *PStore* with an asynchronous compaction mode eliminates waiting

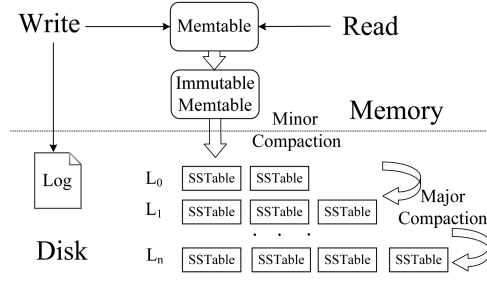


Fig. 1. Basic LSM-tree data structure and compaction.

time on a side that has a shorter execution time spent in the compaction sub-task, thereby boosting compaction performance and optimizing resource utilization. The contributions of this study are summarized as follows.

▷ PStore advocates for multi-task queue and three priority-based scheduling schemes. PStore efficiently offloads compaction tasks to the host and its NDP-enabled device by considering various compaction levels, the number of SSTables and the number of compaction tasks. PStore achieves a high performance of asynchronous compaction; meanwhile, PStore reduces write amplification compared with the synchronous compaction mode.

▷ We bring forth the task-level parallelism in PStore to balance compaction load between a host and its NDP-enabled device by adopting the NDP model to perform asynchronous compaction for key-value stores.

▷ PStore fully utilizes computing resources on both a host and its NDP device. We evaluate the performance of PStore using *DB_Bench* and *YCSB-C*. PStore improves the throughput of the other synchronous NDP-enabled KV stores by a factor of 2.09. The CPU utilization on the device is improved by an average of a factor of 1.65, and the bandwidth utilization rate is increased by a factor of 2.42.

This article is organized as follows. We present background and motivation in Section 2. In Section 3, we describe our proposed overview system of PStore. Section 4 denotes the details of our experimental setup. Then, we discuss the experimental results in Sections 5 and 6. Section 7 summarizes the related work. Finally, We conclude this study with remarks and our future works in Section 8.

2 BACKGROUND AND MOTIVATION

LevelDB is a LSM-tree-based key-value store library developed by Google. LevelDB is widely used in large-scale storage systems like Google's BigTable to achieve high throughput [4]. LevelDB stores MemTable and Immutable MemTable in memory, but **sorted string table files (SSTable)** are stored on the disk.

Figure 1 depicts that key-value items from an application are firstly written into a commit log and buffered in a sorted data structure in main memory called MemTable, which has limited space (e.g., 4 MB). When the size of a MemTable reaches to its predefined capacity, the MemTable is transformed as an immutable MemTable. Then, the transformed MemTable is dumped onto disk as a sorted string table file (i.e., SSTable) by a background thread. On disk, SSTables are organized into a series of levels that grow exponentially in size. A new SSTable is placed in a low level L_0 . Different SSTables may have overlapping ranges at level L_0 , but the overlapping of key ranges of SSTables is not allowed at the other levels. Each level has a limited number of SSTables. To maintain this scheme, a storage system migrates an SSTable from level L_k to L_{k+1} , the process

of which is referred to as *compaction*. A compaction procedure selects an SSTable at a full level and finds SSTables with overlapping ranges of keys at a subsequent level. Next, an n-way merge operation performs to produce new SSTables, which are stashed at a higher level. The previous SSTables, marked obsolete, are deleted from the disk.

In data intensive applications, large-scale data movement increasingly consumes system resources and energy, which leads to a bottleneck for data movement between storage and memory in high performance systems [2]. A traditional storage contains a low-energy ARM processor, memory, and storage mediums (e.g., magnetic media or non-volatile memory), which take various responsibilities for data processing. More specifically, the ARM processor performs address mapping and garbage collection, yet the memory and the storage mediums are deployed to store data. In other words, a storage device is a computer with light-weighted computing resources [39]. To address this shortcoming, **near-data processing (NDP)** is proposed to move data computation at both memory [9, 23] and storage-level [7, 11]. Because computing units usually are directly embedded into storage devices for near data processing, computation tasks are offloaded from host to the devices instead of moving data from the devices to host. In an NDP-based storage architecture, the host-side computing tasks can be offloaded to the NDP-based device, which later retrieves results back from the device.

When it comes to NDP-based KV stores, host- and device-side computing resources are used to perform compaction in parallel, reducing write amplification and boosting system performance. The host- and the device-side subsystem in Co-KV collaboratively conduct compaction tasks. A compaction task is divided into two parts: offloaded to the host side and the device; then, results are returned to the host for updating purpose. The two compaction subtasks are divided in a fixed ratio of 1:1 data volume. In prior work, TStore [32] is featured with a time-aware compaction task offloading scheme to dynamically dispatch compaction tasks to both host and device according to compaction time. In particular, the host- and the device-side subsystems work together to synchronously participate in compaction. Compacted SSTables¹ are partitioned and offloaded to the compaction modules on both sides. When the two parties complete their compaction tasks, integrated feedback is submitted to the host. We conduct experiments to study the compaction time of Co-KV and TStore on both the host and the NDP-enabled device. Please refer to Section 4.1 for the experimental setup and parameter settings. TStore adjusts the compaction task allocation between the host and the device sides according to the perceived compaction time. For example, if the execution time of the device-side TStore is shorter than that of the host side, the host is being overly loaded by processing numerous tasks. In this scenario, TStore cuts back the number of tasks being offloaded to the host in the next cycle, with a hope to achieve a balanced compaction time. The total compaction time is determined by the side on which the compaction time is longer between the host and its device. Figures 2(a) and 2(b) show the compaction time discrepancy between the two laterals when the host and device conduct Co-KV and TStore—two synchronous NDP-enabled KV stores. We reckon that TStore takes the advantage of a dynamic scheme for tasks partitioning and offloading, which synchronizes and parallelizes compaction. Unfortunately, TStore is incapable of narrowing the computing-capacity gap between the host and its device. In the synchronous approach, a compaction task is divided into two subtasks offloaded to the host and the device. The heterogeneous computing resources inevitably cause one side to waste time in awaiting another party.

The compaction time discrepancy between the two parties is represented as $T_{\text{difference}} = |T_{\text{host}} - T_{\text{device}}|$, in which T_{host} and T_{device} denote the compaction time spent on the host and the device, respectively. The execution times of both laterals are unequal when the same amount of

¹SSTables were selected for compaction in the compaction queue but have not yet been executed.

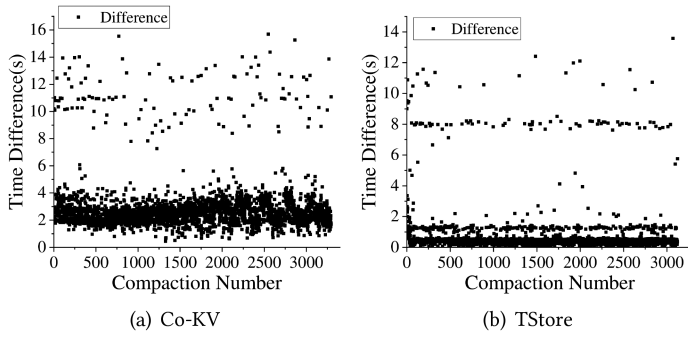


Fig. 2. Execution time difference between host- and device-side subsystems on NDP-enabled KV stores in the synchronization compaction mode.

compaction load is handled. The average execution time difference in Co-KV and TStore is 2,805.4 and 749.6 ms, respectively. Although TStore curtails compaction execution time compared with Co-KV; however, it is arduous to eliminate the time discrepancy in the context of the synchronization method. With a minimum data volume of 10 GB in the experiment, the compaction operation is around 1,000, and the execution time wasted by TStore is approximately 12 min.

To address this problem with the expectation to fully utilize the computational resources, we propose the asynchronous mechanism to reshape the performance of heterogeneous computational resources in the synchronous mode. Without waiting time, the host and the device are slated to continually conduct compaction tasks without a hitch. We compare the synchronous method (TStore) with the asynchronous one. TStore embraces one of the synchronous compaction mode-based KV stores, providing a minor time difference on both sides. In the experiment (see Section 5), we investigate the performance of PStore and TStore under the workloads with a wide ranged data sizes and key sizes. PStore performs better than TStore in terms of write amplification, throughput, CPU utilization, and bandwidth.

3 ASYNCHRONOUS COMPACTION MODE-BASED PSTORE

In this section, we delineate a system framework and the asynchronous mechanism of PStore.

3.1 Overview System of PStore

3.1.1 System Framework. Figure 3 shows the overview system of PStore with host- and device-side subsystems. The host-side subsystem is adapted from LevelDB, and the device-side subsystem enables the NDP-enabled device to conduct compaction.

The host-side system is configured with three modules.

Three-level priority scheduling module. Three-level priority scheduling is implemented by setting `score_trigger`, `score_do`, and double-ended queues. The three-level priority scheduling module prioritizes compaction triggers, the selection of SSTables accompanied with elected compaction tasks (see Section 3.2)

Host-side compaction module. When compaction is invoked on the host, the compaction task is executed on the host side. These keys are from SSD after separating KV to execute the compaction task through the semantic management module, which is capable of speeding up data transfer. The host sorts and merges these keys, followed by writing compaction results back to the device after accomplishing compaction; meanwhile, these SSTables are updated (see Section 3.4).

Semantic management module. When the host performs the compaction task, the host transmits information required by the compaction task to the device through the data reallocation inside

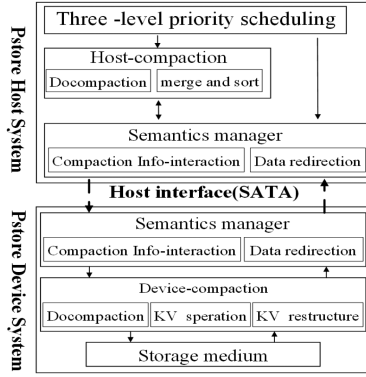


Fig. 3. Overview System of PStore.

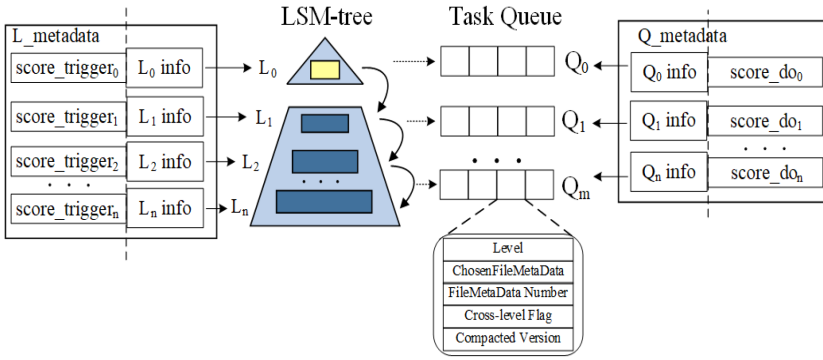


Fig. 4. Asynchronous compaction mechanism of PStore.

the semantic management module. This module running on the device parses file operations from the host and transmits the operations to the host via the transfer information interface.

There are two modules on the NDP-enabled device.

Device-compaction module. The device-side compaction module embraces compaction execution, key-value separation, and key-value reorganization. The key-value separation and key-value reorganization are integral components for performing compaction on the host. When the device executes a compaction task, the task information, such as level, compacted SSTables, key range, is transferred to the device through the semantic management module. Based on this information, the device operates the compaction task and transfers results to the semantic manager. Finally, the host updates the results in parallel.

Semantic management module. Semantic management module on the device side is similar to that on the host, but the information from the device to the host is updated after compaction completes.

3.1.2 Asynchronous Mechanism. PStore aims to orchestrate the asynchronous compaction model and the parallelized compaction model between a host and its device. In Figure 4, each level has a separate queue of compaction tasks. Specifically, the information of compaction tasks embraces the current level, the compacted SSTables (ChosenFileMetaData) metadata, the cross-level flag, and the version information of compaction. Notably, the cross-level flag indicates whether to

undertake cross-level compaction. The version information maintains the metadata of SSTables at each level, and the representative metadata include, but not limit to, file name, file size, minimum and maximum keys in a file, and the status (compaction or delete) of each SSTable. Each SSTable share the same structure catering for metadata, but the metadata size for an SSTable may be different. This nature is justified by the factor that metadata encompass a maximum key, a minimum key, and SSTable size. Values in a key range of an SSTable contain various data, but values are unlikely to be precisely the same for each SSTable. Apart from data, queue Q_i stores its metadata such as the number of elements and length. Compaction task information is kept in each task queue, including levels triggered by the compaction task, SSTable metadata in the task, the amount of SSTable metadata, cross-leveled compaction marks, and the like.

The two sub-phases during the course of executing compaction include (1) task triggering and (2) task execution. The task queue caches compaction tasks activating the host- and device-side compaction modules to asynchronously execute compaction tasks. However, when the compaction task triggering and the compaction execution are asynchronously activated, there is existing a problem as follows. For example, according to the original score rule in LevelDB, when a compaction task is prompted at a level, the compaction task is inserted into the task queue. Then this task is asynchronously executed when a compaction task is triggered at a level. When another compaction task is being performed, the score of this level remains unchanged. However, when the score value is the largest among all the levels, the compaction task is continually selected at this level. The subsequent selection operation tends to cause extra compaction, thereby enlarging write amplification. In addition, this process is likely to block the process of selecting compaction tasks at this level. Thus, the asynchronous mechanism is implemented between the task triggering and task processing phases of executing compaction tasks. PStore defines the *score_trigger* and *score_do*, which is used to asynchronously process trigger compaction and executing compaction.

