DStore: A Holistic Key-value Store Exploring Near-Data Processing and On-Demand Scheduling for Compaction Optimization

HUI SUN1, WEI LIU1, ZHI QIAO 2, SONG FU2, (SENIOR MEMBER, IEEE) AND WEISONG SHI3, (FELLOW, IEEE)

1School of Computer and Science, Anhui University, Hefei, China. (e-mail: sunhui@ahu.edu.cn, liuweiahu@gmail.com)
2Computer Science and Engineering, University of North Texas, Denton, Texas, USA (e-mail: ZhiQiao@my.unt.edu, Song.Fu@unt.edu)
3Department of Computer Science, Wayne State University, Detroit, MI, USA. (e-mail: weisong@wayne.edu)

This work was sponsored in part by National Natural Science Foundation of China under Grants 61702004 and 61572209. This work was also supported in part by an U.S. National Science Foundation grant CCF-1563750.

ABSTRACT Log-structured merge-tree (LSM-tree) based key-value stores are widely deployed in large-scale storage systems. The underlying reason is that the traditional relational databases cannot reach the high performance required by big-data applications. As high-throughput alternatives to relational databases, LSM-tree-based key-value stores can support high-throughput write operations and provide high sequential bandwidth in storage systems. However, the compaction process triggers write amplification and is confronted with degraded write performance, especially under update-intensive workloads.

To address this issue, we design a holistic key-value store to explore near-data processing and on-demand scheduling for compaction optimization in an LSM-tree key-value store, named DStore. DStore makes full use of various computing capacity in the host- and device-side subsystems. DStore dynamically divides the whole host-side compaction tasks into the above two-side subsystems according to two-side different computing capabilities. Meanwhile, the device must be featured with a near-data processing model. The divided compaction tasks are performed by the host and the device in parallel. In DStore, the NDP-based devices exhibit low-latency and high-bandwidth performance, thus facilitating key-value stores. DStore does not only accomplish compaction for key-value stores but also improve the system performance.

We implement our DStore prototype in a real-world platform and different kinds of testbeds are employed in our experiment. LevelDB and a static compaction optimization using the NDP model (called Co-KV) are used to compare with the DStore in our evaluation. Results show that DStore achieves about 3.7x performance improvement over LevelDB under the db_bench workload. In addition, DStore-enabled key-value stores outperform LevelDB by a factor of about 3.3x and 77.0% in terms of throughput and latency under YCSB benchmark, respectively.

INDEX TERMS LSM-tree, Key-Value Store, On-Demand Scheduling, Near-Data Processing

I. INTRODUCTION

DC reveals that the digital universe is doubling in size every two years. By 2020, the digital universe - the data we create and copy annually - will reach 44Zettabytes (ZB). Data-intensive applications have become mainstream in the big data era. With the growth of data volume in these applications, storage systems must support high performance, reliability, and real time. Relational database management systems (i.e., RDBMS) cannot offer scheduling that is efficient enough to manage a large amount of data in large-scale distributed storage systems due to their complex transaction consistency and heavy SQL queries [1] [2]. To deal with these issues, log-structured merge-tree (LSM-tree) [3] based key-value stores with high performance are widely employed in storage-level and memory-level systems [4] [5]. Many famous IT companies built their data infrastructures using LSM-tree based key-value stores, e.g., Google with BigTable [6] and LevelDB [7], Apache with HBase [8], Facebook with...
RocksDB [5], etc.

The LSM-tree based key-value stores provide high throughput and play an important role in big data applications. It has two data structures, i.e., a memory component and a disk component. When the memory component is full, it dumps buffered data into a file (called SSTable), the key range of which may overlap with those in the underlying LSM-tree. These device-side SSTables with overlapping key ranges transfer to the host-side memory perform sort, and multi-merge operation, which will produce new non-overlapping SSTables. This process called compaction is the feature of LSM-tree based key-value stores and plays an important role in a fast data query. However, compaction moves copious amounts of background data from the device to the host-side memory, which affects the foreground performance in the overall system. Furthermore, there is more data volume than user requirement written into the device after compaction, which is named write amplification. Thus, these two shortcomings prevent user I/Os in applications from transmitting data between the host and the device until the compaction is complete. In this process, the overall system performance is dramatically degraded by write amplification in large-scale storage systems.

To address this issue, most optimized solutions for the compaction problem includes three aspects. First, the host-side approaches to reduce I/O accesses to the device in compaction [9]. The host-side parallelism of CPU and I/O resources is employed to improve compaction [10]. The CPU usage in the host increases while the computation resources in the device are underutilized. Second, a prior method of offloading the whole compaction from the host into the device is used to optimize compaction [11]. This approach avoids intensive data movement but relies on CPU computation in a resource-constrained device. It barely maximizes the overall system performance. Third, new high-throughput devices have been widely employed in the storage systems to improve I/O performance [12] [13]. Therefore, the I/O latency can be improved in compaction. However, the update-intensive workloads also aggravate the high I/O latency in compaction [14].