Compaction time largely depends on the number of compacted SSTables. For instance, a small number of compacted SSTables involve small-size data volume in merge-sort operations, thereby shortening execution time and balancing load. When compaction is embarked upon at a level, any SSTable at the level can be involved in compaction. To reduce compaction time, PStore preferentially designates an SSTable overlapped with a few SSTables at the low level to perform compaction. Because of the diversified computing capabilities between the host and device, PStore offloads compacted SSTables to the high-efficient processing subsystem, enhancing the overall system performance. Thus, an optimized setting of task queue can prompt PStore to assign compaction tasks to an appropriate subsystem.

An asynchronous parallel compaction mechanism devised in PStore includes three types of priorities to speed up the performance of compaction tasks. The priorities are determined by two metrics, namely, *score_trigger* and *score_do*. At levels L_0 and L_i ($i > 0$), the approaches to calculating *score_trigger* differ. At level L_0 , *score_trigger* determines whether the number of SSTables that are not involved in compaction exceeds a preset threshold—the default value of which is 4. At level L_i ($i > 0$), *score_trigger* determines whether the size of data volume of SSTables that do not participate in compaction is larger than the threshold. The metric *score_trigger* dictates the time at which PStore kicks in compaction. The two metrics are expressed as follows:

$$\text{score_trigger} = \begin{cases} \frac{N_{\text{SSTable}} - N_{\text{chosen}}}{N_{\text{threshold}}}, & \text{level } i, i = 0, \\ \frac{S_{\text{SSTable}} - S_{\text{chosen}}}{S_{\text{threshold}}}, & \text{level } i, i > 0, \end{cases} \quad (1)$$

$$\text{score_do} = \begin{cases} \frac{N_{\text{SSTable}}}{N_{\text{threshold}}}, & \text{level } i, i = 0, \\ \frac{S_{\text{SSTable}}}{S_{\text{threshold}}}, & \text{level } i, i > 0, \end{cases} \quad (2)$$

where $N_{SSTable}$ and N_{chosen} represent the number of all SSTables and compacted SSTables at level 0, respectively; $N_{threshold}$ denotes the maximum number of SSTables at level 0 (default 4 in *LevelDB TStore* and *PStore*); $S_{SSTable}$ and S_{chosen} means data volume of all SSTables and data volume of compacted SSTables at level n , respectively; $S_{threshold}$ is the data volume of total SSTables at level n .

At level 0, the *score_trigger* is the ratio of the number of SSTables that are not involved in compaction ($N_{SSTable} - N_{chosen}$) to the threshold, see Equation (1). We advocate for this strategy, because in the asynchronous compaction mode, SSTables are selected for a compaction task (N_{chosen}) that cannot be immediately performed. Therefore, these SSTables are still at the current level without being applied to gauge *score_trigger* measures. Thus, we ought to subtract the compacted SSTables that have not yet been performed. The *score_do* metric determines at which level to carry out local compaction tasks. At level 0, *score_do* represents a ratio of the number of all SSTables to the threshold. At the other levels, we obtain the ratio of the data volume of all SSTables to the threshold. When the ratio is larger than one, the compaction task awaits execution; afterwards, the compaction task will be conducted when the *score_do* value of this level becomes the largest among all the levels.

Notably, the *score_trigger* at level 0 is computed differently from those at the other levels. We present the rationale as follows. The level-0 size is relatively smaller than the sizes of the other levels: using data volume to spark compaction inevitably incur serious write stalls. When level-0 data are compacted to level 1, the process of immutable flush at level 0 will be blocked. It is arguably true that using the number of SSTables instead of data volume in the metric *score_trigger* at level 0 is expected to mitigate write stalls in our proposed PStore.

At level 0, we employed the number of SSTables instead of data volume to obtain the values of *score_trigger* and *score_do*, because the key ranges of SSTables are overlapped at level 0. If there are too many SSTables at level 0, then a large number of SSTables should be scanned as read data—degrading read performance. Thus, it is a necessity to ensure that the data at level 0 must be compacted in time: The amount of data at level 0 is minimized. An immutable flushing to level 0 generates at least one SSTable. When the size of an immutable exceeds the default threshold of one SSTable at level 0, a new SSTable will be generated to save the remaining data. In this situation, SSTable won't be fully written. In the subsequent flushing procedure, a new SSTable is continually generated rather than writing data to the unfilled SSTable. If we calculate *score_trigger* and *score_do* based on the data size, then many SSTables at level 0 may not reach the preset threshold, resulting in an excessive number of SSTables at level 0. Therefore, using the number of SSTables in the assessment accurate and makes sense. At the other levels, there are no overlapped key ranges; therefore, it is unnecessary to scan numerous SSTables when data are retrieved. More importantly, these frequent compaction operations conflict with flush compaction operations, causing write pauses and degrading the system performance. Thus, we use the data volume of SSTables to obtain the *score_trigger* and *score_do* measures at level L_i ($i > 0$). The two-type metric excludes SSTables that have been involved in a compaction task. Unlike *score_trigger*, metric *score_do* contains the SSTables engaged in the compaction task. It is noteworthy that the larger the *score_trigger* and *score_do* present, the higher the priority is given to the level for compaction.

In the following section, we implement the asynchronous compaction mechanism at three-level priorities in Section 3.2. This mechanism migrates SSTables between the host and NDP-enabled device. The amount of migrated data among CPU, memory, and NDP devices spikes in the host-side compaction procedure—a culprit that exacerbates write amplification, blocks user-level I/Os, and impairs the overall system performance. Two optimized methods in PStore are incorporated to reduce the write amplification in the asynchronous mechanism. First, the cross-level compaction mechanism (see Section 3.3) is designed to optimize incoming compaction procedures in advance

by assessing the current compaction situation. In this case, if the `score_trigger` of the next level is greater than one in the current compaction, the two-adjacent-level compaction mode is converted to the three-adjacent-level compaction mode—cross-leveled compaction. This method not only reduces one compaction operation (from level L_{i+1} to L_{i+2} in compaction) but also pushes down write amplification. Second, we designed a key-value separation method in Section 3.4 to suppress the transmitted data volume between the host and device when the host-side compaction is performed. Only keys rather than key-value pairs are transferred between the two parties, where large-scale data transmissions are averted to shorten the host-side compaction time.

3.2 Three Priorities of Compaction Triggering and Execution in PStore

In PStore, we design three types of priorities of compaction triggering and execution to achieve an asynchronous compaction acceleration scheme for NDP-enabled KV stores (see Figure 5). First, the priorities of compaction triggering and compaction execution can be determined through leveraging metrics `score_trigger` and `score_do` (see **Process 1: the priority of compacted trigger**). Second, speaking of the compaction triggering, we set a priority to an SSTable (SSTable S_h) at a higher level according to the number of SSTables that have an overlapping key range with SSTable S_h at a lower level, which optimizes the selection for SSTables in the compaction triggering. Upon the selection of SSTables catering for compaction, the higher-level SSTables that have the smallest number of lower-level overlapping SSTables are given the highest priority. Please refer to **Process 2: the priority of compacted-SSTables selection based on the number of overlapped SSTables**. Third, according to the number of SSTables that are involved in a compaction task, we configure different priorities to the compaction task to optimize the selection for compaction tasks (see **Process 3: the priority of selecting compaction tasks for the host and the device**). In summary, the above three priorities of compaction triggering and execution in above three processes allow the proposed system to make fully use of computational resources on the host and the device, thereby resulting in improved overall performance.

3.2.1 Process 1: The Priority of Compaction Trigger. An application provokes compaction, which is the first-level priority—the priority of the compaction trigger—achieved by using `score_trigger`. PStore nominates the compaction queue with the maximum `score_trigger` value and > 1 according to `score_trigger`: these tasks in the queue ought to be compacted (see ❶ in Figure 5). Unlike LevelDB and TStore, the synchronous compaction mode in PStore does not immediately undertake the compaction task even if the task is invoked. The compaction task is performed as long as it is scheduled in a queue. On the host/device side, a compaction task is put in the queue according to the maximum `score_trigger` at each level; the task information is assigned to a task queue, see Lines 2, 14, and 17 in Algorithm 1. In Figure 5, the maximum value of `score_trigger` determines whether level 1 is compacted at this level. Compared with `score_do` in each queue, the compaction task in a queue that has maximum `score_do` is chosen to be executed (see Lines 2 ~ 8 in Algorithm 2). The multi-queues mechanism employs the values of `score_trigger` and `score_do` to regulate dynamic priorities, which establishes an order of executing compaction at each level. In addition, the host- and device-side compaction modules process compaction tasks, when the number of tasks in the queue is greater than one. PStore deploys an asynchronous mode and compaction tasks in a task queue are independent of each other; therefore, compaction tasks on both sides have no interaction with each other. Then, PStore enables to perform asynchronous compaction tasks in a parallel manner. In case of using multi-task queue, the asynchronous task parallelism has two scenarios: (1) parallel computing among major compaction tasks and (2) parallel computing among major compaction and minor compaction tasks. To further improve compaction performance, we optimize the selections of compacted SSTables and rules of scheduling compaction tasks in queues.

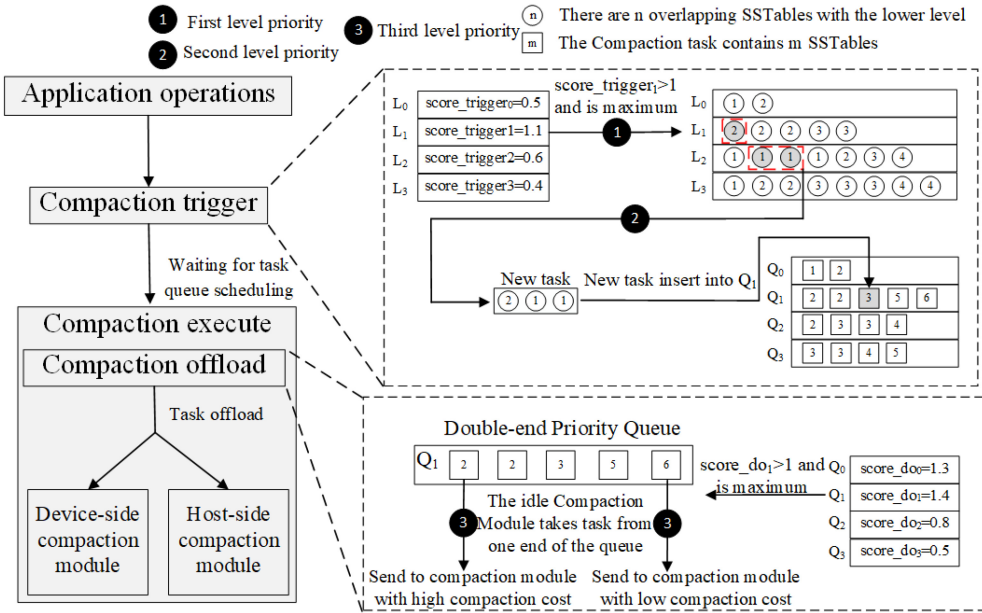


Fig. 5. Multi-task queues and priority scheduling scheme in PStore. This figure presents the structure and procedure in PStore. Three priorities of compaction triggering and execution are designed in three processes. For example, Process 1 selects an SSTable with overlapping key range at level L_1 to involve in compaction tasks and put it into queue Q1, waiting to be scheduled for compaction according to the values of *score_trigger* and *score_do*. For tasks in the task queue, we set the priority of high-level SSTables in accordance with the overlapped number of SSTables at low levels in Process 2. In queue Q1, Process 3 gives priority to compaction tasks for the host and the device from the double-end priority queue. These dark-gray circles indicate compacted SSTables, and these black-solid circles represent three-level priorities.

3.2.2 *Process 2: The Priority of Compacted-SSTables Selection based on the Number of Overlapped SSTables.* In LevelDB, SSTables are involved in compaction in a round-robin manner. When a compaction procedure is invoked at a level, a SSTable is involved in the compaction to reduce the level's score. The round-robin manner enables each key range key to be involved in compactions, but it rarely considers *hot* SSTables and *cold* SSTables. In PStore, we forge metadata information to keep track of the number of overlapped SSTables at a low level (i.e., FileMetaData of each SSTable). The Metadata at each level is ascendingly sorted according to the number of overlapped SSTables across all the levels. Metadata is managed in a form of a linked list offering high-speed data insertion and deletion. When an SSTable is inserted, it is necessary to find a location of the SSTable and insert metadata to keep orderly using the time complexity of $O(n)$. When compaction is prompted at level L_i , metadata at the head of the linked list is retrieved (see Lines 2 ~ 4 in Algorithm 1). If the SSTable corresponding to this metadata has already been elected by another compaction task (compaction_flag of metadata=true), then the next SSTable is checked. Otherwise, the SSTable is chosen, and the loop terminates (see Lines 5 ~ 8 in Algorithm 1). The following step looks for a linked list at the next level L_{i+1} . In the linked list at level L_{i+1} , if the key range of SSTables correlating to metadata overlaps with the SSTable participating in compaction at level L_i , the system determines whether any other compaction tasks have tapped the SSTable. If the answer is a no, then the SSTable is picked up in compaction (see Lines 9 ~ 13 in Algorithm 1). All the elected SSTables are managed in a new compaction task, and the compaction_flag of these SSTables is set

ALGORITHM 1: Compaction task triggering

```

1: compaction_flag : Whether an SSTable is selected to be involved in compaction
2: if score_triggeri > 1 and is the maximum then
3:   while do not get the end of the Li linked list do
4:     Get metadata from the head of Li FileMetaData linked list
5:     if compaction_flag of metadata == true then
6:       Get the next metadata of the Li linked list
7:       continue
8:     end if
9:     Get the metadata in Li+1 that overlaps with the Li metadata key range
10:    if there is compaction_flag == true in the metadata retrieved by Li+1 then
11:      Get the next metadata of the Li linked list
12:      continue
13:    end if
14:    Generate compaction task information
15:    Set metadata compaction_flag=true
16:    Insert into Qi task queue
17:    Update score_trigger
18:  end while
19: end if
20: Waiting for scheduling in the task queue

```

to true. Then, the task is inserted into the queue. Finally, the *score_trigger* is updated (see Lines 14 ~ 18 in Algorithm 1). This process is obliged to detect the overlapped SSTables at low levels while maintaining the order of the linked list. However, the number of SSTables engaged in each compaction can be minimized that immensely shorten compaction time. Especially, compaction manipulates metadata in memory rather than conducting writes/reads or merge-sort operations for SSTables when the number of overlapping SSTables between two levels is zero.

If a compaction task involves a small-sized data, then the amount of data to be transferred is marginal; thus, the merge-sort operations in each compaction task is reduced, thereby shortening the compaction time. Compared to LevelDB, the second-level priority scheme elects the SSTable at level *L_i*, which overlaps the minimum number of SSTables at level *L_{i+1}*. The SSTable having the fewest overlapping ones is selected in the compaction task – and is inserted into the compaction queue. In this case, the number of SSTables in a compaction task is the fewest, with a value of *score_do* being lower than the threshold after a compaction operation, to curtail compacted data volume. Meanwhile, since the number of involved SSTable is small and the SSTable overlapping with ones at the next level is minimal, the size of data to be repeatedly written in compaction is compressed to mitigate high write amplification.

At high levels, we opt for SSTables in accordance to the overlapped number of the SSTables at low levels; thus, PStore assigns the second-level priority to select SSTables that are involved in compaction. In Figure 5, PStore selects the first SSTable with the smallest overlapping-key-range SSTables at the next level in a compaction task. The two overlapping SSTables at the next level will be composed as a new compaction task (see ② in Figure 5). So far, the work done by the first two priorities is over. The compaction task has been generated, the asynchronous first process (compact trigger) has ended, and the compaction task enters the task queue to wait for scheduling.

3.2.3 Process 3: The Priority of Selecting Compaction Tasks for the Host and the Device. The host side and the corresponding NDP-enabled devices are heterogeneous in terms of hardware configuration, read/write performance, transmission paths, and computing capabilities. The two laterals exhibit various compaction cost. In PStore, the host- and device-side records the execution time of compaction operations within a period. We set one period within which five compaction tasks are processed and; nevertheless, the period can be readily reconfigured without a hitch. Then, we define a metric—compaction cost—as the ratio of execution time and the number of compacted SSTables in the period. A high ratio signifies low compaction efficiency, because a compaction task takes more time to process an SSTable. PStore ought to allocate appropriate compaction tasks to each party, aiming to achieve high compaction performance. When all SSTables associated with compaction tasks are selected, the information of corresponding tasks is originated and inserted into the compaction task queue at this level. It is worth noting that the compaction task queue is a double-ended priority queue, in which data items are sorted in an ascending order of the number of compacted SSTables. The heap-sorting method is applied in the queue with a time complexity of $O(n\log n)$.

We adopt compaction cost to gauge the compaction processing capability of the host- and device-side subsystems. The subsystem with low compaction cost manages the compaction tasks with multiple SSTables from the right side of the double-ended queue, and another subsystem handles the compaction tasks from the left side. Such salient design enables the more efficient side to promptly process the compaction tasks on the right side. Meanwhile, we assign a score flag for each side. When the compaction cost of one party is higher than its counterpart, the score flag is advanced by one. The score flag indicates which party should schedule tasks in the subsequent compaction procedure.

The score flag carries out three use-cases.

Case 1. The flag score of the host-side is greater than that of the device.

Case 2. The flag score of the host-side is lower than that of the device.

Case 3. The flag score of the host-side is aligned with that of the device.

In case 1, the scheduling strategy of the default compaction task remains unchanged. In case 2, the device-side compaction module assigns compaction tasks to the left side of the queue and the host side schedules tasks on the right side. In case 3, the scheduling scheme of the default task still remains unchanged. The host and device sides restart recording compaction cost in coming compaction procedure within a period. PStore schedules compaction tasks to left or right of a queue depending on comparison results (see ③ in Figure 5). It must be noted that compaction tasks in PStore are complete and independent of one another. No compaction task is partitioned into two subtasks to be designated to the host and device sides. Nevertheless, the two-side results are combined.

Algorithm 2 sketches the compaction execution correlated to the third priority. If the host-side compaction is being performed, then it jumps to the device side; otherwise, the host-side subsystem processes the compaction task in the task queue in the level with the largest *score_do* and *score_do* > 1 (see Lines 1 ~ 2 in Algorithm 2). Because our compaction queue is double-ended, the host and device sides take the tasks from both ends. The left-end compaction tasks are sent to the side with higher compaction cost. Compaction tasks from the right end are delivered to the end with lower compaction cost. The host retrieves a compaction task from the left end of the queue (default) and immediately executes the task and the *score_do* value is amended after compaction (see Lines 3 ~ 5 in Algorithm 2). Similarly, if the device is idle, the device will process the compaction task in the task queue at the level with the largest *score_do* and *score_do* > 1 (see Lines 7 ~ 8 in Algorithm 2). The device, by default, takes the compaction task from the right side of the compaction queue. After compaction completes, the value of *score_do* is revised (see Lines 9 ~ 12 in Algorithm 2).

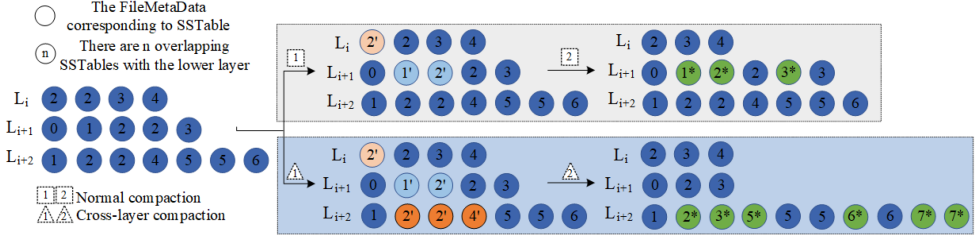


Fig. 6. Cross-level compaction in PStore. This figure presents two types of compaction procedures, i.e., normal compaction (two-adjacent levels) and cross-level compaction. In the normal compaction, symbols $2'$ at level L_i and $1'$ and $2'$ at level L_{i+1} represent the number of SSTables involved in compactions in the two adjacent levels. Symbols 1^* , 2^* , and 3^* are the number of new SSTables after compaction. In the cross-level compaction, symbols, $2'$ at level L_i , $1'$ and $2'$ at level L_{i+1} , and $2'$, $2'$, and $4'$ at level L_{i+2} denote the number of SSTables involved in compaction from three adjacent levels. Symbols 2^* , 3^* , 5^* , and 7^* express the number of new SSTables after the cross-level compaction. Dark-blue circles signify SSTables that are not associated in compaction. The circles with light brown, light blue, and orange colors represent SSTables that are involved in compaction at different levels. In addition, the circles with green color resemble new SSTables after compaction.

ALGORITHM 2: Compaction task execution

```

1: if Host-side compaction module is idle then
2:   if  $score\_do_j > 1$  and is the maximum then
3:     The host takes the task from the left side of  $Q_j$  (default)
4:     Update  $score\_do$ 
5:   end if
6: else
7:   if Device-side compaction module is idle then
8:     if  $score\_do_j > 1$  and is the maximum then
9:       The device takes the task from the right side of  $Q_j$  (default)
10:      Update  $score\_do$ 
11:    end if
12:  end if
13: end if

```

In summary, PStore embraces three-level priorities to perform compaction tasks in an asynchronous and parallelism fashion. Figure 5 depicts the three types of priority schemes in asynchronous compaction. The workflows of the three priorities are implemented according to Algorithms 1 and 2. The host-side of PStore is obliged to handle a complete compaction task in an asynchronous compaction mode. The host in PStore writes more data to the NDP-enabled storage device than prior synchronous compaction-based NDP-enabled KV stores (e.g., TStore). Therefore, the asynchronous compaction mode enlarges write amplification. To address this issue, we design a cross-level compaction mechanism to mitigate write amplification and to improve the compaction performance (see Figure 6).

3.3 Cross-level Compaction Mechanism

The original compaction approach involves the current level L_i and its adjacent level L_{i+1} . The cross-level compaction mechanism is focused on the simultaneous compaction procedures at

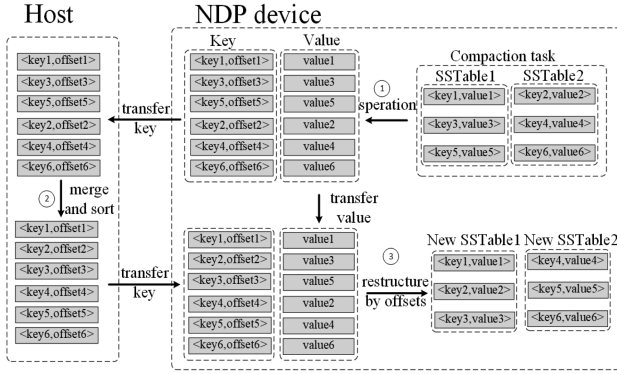


Fig. 7. Host-side compaction with KV separation. A compaction task processed by the host includes two SSTables, each with three KV pairs, and two new SSTables are generated after compaction.

multiple levels L_i , L_{i+1} , and L_{i+2} . In the compaction occurred at level L_i and L_{i+1} , if *score_trigger* of level L_{i+2} is greater than 1, we will need to select overlapped SSTables at level L_{i+2} in compaction. Then, it is prudent to elect SSTables having overlapped key ranges at levels L_{i+2} , L_{i+1} , and L_i in compaction. Subsequently, the new SSTables after compaction are written into level L_{i+2} , thereby avoiding a compaction (from level L_{i+1} to L_{i+2}). The cross-level compaction also slashes the number of rewritten SSTables having overlapped key ranges at levels L_i and L_{i+1} involved in compaction. These overlapping SSTables will be compacted to level L_{i+1} according to the original compaction mode and then they will be compacted to level L_{i+2} when level L_{i+1} triggers compaction. The cross-level compaction mode prevents these overlapping SSTables from being repeatedly written from level L_{i+1} to level L_{i+2} ; as a result, the write amplification is immensely reduced.

Our designed *compaction triggering* mechanism determines to prompt a compaction procedure at level L_i . SSTables are selected to be compacted according to the number of overlapped SSTables between the levels L_{i+1} and L_{i+2} . The new produced SSTables are stored at the level L_{i+1} in the leveled compaction mode, and the tables can be written at level L_{i+2} under the cross-leveled compaction mode. Therefore, the cross-leveled compaction weakens write amplification through write operations. In addition, the cross-leveled compaction mechanism alleviates the unbalanced status of two-leveled SSTables, thereby reducing the number of compaction operations. However, the cross-leveled compaction involves an excessive number of SSTables and enlarges the execution time of compaction. This issue marginally affects the performance of PStore, because both the host and its NDP-enabled device provide powerful computing resources for compaction. When we extend the cross-level compaction to four levels, the number of SSTables participating in one compaction is enormously large, which significantly increases the execution time of a compaction task. It is worth noting that this problem exists in the original cross-level compaction even though it rarely affects the system performance thanks to sufficient computational resources on the host and device sides. If more levels are augmented in compaction, as we expect, then the number of SSTables handled in compaction will exponentially grows—certainly affecting the performance of PStore.

The cross-leveled compaction task tends to be conducted by the PStore's subsystem that has high compaction efficiency. This procedure is completed according to the priority on selecting compaction tasks for both sides to shorten execution time. Meanwhile, the asynchronous compaction mode avoids procedure interference for both sides, thereby cutting back waiting time during compaction.

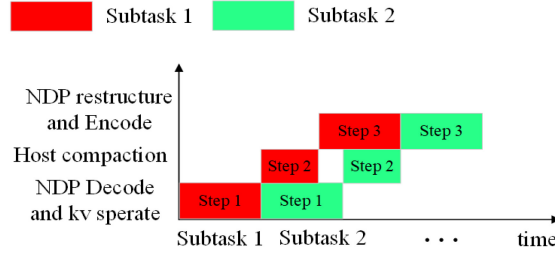


Fig. 8. Practical pipelined compaction on the NDP device.

3.4 Key-value Separation and Pipelined Compaction in the Host

In PStore, we implement an asynchronous execution mode of compaction tasks between the host and the device enabling efficient and sufficient computational resources utilization in NDP-enabled systems. When a task is placed in a queue, the task will be processed by any subsystem (i.e., host or device), which has free computing resources. Each side of PStore does not await compaction tasks comparing with the synchronous-compaction-based NDP-enabled KV stores.

We propose key-value separation and pipelined compaction methods to curb data migrations between the host and its device, thereby alleviating write amplification and improving compaction performance. Deploying the key-value separation concept, PStore separately stores keys and values. The keys of KV pairs are read, sorted, and written to a storage device in compaction tasks. The key size is mostly smaller than 1 KB, whereas the size of a value is usually greater than 1 KB. PStore only involves keys-focused compaction, which fends off read/write operations on values. Therefore, PStore is capable of cutting back data migration among CPU, memory, and storage devices.

More times than not, existing key-value separation solutions separately store keys and values; only keys are processed in compaction (e.g., Wiskey[24], HashKV[3]). Because the values are disorderly stored, garbage collection must be periodically conducted to clean expired values. Meanwhile, the disorderly stored values become a challenge on exploiting spatial localities of data. Thus, a sequential lookup traversal operation is taken over by a random lookup. The existing pipelined compaction contains three steps, namely, reading, sort-merging, and writing. This approach speeds up the execution of compaction, but it impacts the performance of user-level applications by stressing workloads on the host-side CPU.

We implement the key-value separation (see Figure 7) and pipelined compaction (see Figure 8) mechanisms in PStore to suppress the data movement between the host-side memory and its NDP device, which relieves write amplification and improves the host-side compaction. The key-value separation mechanism in Figure 7 is conducted in the host-side compaction, and the whole key-values are stored in the NDP-enabled device. Then, only keys are handled in the host-side compaction, which reduces device-to-host data transfer and host-to-device compaction. Figure 8 depicts the pipelined compaction procedure that entails three steps. As shown in Figure 7, the host interacts with the device, and only keys are conveyed between the host and the device. According to compaction information, the device separates keys and values in SSTables. The key array stores the addresses of values for subsequent reorganization and; then, the key is relayed to the host for sort-merge operations. Afterward, results are shipped back to the device, where the keys are reorganized to new SSTables according to the addresses of values. In Figure 7, a host-side compaction task includes two SSTables. Each of the SSTables has three keys, and then two new SSTables are generated after compaction. Each step is separately realized by a thread, meaning that there are three threads: (1) separation thread on the device, (2) merge-operation thread on the host, and

(3) the key-value restructure thread on the host. Thus, we divide a host-side compaction task into the following three steps.

Step 1. The key-value separation thread in NDP devices retrieves KV pairs according to the data format of an SSTable. From the perspective of a KV pair, the key or value is stored to a respective array in device-side main memory. In the key array, a key item records its corresponding value-item subscript of the value array. In an array, the subscripts-related time complexity on searching a data item is $O(1)$. The value array is transferred to the restructure thread through memory sharing, followed by awaiting sorted key arrays obtained from the host. The key array is transferred to the host via an interface between the host and its NDP-enabled device.

Step 2. The host acquires the key array, sorts key items, and delivers the new sorted key array to the restructure thread on NDP devices. It is noteworthy that all the above steps are orchestrated in one thread.

Step 3. When the restructure thread in an NDP device receives new key arrays, each key finds its corresponding value according to the subscript. Meanwhile, the key reconstructs the new SSTables according to the sequence of the key arrays.

Recognizing that it is time consuming to transfer data between the host and its NDP-enabled device prolongs, we utilize a pipelined mode to shorten waiting time in three steps. More specifically, the waiting time is caused by transferring key arrays between the host and its NDP device. Users determine the number of subtasks and sizes by setting up the number of KV pairs. Thus, the pipelined compaction mechanism accelerates the key-value separation process, thereby vastly optimize the overall system performance.

3.5 Discussion

Our proposed PStore focuses on optimizing the compaction performance in terms of write amplification. The asynchronous compaction mechanism is implemented based on the three-level priority. We discuss the GC overhead in KV separation and crash consistency and tasks of a level in the asynchronous compaction.

GC overhead in KV separation. A traditional key-value separation method stores keys in LSM-tree and registers values in a value-log file. Since the keys and values are separately stored on disks, only keys are processed in compaction. Then, a garbage collection procedure retrieves invalid values in the log file. In PStore, the key-value separation approach curtails the data volume transmitted from the NDP device to its host without tapping additional value logs. The key-value separation is invoked when the host-side compaction is conducted. The LSM-tree still stores common key-value pairs, where the key and value in the data distribution are orderly stored. The device-side subsystem transmits keys instead of key-value pairs to the host, and the key array stores the subscripts of the corresponding values. Upon the arrivals of keys, the host issues merge-sort operations on keys and transmits result backs to the device. Thereafter, the device reorganizes the key-value pairs according to the subscripts of corresponding values. Then, new SSTables—being rebuilt in accordance with the compaction information—are ordered structures in the normal LSM-tree, which avoids unordered data storage and GC problem; meanwhile, transmission data volume is dramatically pruned.

Crash consistency in PStore. Crash consistency is achieved in PStore by the virtue of the write-ahead logging (WAL) mechanism. Upon updating data, PStore records the update operation to a log file—the Write-Ahead Log—and then applies the updated data to an in-memory data structure such as MemTable. When compaction is launched, PStore originates a new SSTable, followed by writing relevant metadata and data-update operations into the log file. Notably, the metadata includes SSTable size, SSTable start and end key ranges, levels whose SSTables were involved in the compaction, and a new level that stores the new SSTables.

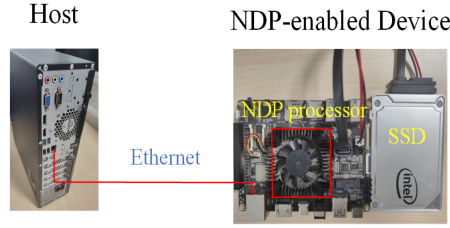


Fig. 9. NDP device-based tested platform.

Once metadata and update operations are registered into the log file and are persistent, PStore merely appends new SSTables generated by compaction into the database whose consistency is maintained. If a crash occurs during compaction, then the operation will be re-played by restoring the log file to restore data consistency.

Tasks of a level in the asynchronous compaction. Notably, in the asynchronous compaction mode, when a compaction task is executed at a level, compaction tasks can also be originated continuously. It is uncommon that all levels have only one compaction task or none. Rather, there are a handful of compaction tasks in the task queue at each level. Even though this case exists, the compaction task in the task queue will be allocated to the host as the task is sitting idle; otherwise, the compaction task is offloaded to the device when the host side is kept busy. Afterwards, we select another task queue with `score_do > 1` and the score is the largest among its peers. In doing so, the host- and device-side computing resources are primed to be fully utilized.

4 EXPERIMENTAL SETUP

4.1 Platform

We implement a prototype platform to emulate a near-data processing environment, where we conduct extensive experiments to validate our proposed PStore method compared against the *TStore* system—an NDP-enabled KV store powered by a scheme of offloading dynamic compaction tasks. Meanwhile, the synchronous method orchestrated by *TStore* is plugged in to vet the performance comparison against PStore—an asynchronous method. LevelDB is a second prime contender evaluated in our empirical study. The host and the device are connected through the Ethernet interface (see Figure 9).

The prototype platform consists of a host subsystem and a device subsystem. The host-side subsystem is deployed on a computer that contains a Pentium E6700 CPU and 4 GB memory. The operating system of the host subsystem is Ubuntu 16.04. The device-side subsystem is realized by an ARM development board [1], which is comprised of six cores (two Cortex-A72 and four Cortex-A53) and 2 GB main memory. We apply the 256 GB Intel 545s SATA SSD on the development board as a storage medium. The operating system running on the device subsystem is a lightweight embedded Ubuntu system. The power consumption of the device is 5 ~ 6 Watt. Table 1 lists details of our NDP prototype.

4.2 Workloads

PStore processes compaction in an asynchronous parallelism manner. Since random-write patterns mostly spark compaction in KV stores, our experimental study is mainly focused on random-write patterns found in a wide range of applications.

We deploy cross-leveled compaction and host-side pipelined compaction to optimize asynchronous parallelism in compaction. We expand the proposed PStore into four variants, including PStore-CP, PStore-TP, PStore-C, and PStore-P, where *C* represents the cross-leveled compaction,

Table 1. PStore Evaluation Platform

Hardware Parameter	Value
Host CPU	Pentium Dual-Core E6700 @3.2 GHz x2
Host DRAM	4 GB DDR3
Protocol	PStore protocol based on TCP/IP
NDP Processors	Two Cortex A72 and Four Cortex A53
NDP DRAM	2 GB DDR3
NDP Storage Medium	256 GB Intel 545S SSD 550 MB/s (Read) and 500 MB/s (Write)
Host-and-Device Interface	Gigabit Ethernet

Table 2. Workload Characteristics

Basic workload in DB_Bench				
Type	Fillrandom	Fillseq	Value Size (1 KB)	Data Size (10 GB)
db_bench_1	✓		1×, 4×, 16×, 64×	1×, 2×, 3×, 4×, 5×
db_bench_2		✓	1×, 4×, 16×, 64×	1×, 2×, 3×, 4×, 5×
Basic workload in YCSB-C				
Write(%)	Data Size		Record Size (1 KB)	Distribution
	Load (10 GB)	Run (5 GB)		
100%	1×	2×	1×, 4×, 16×, 64×	Zipfian
	2×	1×, 2×, 3×, 4×	1×	
90%, 70%, 40%, 10%	2×	2×	1×	

T denotes the transitional leveled compaction, and P indicates the pipeline compaction in the host. All the configurations (e.g., 4 MB Memtable, 2 MB SSTable, 4 KB data block, and 16 B key) among the four variants are identical to make fair comparisons.

We utilize DB_Bench and YCSB-C to generate micro-benchmarks and macro-benchmarks to gauge performance metrics—write amplification, throughput, bandwidth utilization, and CPU utilization. Characteristics of workloads are listed in Table 2. We vary data volume to validate the performance of the tested systems, where the data size is registered in YCSB-C. The requested I/Os obeys the Zipfian distribution.

The workloads db_bench_1 and db_bench_2 are extended from db_bench to capture wide ranged data access patterns. We configured db_bench_1 with value sizes (1, 4, 16, and 64 KB) and data sizes (10, 20, 30, 40, and 50 GB) in a random-write mode. db_bench_1 is configured in a sequential-write mode with the same value and data sizes.

5 EXPERIMENTAL RESULTS

5.1 Results Under the DB_Bench Workload

We compare the performance of PStore-CP, PStore-TP, TStore, and LevelDB in the perspectives of write amplification, throughput, bandwidth utilization, and CPU utilization under the DB_Bench workload (see Table 2). Meanwhile, the workload conditions are configured with various data volume and the value size of KV pairs.

The results plotted in Figures 10 and 11 demonstrate that PStore-CP and PStore-TP share similar write amplification performance (e.g., around 2). We observe that the write amplification is primarily incurred by write-ahead logs; the host-side compaction causes no excessive write amplification. PStore-CP and PStore-TP merely read and sort keys by applying the host-side pipelined

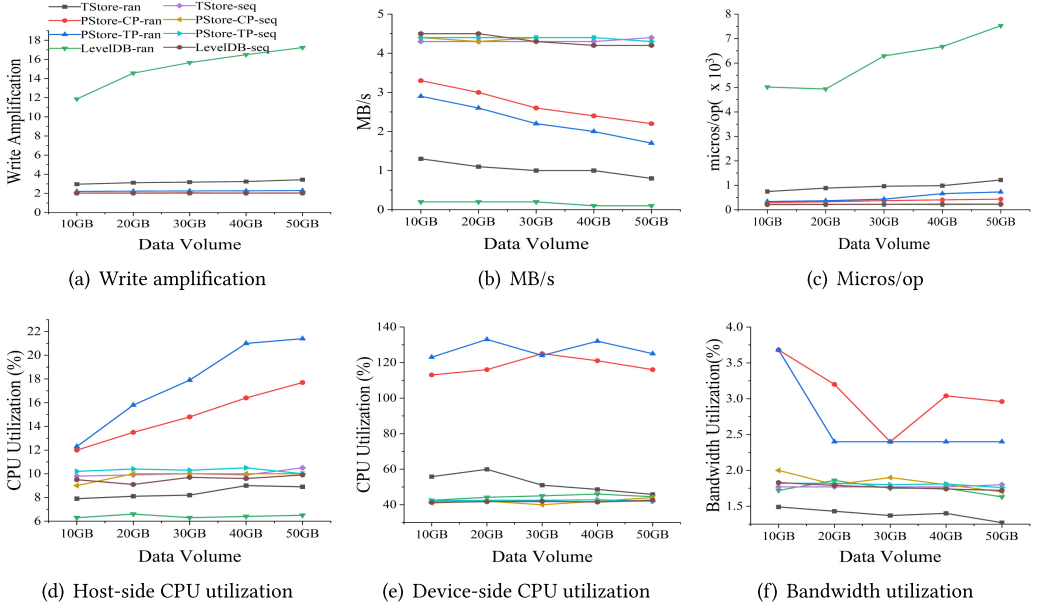


Fig. 10. Results of PStore-CP, PStore-TP, TStore, and LevelDB under the Fillrandom- and Fillseq-based DB_Bench various data volume.

compaction. The device matches and rewrites the values in accordance with the sorted key, thereby reading, sorting, and writing only keys on the host. With an increasing data volume and value size, this approach alleviates an increasing write amplification as data volume rises.

In spite of applying different value sizes in the experiments, PStore-CP delivers similar write amplification as that of PStore-TP (i.e., two). Because write amplification depends on the amount of data written from the host to its NDP-enabled device. The host-side TStore processes 10% of compaction data, but the amount of processed data in the host slowly goes up. In LevelDB, the compaction tasks are all performed by the host and its has the largest write amplification. PStore-CP and PStore-TP compact keys in the host to marginally pushes up the write amplification.

The results unveil that the cross-leveled compaction mechanism lightly impact write amplification in PStore-CP and PStore-TP. The write amplification can be improved when we apply the cross-leveled compaction but the pipelined compaction in PStore. We design experiments to explore this phenomenon. We made use of a line graph to present the performance trend of PStore as a function of the size of the value and key.

Throughput. As the volume of data is expanded, the throughput of PStore-CP, PStore-TP, TStore, and LevelDB is decreased accordingly. However, the metrics are lifted up along with the increased value size. The intensive compaction operations bring forth different throughput (see Figures 10(b) and 11(b)), because the large data volume tends to cause the decrements of KV pairs. High throughput means less time spent in each operation. The throughput of PStores (PStore-CP and PStore-TP) is higher than those of both TStore and LevelDB. Under the 20 GB and 1 KB workloads, the performance of PStore-CP and PStore-TP is 3 and 2.6 MB/s, respectively. The throughput of PStore-CP is 15.4% higher than that of PStore-TP. The performance of TStore is 1.3 MB/s, and its throughput is 2.31 $\times$ and 2 $\times$ larger than those of PStore-CP and PStore-TP, respectively. When the data volume is 40 GB, the throughput of PStore-CP and PStore-TP reaches up to 24 $\times$ and 20 $\times$ that of LevelDB, respectively. The synchronous parallel mode applies the asynchronous parallel

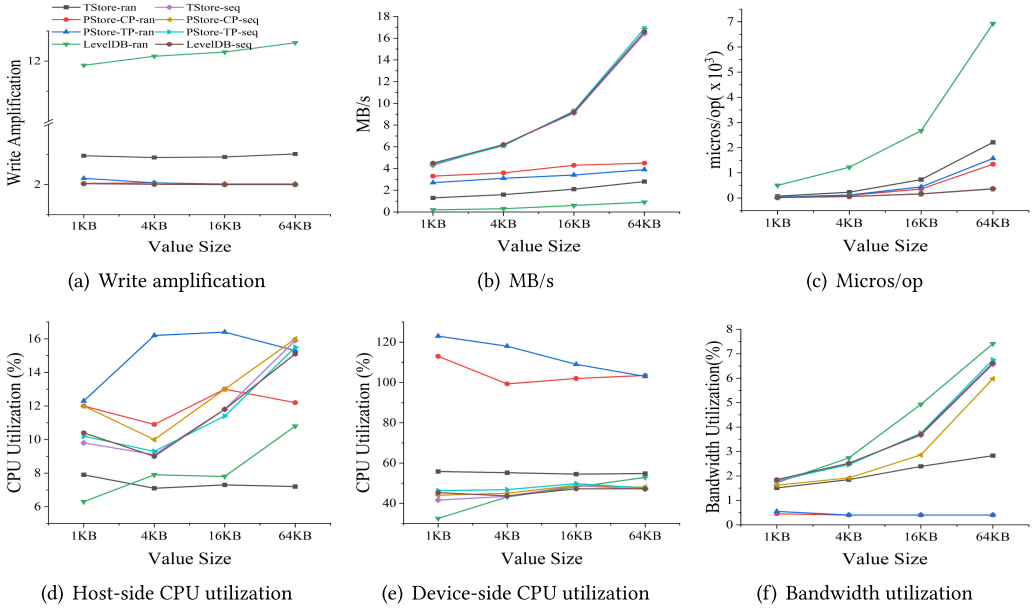


Fig. 11. Results of PStore-CP, PStore-TP, TStore, and LevelDB under the Fillrandom- and Fillseq-based DB_Bench various value size.

compaction cutting down task partitioning and merging. Thanks to no waiting time between the host and the device, the performance of compaction is profoundly improved. PStore-CP increases the throughput of TStore and LevelDB by an averages of 1.73 $\times$ and 10.52 $\times$, respectively.

CPU Utilization. Figures 10(d), 10(e), 11(d), and 11(e) reveal that data volume and value size impact CPU utilization on the host and its device. The CPU utilization of PStore-CP and PStore-TP are too close to call; PStore-CP and PStore-TP's CPU utilization is higher than that of TStore on both host and device. It is noteworthy that the ratio of CPU utilization between PStore-CP and TStore reaches to a maximum level under the 40 GB workload or 1 KB workload. More specifically, TStore's CPU utilization on the host- and device-side are 9% and 48.6%, respectively. The *score_trigger* and *score_do* metrics are measured as 16.4% and 121% in PStore-CP, but these metrics are 21% and 132% in PStore-TP. Compared with TStore, PStore-CP's *score_trigger* and *score_do* surge by 77% and 171%, and PStore-TP's CPU utilization increases by 133% and 171%. However, PStore-CP improves the CPU utilization of TStore by an average of 74.6% and 131% on the host and device. Because compaction tasks in LevelDB are barely performed on the host, the data frequently transfer between the host and the device. Therefore, PStores-CP and PStore-TP have an edge over LevelDB in terms of CPU. For example, the CPU utilization of PStore-CP is 1.51 $\times$ and 2.656 $\times$ of LevelDB on average on the host and device, respectively.

Bandwidth Utilization. Figures 10(f) and 11(f) unveil that PStores-CP and PStore-TP share similar bandwidth utilization. A large amount of data movement between the host and the device consumes high bandwidth. With an increasing data volume, PStore-CP, and PStore-TP present higher bandwidth utilization than those of both TStore and LevelDB. PStore-CP's bandwidth utilization is higher than that of PStore-TP by an average of 16.4%. The results confirm that the bandwidth utilization can be improve by applying the cross-leveled compaction mechanism. PStore-TP increases TStore and LevelDB's bandwidth utilization by an average of 124% and 68.4%, respectively. When the data volume is fixed, the bandwidth utilization of PStore-CP and PStore-TP is

marginally changed with an increasing value size. In contrast, TStore and LevelDB's the bandwidth utilization is sensitive to value size. This trend is reasonable, because only keys are transferred in PStore-CP and PStore-TP, entailing that a constant amount of data cannot push up the number of keys. Thus, the bandwidth utilization of PStore-CP and PStore-TP is lightly altered even if the value size in workloads varies. However, both keys and values are obliged to be transferred in TStore and LevelDB. Thus, the bandwidth utilization of TStore and LevelDB tend to be enhanced by an increased amount of data transmission caused by a big value size.

Sequential Workloads. The performance of the two PStore extensions under sequential workloads is aligned with those of TStore and LevelDB, because write operations of sequential workloads lead to minor compaction rather than major compaction. TStore and PStore are capable of optimizing major compaction. Although PStore revamp minor and major compaction operations in a parallel processing manner, the minor compaction in PStores is almost identical to those in TStore and LevelDB. Thus, PStore-CP and PStore-TP have little effect on write performance.

5.2 Experiments under YCSB-C

This section sheds bright light on performance validations for PStore-CP, PStore-TP, TStore, and LevelDB in perspectives of write amplification, write latency, CPU utilization, and bandwidth utilization under YCSB-C with different data volume and record size.

Data Volume. In this test, we configure a *Zipfian* distribution for the requests in YCSB-C. Figure 12 unravels that the size of a dataset influences performance in terms of write amplification, write latency, CPU utilization, and bandwidth utilization. The performance of PStore-CP is maximum when the workload is 15 GB, but the host's CPU utilization is sitting at the low end. PStore-CP enhances PStore-TP's throughput by up to 15.5% with an average of 7%. The large amount of data worsens the write amplification in TStore and LevelDB. Write amplifications in TStore and LevelDB reach the maximum (i.e., 2.71 and 9.39) under the 20 GB workload. Conversely, write amplifications in PStore-CP and PStore-TP remains at 2, which is independent of the increasing data volume.

Under the 15 GB workload, the write latency of PStore-CP, PStore-TP, TStore, and LevelDB is 306.3, 353.7, 741.77, and 4,059.5 μ s, respectively. PStore-CP cuts down the write latency of TStore and LevelDB by an average of 58% and 75%, respectively. The number of compaction operations is increased along with a growing data volume, which costs bandwidth and downgrades performance. LevelDB has lower CPU utilization than those of PStore-CP and PStore-TP under the workloads with various data volume. Although compaction tasks are undertaken on the host, data movement between the host and device undoubtedly reduce CPU utilization. In LevelDB, the device is dedicated to storing data rather than conducting compaction. Thus, the CPU utilization of the device is lower than that of PStore and TStore.

Both PStore-CP and PStore-TP have the similar CPU utilization on both host and device, even though DB_Bench has different amount of data. The results indicate that the cross-leveled compaction impacts CPU utilization a bit on both sides. Compared with TStore and LevelDB, PStore-CP and PStore-TP's CPU utilization on the host are increased by a factor of 1.87 and 2.62 on average, respectively. However, the CPU utilization of the device is improved by a factor of 1.30 and 2.00 on average. In addition, the bandwidth utilization is increased by a factor of 1.67 and 1.1 on average, respectively.

Record Size. In this study, we apply the 10 GB dataset and the requests meet the *Zipf* distribution (see Figure 13). The large-size record aggregates compaction overhead while increasing throughput when data volume is fixed. The amount of data processed on the host grows when KV pairs are large, thereby worsening the write amplification in TStore and LevelDB. The write amplifications in TStore are maintained at a minimum level of 2.61 and 7.85 under the 1 KB

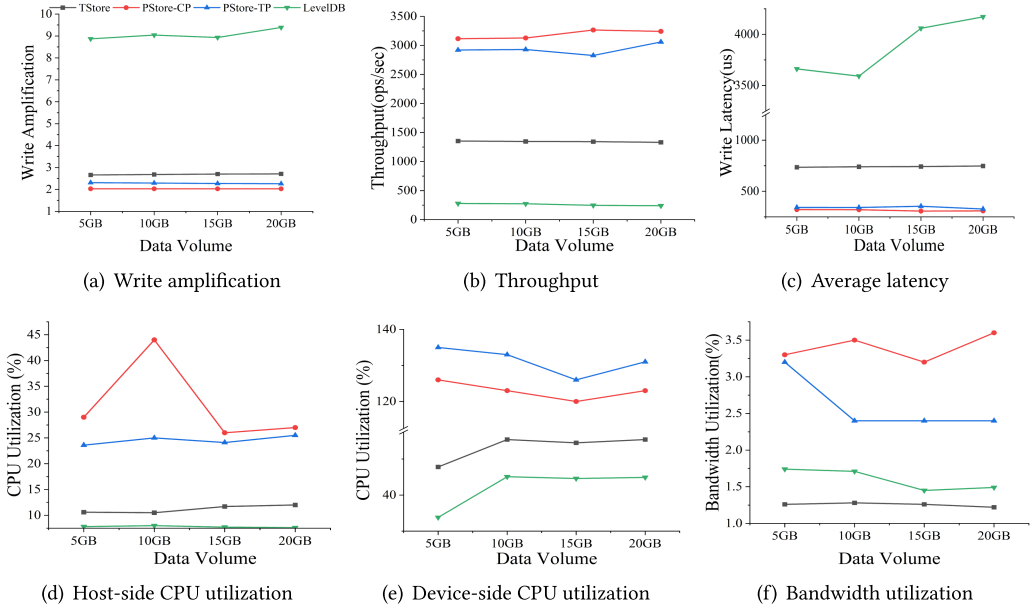


Fig. 12. Results of PStore, TStore, and LevelDB under Zipfian based on the YCSB-C with various data volume.

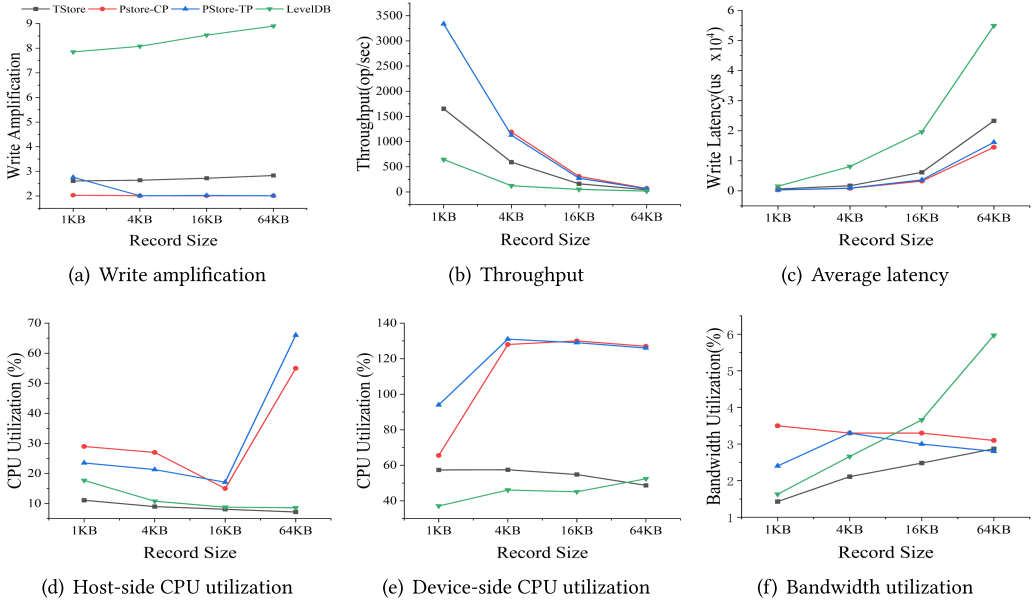


Fig. 13. Results of PStore, TStore, and LevelDB under Zipfian based on the YCSB-C with various record size.

workload; the write amplifications become 2.83 and 8.90 under the 64 KB workload. The large-size values in the host-side pipelined compaction cannot enlarge the write amplification of PStore-CP and PStore-TP, because the values are not read or written in compaction. Although large-sized records raise the overhead of compaction, the number of KV pairs decreases when data volume

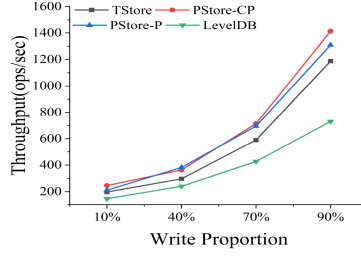


Fig. 14. Throughput under YCSB-C with the mixed read and write workloads.

is fixed. A few of compaction operations reduce CPU utilization that reaches the lowest one under the workloads with 64 KB value size. The host- and device-side CPU utilization measures in PStore-CP are 15% and 127%; when it comes to PStore-TP, these measures are 17.5% and 126%. In addition, the both-side CPU utilization in TStore are 7.2% and 48.7%, respectively. In other words, the average CPU utilization in PStore-CP and PStore-TP is 2.15 $\times$ and 1.43 $\times$ of that in TStore on the host and the device, respectively. In LevelDB, the device-side CPU utilization largely depends on operating systems, data movements, and write/read operations. The movements of large-sized records cost CPU resources, even if there is no compaction on the device. Because the large-sized records escalates bandwidth utilization, PStore-CP improves the average performance of TStore by a factor of 1.6.

Mixed read and write Workloads. We vary read ratio to study its impact on the performance of PStore (see Figure 14). Because TStore and the two PStore variants are slated to optimize compaction performance, a small number of major compaction operations has no noticeable improvement. The performance of the two PStores is on par with that of TStore and LevelDB. Processing read requests in the two variants of PStore and TStore keeps consistent. It is observed that the NDP-enabled PStore provides a distinct advantage under write-intensive workloads. When write operations account for 10%, the performance is minimized. In such a case, the throughput of PStore-CP, PStore-TP, TStore, and LevelDB are 246, 211, 197, and 147 ops/s, respectively. The performance is optimized at a high level when 90% operations are writes. Meantime, the throughput of PStore-CP, PStore-TP, TStore, and LevelDB are 1,413, 1,309, 1,187, and 730 ops/s, respectively.

6 SCALABILITY STUDY

Now, we are positioned to carry out a group of experiments to investigate the impact of cross-leveled compaction on the system performance of the *host-side pipelined compaction*-enabled PStore. On the one hand, PStore-CP has an edge over PStore-TP in terms of throughput. On the other hand, there is a narrow discrepancy in write amplification, CPU utilization, and bandwidth utilization between PStore-CP and PStore-TP. The results reveal that the cross-leveled compaction lightly improves the compaction performance when the pipelined compaction is deployed on the host. Finally, we evaluate the performance of PStore-CP, PStore-TP, PStore-C, and PStore-P to exploit how the host-side pipelined compaction and the cross-leveled compaction affect the long-tail latency. Table 3 displays workloads adopted in this group of experiments.

Aiming to boost the host-side compaction, the stages of pipelined compaction on the host embraces (1) KV pairs decoding, (2) KV pairs sorting, and (3) KV pairs encoding. Figure 15 depicts that the host-side pipelined compaction mechanism applied in PStore-CP presents lower write amplification than PStore-C, where the pipelined compaction mechanism is disabled on the host. In other words, PStore-C degrades the overall system performance when the pipelined compaction mechanism is not applied. Meantime, the device-side CPU utilization drops. Recall that the host-side

Table 3. Workload Characteristics used in Scalability Study

Workloads in DB_Bench					
Type	Feature	Fillrandom	Fillseq	Value Size (1 KB)	Data Size (10 GB)
db_bench_1	100% writes	✓		1×, 4×, 16×, 64×	1×, 2×, 3×, 4×, 5×
Workloads in YCSB-C					
Type	Feature	Data Size		Record Size	Distribution
		Load (10 GB)	Run (5 GB)	(1 KB)	
YCSB-1	50% writes	2×	2×	1×	Zipfian
YCSB-2	10% writes				
A	50% Reads, 50% Updates	1×	2×	16×	Zipfian
B	95% Reads, 5% Updates				Zipfian
C	100% Reads				Zipfian
D	95% Reads, 5% Inserts				Latest
E	95% Range Queries, 5% Inserts				Uniform
F	50% Reads, 50% Read-Modify-Writes				Zipfian

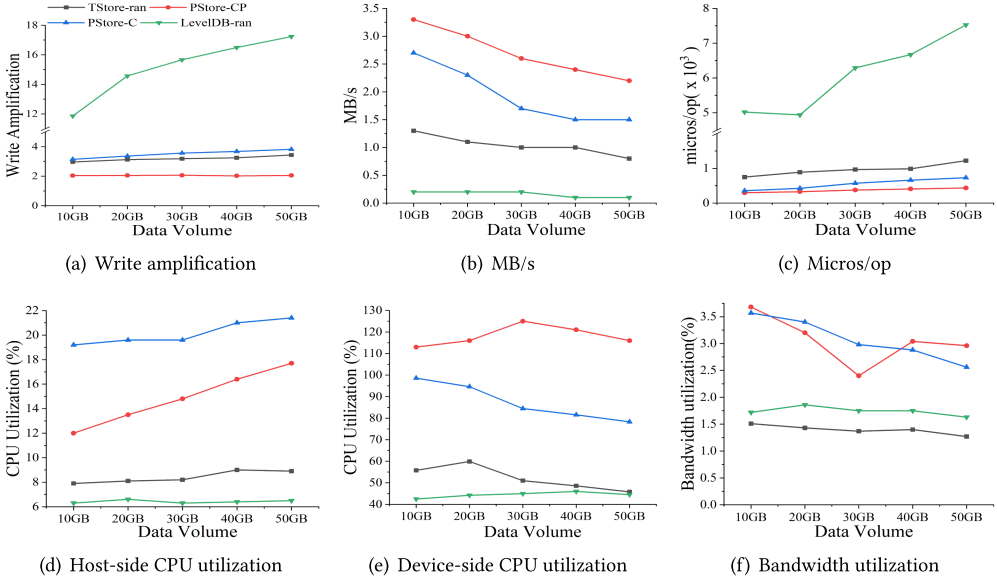


Fig. 15. Results of PStore-CP, PStore-C, TStore, and LevelDB under the Fillrandom-based DB_Bench with various data volume. PStore-C is a PStore variant without enabling the host-side pipeline compaction mechanism.

pipeline compaction module offloads decoding and encoding of KV pairs to the device. During the course of host-side compaction, the number of value transfers between host and device reduces; only keys are involved in compaction. Importantly, the three stages (KV decoding, merge-sort, and KV encoding) are executed in the pipelined mode, thereby enhancing compaction performance. The CPU utilization ratios grow through sending/receiving key arrays on the device. Thus, the performance of PStore-CP is improved, because its write amplification (around two) is relieved compared with the PStore-P case. The host-side pipelined compaction cuts back the write amplification of PStore-CP by an average of 75.4%, with an increase in PStore-CP's throughput of by 29%. The host-side CPU utilization of PStore-CP decreases by an average of 37%. Conversely, the

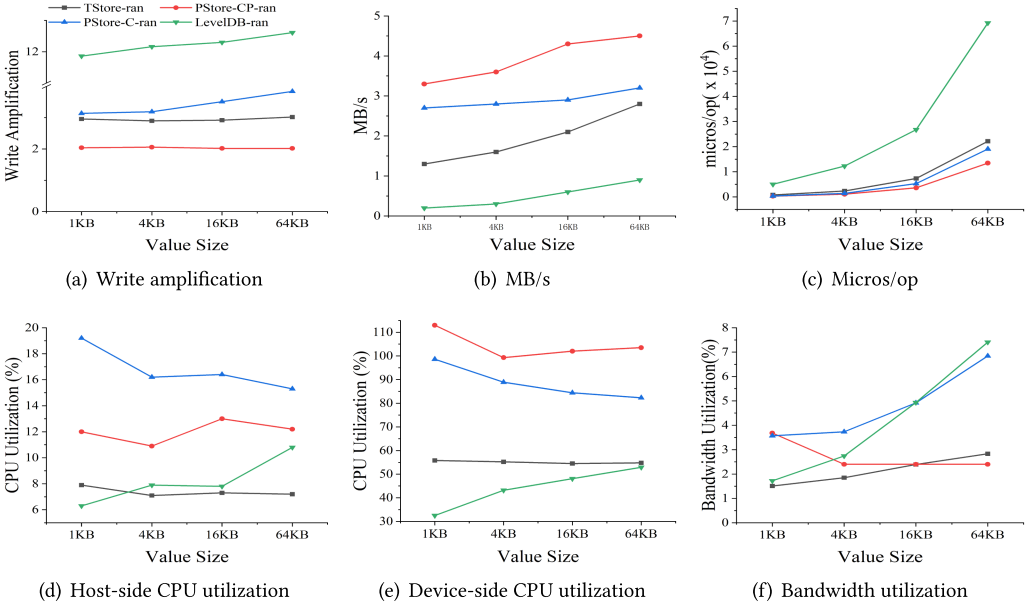


Fig. 16. Results of PStore-CP, PStore-C, TStore, and LevelDB under the Fillrandom-based DB_Bench with various value size. PStore-C is a PStore variant where the host-side pipeline compaction module is disabled.

device-side CPU utilization of PStore-CP increases by an average of 36.4%, because the decoding and encoding of KV pairs are taken place in host-side compaction. Aforementioned improvements are contributed by applying the priority scheduling scheme to compaction tasks in PStore.

Figure 16 shows that PStore-CP's write amplification keeps at 2, whereas PStore-C's write amplification increases with a growing value size. PStore-CP has a higher throughput than that of PStore-C under the workloads with large-sized values. Specifically, PStore-CP pushes up PStore-C's throughput by 0.6 and 1.3 MB/s under the workloads with 1 and 64 KB value sizes, respectively. The results unveil that the host-side pipeline compaction performance offers good optimization with a growing value size. The other performance metrics of PStore-CP have meager changes when the value size goes up, meaning that PStore-CP always provides a constant performance improvement to PStore-C. The discrepancy between PStore-CP and PStore-C is alike under various workloads with different data volume.

Non-pipelined PStore-C is superior to TStore and LevelDB on the host, even though the data volume and value size in the workloads are different. Because the asynchronous compaction mechanism delivers better performance improvement than the synchronous one, PStore-C achieves 44% and 74.5% improvement in terms of throughput compared against TStore and LevelDB, respectively.

Long-tail Latency. We examine the long-tail latency in PStore-CP, PStore-TP, PStore-C, and PStore-T to inspect the impact of host-side pipeline compaction and cross-leveled compaction on the system performance. We apply the YCSB-1 and YCSB-2 workloads to gauge the long-tail latency of the four KV stores in the three cases, namely, (1) the load state, (2) the mixed read/write state, and (3) the read-intensive state.

Figure 17 unfolds that the long-tail latency of PStore-CP and PStore-TP in YCSB-1 is 7.246 and 7.252 ms, which are on par with one another. Similarly, the long-tail latency of PStore-C and PStore-T is 6.579 and 6.867 ms, implying that the cross-leveled compaction narrowly affect the long-tail

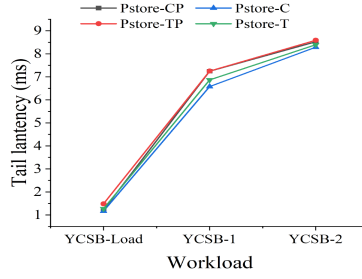


Fig. 17. Results of PStore-TP, PStore-CP, PStore-T, and PStore-C under the Fillrandom-based DB_Bench with various data volume.

latency. The decoding KV pairs are processed on the device coupled with the host-side pipeline compaction, thereby enlarging CPU cost and impairing the performance of host-side read operations. Thus, the long-tail latency of PStore-CP is 10% higher than that of PStore-C; the average response time of PStore-CP is 7% longer than that of PStore-C. In addition, the average long-tail latency of PStore-CP, PStore-TP, PStore-C, and PStore-T is 1.918, 1.921, 1.787, and 1.877 ms, respectively. Hence, the average latency of PStore-CP is 7.3% higher than that of PStore-C, and PStore-TP's latency is 2.3% larger than that of PStore-T.

The long-tail latency of PStore-CP, PStore-TP, PStore-C, and PStore-T are measured as 8.525, 8.59, 8.287, and 8.395 ms, respectively. When the workloads exhibit a high read ratio, the long-tail latency of the four KV stores is further shortened, because the two proposed techniques have zero impact on read performance. Thus, the long-tail latency measures among the four KV stores are close to each other under the YCSB-1 and YCSB-2 workloads. In the YCSB-load case, the long-tail latency readings of PStore-CP, PStore-TP, PStore-C, and PStore-T are 1.216, 1.484, 1.168, and 1.27 ms, respectively. In this case, PStore-C experiences the lowest long-tail latency among the four KV stores, entailing that the cross-leveled compaction is adroit at improving the long-tail latency of the host without the pipelined compaction mechanism. In contrast, the host-side pipelined compaction mechanism lightly increases the long-tail latency by less than 5%.

In addition, we conducted the experiments on PStore, TStore, and LevelDB using the YCSB-C workloads, including workloads A, B, C, D, E, and F. Each workload performs load and run phases. Table 3 lists the characteristics of these workloads driving in the experiment. We measured throughput, latency, write amplification, and host- and device-side CPU utilization. We cleared the previous database for each workload and loaded the data before executing the run phase. The value size in YCSB-C is 16 KB; the key size is configured to a default value in this test. We set the data volume in the load 10 GB and run 10 GB in two phases.

Throughput: In Figure 18(a), PStore improves the throughput compared with TStore and LevelDB. PStore-TP and PStore-CP perform better than the other alternative KV stores under all the workloads. For example, PStore-TP improves the throughput by 60%, 11%, 20%, 6%, 27%, 6%, and 24% over TStore under workloads Load, A, B, C, D, E, and F, respectively. This optimization becomes possible because the asynchronous mechanism of PStore separates the compaction triggering and compaction execution—and the two procedures are enabled to perform in an asynchronous manner. This approach eradicates the wait time on the host and device ends for merging compacted results on both sides, thereby curbing the compaction time. Meanwhile, in the first-level priority, according to score_trigger in Equation (1), these tasks in the queue that ought to be compacted are not considered in the compaction triggering condition. This method makes the compaction triggering reasonable. Then, compaction tasks are initiated at a level, the score_trigger of which reaches the maximum value. In the second-level priority, PStore preferentially selects the SSTable

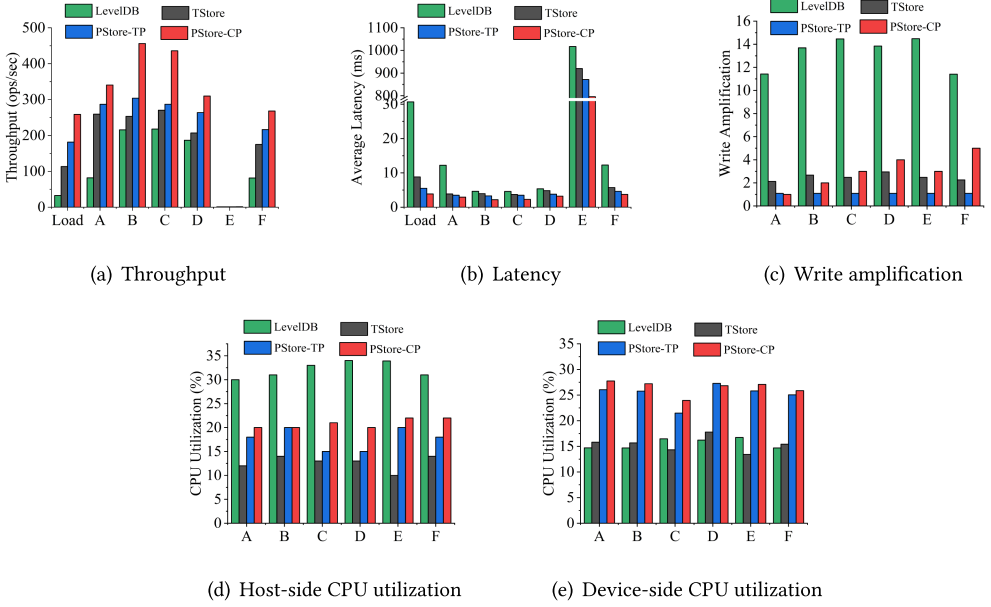


Fig. 18. Results of PStore-CP, PStore-C, TStore, and LevelDB under YCSB-C.

with the least number of key-range-overlapped SSTables. As a result, the compacted data volume and compaction time are reduced. The synchronization compaction mode in TStore divides each compaction task, followed by offloading these subtasks on the host and device. When one side finishes, it must await another side to wind up—wasting a lot of wait time. As shown in Figure 18(a), the synchronization compaction mode impairs the throughput in the two procedures in the load and run phases.

When it comes to workload A, the throughput of PStore-CP is improved by 18% over that of PStore-TP. This improvement is expected, because workload A is update-intensive, which triggers compaction operations. The cross-level compaction mechanism in PStore-CP diminishes the number of compaction operations and shortens compaction time: the throughput is upgraded. For example, PStore-TP enhances the throughput by 1.11 $\times$ and 3.50 $\times$ compared with TStore and LevelDB, respectively. For read-intensive workloads (B and C), the throughput of PStore-TP is improved by 20% and 6% compared with TStore and LevelDB, respectively. The read performance improvement of PStore-TP is marginal, because its asynchronous mechanism focuses on optimizing write performance. PStore-CP outperforms PStore-TP by 50% in throughput thanks to cross-leveled compaction. The read-intensive workload stimulates a large number of cross-level compaction operations, cutting back the number of compaction operations. Then, data is well ordered to accelerate the subsequent read performance. Workload D includes 95% reads and 5% insert operations in the tested dataset. PStore-TP revamps the throughput of TStore and LevelDB by approximately 27% and 41%, respectively. However, PStore-CP's cross-level compaction mechanism simultaneously involves data in multiple levels, which can update old data quickly with elevated throughput. The four tested KV stores share similar performance under workload E with scan. In the LSM-tree-based KV stores, the scan operations impact the system's performance, and the throughputs of PStore-CP, PStore-TP, TStore, and LevelDB are 1.26, 1.15, 1.09, and 0.98 ops/s, respectively. The range query operation, which takes the longest time, invokes the fewest compaction operations. Then, the throughput of tested KV store is small; thus, the throughput improvement offered by

PStore is minor. For workload F with mixed read and write, PStore-CP reshapes the throughput by 24% compared with PStore-TP: more writes in the workload are strong likely to demonstrate the advantages of the cross-level compaction mechanism in PStore. PStore-TP improves the throughput by 1.24 $\times$ and 2.66 $\times$ compared with TStore and LevelDB, respectively. In summary, PStore demonstrates its superiority under the YCSB-C. The collaborative compaction mode in both host and NDP-enabled device significantly curtails compaction time and revamps throughput under YCSB-C.

Latency: PStore reduces the latency in the entire process throughout the load and run phases – and the optimization in latency is on par with throughput improvement. Figure 18(b) unveils that PStore-CP and PStore-TP have a performance edge over TStore and LevelDB. In the load phase, PStore-CP shortens the latency by 56% and 87% compared with TStore and LevelDB, respectively. In the load phase, PStore slashes the latency by up to 87% than other KV stores.

Compared with TStore, in the run phase, PStore-CP reduces latency by 24%, 45%, 38%, 33%, 14%, and 35%, under workloads A, B, C, D, E, and F, respectively. In addition, PStore curtails the latency by 76%, 53%, 50%, 40%, 22%, and 70% than LevelDB under workloads A, B, C, D, E, and F, respectively; see Figure 18(b). There exist numerous compaction tasks in both load and run phases of the workloads; however, PStore with the asynchronous compaction mode continuously conducts compaction tasks on the host and NDP-enabled device sides, thereby eradicating the wait time for the completion tasks in the synchronous-compaction-mode-based KV stores. Thus, PStore optimizes the latency under a diversity of workloads. For mixed read-write workloads (A and F), PStore has an impressive capacity of cutting back latency. The latency of PStore-CP is 16% and 19% lower than that of PStore-TP under workloads A and F, respectively. This trend is reasonable, because the cross-level compaction mechanism reduces the compaction time—decreasing the latency. Under the read-intensive workloads (B and C), PStore-CP achieves around 33% and 34% performance improvement over PStore-TP. More read operations spark cross-level compaction, which lowers execution time in the run phase; therefore, the latency is shrinking. The latency of PStore-CP is 15% lower than that of PStore-TP under workload D. Under Workload E with scan, the latency of PStore is 22% smaller than that of TStore, but PStore-CP had a 10% reduction over PStore-TP in terms of latency.

Write Amplification (WA). The WA of PStore-CP and PStore-TP are lower than those of the other KV stores under these six workloads of YCSB-C (see Figure 18(c)). Compared to LevelDB, PStore-TP trims the write amplification by 92%, respectively. Meanwhile PStore-TP reduces the write amplification to 1.09 compared with TStore. The second-level priority in PStore is to elect the SSTable with the smallest number of overlapped-key-range SSTables at the lower level, reducing the amount of duplicate write data. Then, our proposed PStore optimizes the write amplification. PStore-CP outperforms PStore-TP by up to 1%, but the WA of PStore-TP is around 1.09 under YCSB-C. Then, we observe that there is a modest enhancement for PStore-CP. PStore slashes WA in the load and run phases under the mixed-read-and-write workloads A and F. The asynchronous PStore mechanism enables the host and NDP-enabled device to perform abundant compaction tasks without interruption. In the run phase, some compaction tasks are provoked by read operations in read-intensive workloads B and C. Compared to LevelDB, PStore reduced the write amplification by 92% and 93%, respectively. Meanwhile, PStore reduces the write amplification for TStore to 1.06 and 1.08. Under workloads D and E, the WA is primarily caused in the load phase, whereas in the run phase, only the triggering compaction procedure executes the data update, which causes write amplification. Because the percentage of newly written data is small, PStore-CP and PStore-TP share a similar WA measure under workload D. The cross-level compaction mode in PStore is not frequently activated; therefore, there is a rare difference between PStore-CP and PStore-TP. Under workload E, WA of PStore-CP slightly decreases (around 0.01).

CPU Utilization. Figure 18(d) unravels that the host-side CPU utilization of LevelDB is the highest under all the workloads. This observation is expected, because in LevelDB, all compaction tasks are performed on the host rather than the NDP-enabled device. The device-side resources are dedicated for data transmission and communication. Thus, the device-side CPU utilization of LevelDB is the lowest (see Figure 18(e)). Under workloads A, B, C, D, E, and F, the host-side utilization of PStore-CP is 67%, 43%, 62%, 54%, 120%, and 57% higher than that of TStore, while the device-side PCU utilization of PStore-CP is 75%, 74%, 67%, 51%, 102%, and 68% higher than that of TStore (see Figures 18(d) and 18(e)). This performance trend can be explained by the fact that the asynchronous mechanism in PStore eliminates wait time to fully utilize resources at both the host and device sides. Differences in CPU utilization of tested KV stores are apparent under YCSB-C workloads. Under workload E, the host-side CPU utilization of PStore-CP is 10% higher than that of PStore-TP, and the device-CPU utilization is 5% higher. The host-side CPU utilization of PStore-CP is 11% and 22% larger than that of PStore-TP under workloads A and F, respectively. The device-side CPU utilization of PStore-CP is 7% and 3% larger than that of PStore-TP under workloads A and F, respectively. The host-side CPU utilization of PStore-CP is 40% higher than that of PStore-TP under read-intensive workloads such as B and C. For instance, given workload B, the host-side CPU utilization is unchanged, and the device-side CPU utilization achieves an increment of 11%. As shown in Figures 18(d) and 18(e), the asynchronous mechanism of PStore elevates the CPU utilization on both ends. Under workload C, PStore-CP bolsters host-CPU utilization by 40% than PStore-TP, whereas the host-side CPU utilization remains unaltered under workload B. The device-side CPU utilization is improved by 6% and 11% under workloads B and C, respectively. Figures 18(d) and 18(e) reveal that the asynchronous mechanism of PStore find-tunes the CPU utilization on both sides.

7 RELATED WORK

LSM-tree-based Key-value Store. Log-structured merge tree (LSM-tree)-based key-value stores (i.e., KV stores) [20, 27] increasingly become one of mainstream data management methods in large-scale storage systems. It is arduous for KV stores to deal with the compaction procedure – a key player determining the performance of large-scale storage systems.

Some solutions were proposed to alleviate the compaction problem in KV stores. For instance, *bLSM* [40], which takes full advantage of LSM-tree and B-tree, involves a merge scheduler to lift write throughput. In addition, *bLSM* advocates for a bloom filter to optimize merge processes, thereby avoiding I/O access latency. *VT-tree* [30]—an extension of LSM-tree—makes use of file systems (B+ tree and LSM-tree) to improve I/O performance. *RocksDB* [27] stirs multiple contiguous levels to sort and merge data, which reduces the compaction frequency. LSM-tree [41] relieves write amplification compared with a KV store. The LSM-tree scheme enhances the performance of random accesses on the structure of multiple sorted tables at each level. *LOCS* [28] extends LevelDB that applies internal parallelism (i.e., channels and chips) in SSDs to revamp performance of KV stores. *cLSM* [10] is another efficient concurrent KV store that exploits the scalable concurrency using multi-core processors. The pipelined compaction procedure [44] mainly utilizes the parallelism of CPUs and I/Os on the host. *WiscKey* [24] leveraged an internal parallelism inside an SSD and multiple threads; this technique separates keys and values to achieve high range-query performance. *Atlas* [19] embraces a distributed KV store system to store keys and values on multiple hard drives.

FD-trees [21], utilizing an LSM-tree-like index, applies the logarithmic and fractional cascading methods to drive up read performance on flash memory-based devices. *LSbM* [36] leverages a buffer management on the disk to minimize cache invalidation caused by device-side compaction. In contrast, our PStore optimizes compaction on both host- and device-side subsystems.

Meantime, PStore makes full use of computing resources on both sides to speed up the overall system performance.

Hardware Acceleration for Compaction. Modern hardware architectures eliminate I/O limitations thanks to high-performance interfaces. Thus, the performance bottleneck of KV stores—compaction—can be accelerated by new hardware architectures. MatrixKV [43] is deployed in DRAM, NVM-PMEM, and SSD. Level L0 is placed in NVM-PEME, and compaction in L0–L1 is at other levels to alleviate I/Os competition. NoveLSM [17] extends a component on NVMs to store writes as memory components are full—averting write stalls. A FPGA-based acceleration engine for KV stores [35][13] were designed to speed up the compaction via hardware-software collaboration—improving throughput and averting resource contention. GPGPUs strengthens performance of KV stores in storage systems. Tayler et al. [12] evaluated the performance of memory KV stores on a GPGPU. GPUfs [31], providing a POSIX API for GPGPU programming, exploits parallelism and extends a buffer cache optimize file accesses. LUDA [42] uses GPGPU kernels to accelerate compaction performance.

Near-data Processing in Storage. Current computation in a store device mainly focuses on data management such as address mapping. Although active disks [18, 29] provide powerful computing resources to boost I/O-access performance, the active disk technique has not yet been widely adopted in large-scale storage systems.

Recently, the framework of active disks takes advantages of flash memory that is being attractive to the academia and industry. Internal bandwidth of flash memory is higher than the performance of the interface between hosts and devices [8]. In addition, the computation controller on a flash memory-based device mainly runs a flash translation layer that effectively deals with computing-intensive tasks. Thus, a growing number of software stacks have been deployed to the devices making use of computing resources [5, 38].

A smart flash memory-based SSD [7] was designed by adding a stream processor to existing flash controllers. Thanks to applying its heterogeneous computing resources managed by the developers, the performance of data processing is immensely revamped. Nevertheless, it is arduous to program stream processors under intensive workloads. Another study [8] offloads query tasks into smart SSDs, where APIs were slated to manage commands and query threads in SSDs. However, the internal CPUs limits the performance of data processing when large-scale data are queried. Kang [16] constructed an intelligent SSD model to improve I/O performance by leveraging the advantages of the CPUs on both sides and flash memory devices. The model, which is known as in-storage computing (ISC), applies computing resources of SSDs for performance optimization [6, 26]. Gu [11] defined a computing unit in ISC as *SSDlet*, which coordinates applications in a cluster-like mode.

Liang [22] proposed a cognitive SSD, which involves an energy-efficient engine for deep-learning-based unstructured data retrieval. In a cognitive SSD, a flash-accessing accelerator named DLG-x is placed to the side of flash memory to achieve near-data-based deep learning and graph searching. *nKV* [37] is a KV store that utilizes native computational storage and near-data processing to directly eliminate intermediary layers among the I/O stacks on NVM/flash storage. PinK [14] is a KV-SSD based on the LSM-tree. Jin et al. [15] introduce the key-addressable multi-log SSD with a KV interface, advocating for variable-length records to store data.

Sun et al. developed *Co-KV* [33] to conduct compaction tasks under an NDP framework where ARM-based computing resources are employed. Afterward, *TStore* [32] and *DStore* [34]—two dynamic offloading schemes—were built to speed up compaction performance in KV stores. The compaction mode in these NDP-enabled KV stores is synchronous. The host-side compaction tasks are partitioned and offloaded to the NDP-enabled device side, synchronizing and parallelizing compaction on both sides. Unfortunately, the synchronous compaction mode cannot

narrow the computing-capacity gap between the host and its device. Therefore, the execution time is imbalanced between the two sides, degrading the system performance. To address this issue, our proposed PStore—an asynchronous parallel compaction mode for NDP-enabled KV stores—fully takes advantage of the computing resources of NDP-enabled SSDs to enhance compaction performance.

8 CONCLUSIONS

LSM-tree-based key-value store systems employ background compaction to merge key-value pairs, which are sorted for queries and scans. Recognizing that the compaction significantly decreases the system performance under random-update workloads, we proposed an NDP-enabled framework—PStore—anchored on an asynchronous parallel compaction mechanism. PStore is adroit at improving compaction performance in KV stores. Comparing against the other existing NDP-based KV stores that adopt the synchronous parallel compaction mode, PStore takes advantage of the computing resources on the host and its NDP-enabled storage device powered by the three priority scheduling schemes to achieve task-level parallelisms. We implemented cross-leveled compaction model and key-value separation mechanism to mitigate write amplification. The experiment results unveil that PStore is adept at revamping the compaction performance on the testbed of NDP-based systems.

In a nutshell, PStore is an NDP-enabled compaction engine playing a critical role of running compaction thread in KV stores, which require seamless integration on the host. As a future research direction, we intend to expand PStore to be an optimization catalyst for a wide range of KV stores such as *RocksDB*.

REFERENCES

- [1] AAEON. 2021. RICO-3399. Retrieved from <https://newdata.aaeon.com.tw/DOWNLOAD/2014%20datasheet/Boards/RICO-3399.pdf>
- [2] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H. Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. 2014. Near-data processing: Insights from a MICRO-46 workshop. *IEEE Micro* 34, 4 (2014), 36–42.
- [3] Helen H. W. Chan, Chieh-Jan Mike Liang, Yongkun Li, Wenjia He, Patrick P. C. Lee, Lianjie Zhu, Yaozu Dong, Yinlong Xu, Yu Xu, Jin Jiang et al. 2018. HashKV: Enabling efficient updates in KV storage via hashing. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'18)*. 1007–1019.
- [4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.* 26, 2 (2008), 4.
- [5] Benjamin Y Cho, Won Seob Jeong, Doohwan Oh, and Won Woo Ro. 2013. XSD: Accelerating mapreduce by harnessing the GPU inside an SSD. In *Proceedings of the 1st Workshop on Near-Data Processing*.
- [6] Sangyeun Cho, Sanghoan Chang, and Insoon Jo. 2015. The solid-state drive technology, today and tomorrow. In *Proceedings of the IEEE 31st International Conference on Data Engineering (ICDE'15)*. IEEE, 1520–1522.
- [7] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R. Ganger. 2013. Active disk meets flash: A case for intelligent SSDs. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ACM, 91–102.
- [8] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query processing on smart SSDs: Opportunities and challenges. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 1221–1230.
- [9] Maya Gokhale, Bill Holmes, and Ken Iobst. 1995. Processing in memory: The terasys massively parallel PIM array. *Computer* 28, 4 (1995), 23–31.
- [10] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems*. Association for Computing Machinery, Article 32, 14 pages.
- [11] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanhoo Yoon, Sangyeun Cho et al. 2016. Biscuit: A framework for near-data processing of big data workloads. In *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. IEEE, 153–165.

- [12] Tayler H. Hetherington, Timothy G. Rogers, Lisa Hsu, Mike O'Connor, and Tor M. Aamodt. 2012. Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. 88–98.
- [13] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-engine: An optimized storage engine for large-scale E-commerce transaction processing. In *Proceedings of the International Conference on Management of Data*. 651–665.
- [14] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. 2020. PinK: High-speed in-storage key-value store with bounded tails. In *Proceedings of the USENIX Annual Technical Conference*. 173–187.
- [15] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. 2017. KAML: A flexible, high-performance key-value SSD. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*. 373–384.
- [16] Yangwook Kang, Yang-suk Kee, Ethan L. Miller, and Chanik Park. 2013. Enabling cost-effective data processing with smart ssd. In *Proceedings of the IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST'13)*. IEEE, 1–12.
- [17] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for nonvolatile memory with NovelSM. In *Proceedings of the USENIX Annual Technical Conference*. 993–1005.
- [18] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. 1998. A case for intelligent disks (IDISKS). *ACM SIGMOD Rec.* 27, 3 (1998), 42–52.
- [19] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. 2015. Atlas: Baidu's key-value storage system for cloud data. In *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies (MSST'15)*. IEEE, 1–14.
- [20] Leveldb. 2018. fast and lightweight key/value database library by Google. Retrieved from <http://code.google.com/p/leveldb>
- [21] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. 2010. Tree indexing on solid state drives. *Proc. VLDB Endow.* 3, 1-2 (2010), 1195–1206.
- [22] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. 2019. Cognitive SSD: A deep learning engine for in-storage data retrieval. In *Proceedings of the USENIX Annual Technical Conference (USENIXATC'19)*. 395–410.
- [23] Ling Liu. 2013. Computing infrastructure for big data processing. *Front. Comput. Sci.* 7, 2 (2013), 165–170.
- [24] Lanyue Lu, Thanumalayan Sankaranarayanan Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating keys from values in SSD-conscious storage. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*. USENIX Association, 133–148.
- [25] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inform.* 33, 4 (1996), 351–385.
- [26] Dongchul Park, Jianguo Wang, and Yang Suk Kee. 2016. In-storage computing for hadoop MapReduce framework: Challenges and possibilities. *IEEE Trans. Comput.* PP, 99 (2016), 1–1.
- [27] RocksDB. 2018. A persistent key-value store. Retrieved from <https://rocksdb.org/>
- [28] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A general purpose log structured merge tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 217–228.
- [29] CALD Seminar. 1998. Active storage for large-scale data mining and multimedia. *Center for Automated Learning and Discovery* (1998), 62–73.
- [30] Pradeep Shetty, Richard Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. 2013. Building workload-independent storage with VT-trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*. USENIX Association, 17–30.
- [31] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating a file system with GPUs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 485–498.
- [32] Hui Sun, Wei Liu, Jianzhong Huang, Song Fu, Zhi Qiao, and Weisong Shi. 2019. Near-data processing-enabled and time-aware compaction optimization for LSM-tree-based key-value stores. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–11.
- [33] Hui Sun, Wei Liu, Jianzhong Huang, and Weisong Shi. 2019. Collaborative compaction optimization system using near-data processing for LSM-tree-based key-value stores. *J. Parallel Distrib. Comput.* 131 (2019), 29–43.
- [34] Hui Sun, Wei Liu, Zhi Qiao, Song Fu, and Weisong Shi. 2018. DStore: A holistic key-value store exploring near-data processing and on-demand scheduling for compaction optimization. *IEEE Access* 6 (2018), 61233–61253.
- [35] Xuan Sun, Jinghuan Yu, Zimeng Zhou, and Chun Jason Xue. 2020. FPGA-based compaction engine for accelerating LSM-tree key-value stores. In *Proceedings of the IEEE International Conference on Data Engineering*. 1261–1272.
- [36] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Siyuan Ma, Yanfeng Zhang, and Xiaodong Zhang. 2017. LSbM-tree: Re-enabling buffer caching in data management for mixed reads and writes. In *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems*.

- [37] Tobias Vinçon, Arthur Bernhardt, Ilia Petrov, and Andreas Koch. 2020. nKV in action: Accelerating KV-stores on native computational storage with near-data processing. *Proc. VLDB Endow.* 13, 12 (2020), 2981–2984.
- [38] Jianguo Wang, Dongchul Park, Yang Suk Kee, Yannis Papakonstantinou, and Steven Swanson. 2016. SSD in-storage computing for list intersection. In *Proceedings of the International Workshop on Data Management on New Hardware*. 4:1–4:7.
- [39] Jianguo Wang, Dongchul Park, Yannis Papakonstantinou, and Steven Swanson. 2016. SSD in-storage computing for search engines. *IEEE Trans. Comput.* (2016), 1–1. <https://doi.org/10.1109/TC.2016.2608818>
- [40] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree-based key-value store on open-channel SSD. In *Proceedings of the 9th European Conference on Computer Systems*. ACM, 16.
- [41] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based ultra-large key-value store for small data. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference*. 71–82.
- [42] Peng Xu, Jiguang Wan, Ping Huang, Xiaogang Yang, Chenlei Tang, Fei Wu, and Changsheng Xie. 2020. LUDA: Boost LSM key value store compactions with GPUs. Retrieved from <https://arxiv.org/abs/2004.03054>
- [43] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing write stalls and write amplification in LSM-tree-based KV stores with matrix container in NVM. In *Proceedings of the USENIX Annual Technical Conference*. 17–31.
- [44] Zigang Zhang, Yinliang Yue, Bingsheng He, Jin Xiong, Mingyu Chen, Lixin Zhang, and Ninghui Sun. 2014. Pipelined compaction for the LSM-tree. In *Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 777–786.

Received 28 January 2023; revised 19 August 2023; accepted 21 September 2023