With the computation power in the device increasing, a framework named near-data processing (i.e., NDP) [15] [16] or in-storage computing (i.e., ISC) [11] [17] [18] [19] [20] has attracted a great deal of attention from academia and industry. This framework aims to minimize data movement by moving computing at the resident location of data. It is different from the data movement toward a CPU independent of where it resides, as compaction is done in the host. This principle is applied in the LSM-tree by placing computing resources close to where the data is located. However, researchers may ignore the various computing capabilities in the host and the NDP-based devices, and thus they do not explore the collaborative scheduling between both sides to improve compaction.

In this paper, our goal is two-fold: (1) optimize a load of compaction of LSM-tree based key-value stores by using the near-data processing model to exploit overall system performance efficiently; (2) adjust dynamically offloading compaction into the host- and device-side subsystem according to different computing capabilities on both sides. The host-side compactions are divided into two kinds of compaction subtasks, which are offloaded into the host- and device-side subsystems, respectively. This method can reduce the load of compaction in the host side. An on-demand scheduling is an offloading optimization scheme for LSM-tree based key-value stores (i.e., DStore) on the basis of the compaction processing capability in the host and the device. We find compaction dramatically degrade in three kinds of phenomena: (1) the overall system can perform in the LSM-tree under write-intensive workloads; (2) the emergent near-data processing framework can perform applications with the computation in the device; (3) various computing capabilities exist in the host and the device because of the their different computation resources. These findings motivate us to implement DStore. There are three contributions in our manuscript:

We employ the near-data processing framework and on-demand scheduling to compact SSTables with overlapping key ranges in the device. It alleviates the cost of data movement from the device to the host, thereby improving compaction performance.

We implement an on-demand compaction scheduling for the LSM-tree based key-value stores according to various compaction processing capability in the host- and the device-side subsystems. This scheduling dynamically offloads the two sub-tasks of compaction into both sides and they are performed in parallel.

We propose a collaborative framework to optimize the LSM-tree based key-value stores compaction performance by using the various computation on both sides. A real-world NDP-based DStore platform is first designed to validate the benefits of compaction optimization by extensive experiments.

Our paper is organized as follows. In Section II, we present background about the LSM-tree based key-value stores and the near-data processing model. In Section III, the motivation of our work is presented. Section IV gives an overview of the DStore and its implementation. Sections V and VI present extensive performance evaluation and sensitivity study. Section VII provides the related works and we make conclusion in Section VIII.

II. BACKGROUND
In this section, we present the background of LSM-tree based key-value stores [5] and the compaction procedure in LevelDB, a popular extended version of LSM-tree. Then a short description is given for near-data processing.

A. LSM-TREE AND COMPACTION
LevelDB is an important variant of the LSM-tree. LSM-tree architecture, and its compaction procedure in LevelDB are shown in FIGURE1.
The LSM-tree offers high-throughput sorting of key-value items, which is conducive to fast data retrieval. The LSM-tree consists of a buffer component ($C_0$) in memory and multiple components ($C_1$, $C_2$, \ldots, and $C_k$) in the disk (see FIGURE 1(a)).

In LevelDB, a key-value pair from an application is first located in an in-memory buffer (i.e., MemTable) and is appended to a write-ahead log (i.e., WAL) for recovery. Once a MemTable reaches its maximum size, it converts into an immutable MemTable and is dumped into a sorted string table, (i.e., SSTable) in the disk.

A new SSTable is placed at a high level, such as $L_0$. To keep high throughput, SSTables have overlapped key ranges while the low levels (from $C_2$ to $C_k$) have distinct non-overlapped key ranges.

When the number of SSTables at a level reaches the threshold, a compaction is triggered. In this process, an SSTable from level $L_k$ is migrated to level $L_{k+1}$. Meanwhile, the overlapped-key-range SSTables at a low level are set to be candidate ones in compaction. MakeInputIterator reads these SSTables at the low level to perform a merge-sort operation. If there is no SSTable, a newly empty SSTable stores the ordered key-value pairs by OpenCompactionOutputFile. When a SSTable is full, FinishCompactionOutputFile is called to complete the file content synchronization and verification operations. When the iterator traverses the contents of all SSTables, the ordered ones are produced in a disk once the compaction completes. After integrating the contents of all SSTables, the ordered ones are produced in an immutable MemTable and is dumped into a sorted string table.

Step 4 Making key-value pairs order after the merge sort and writing them into new SSTables.

Step 5 Updating the host-side meta-data when the compaction completes.

Step 6 Writing new SSTables into the device from the host-side memory.

Step 7 Waiting for the next compaction.

The compaction being executed asynchronously in the background affects the performance of the key-value store and incurs high write latency for foreground applications. Write amplification, which means that more data than user requirement is written into the device, occurs in this process. For example, the written data volume of level $L_{k+1}$ is about ten times larger than in $L_k$ in the worst case in the process of compaction [9].

B. NEAR-DATA PROCESSING

The high overload of data movement largely influences the performance and energy consumption in the systems under computing-intensive applications. The more data moves to the host-side memory from the device, the larger the scale of data volume increases. However, a data-intensive processing model rather than a computing-intensive model is becoming the mainstream in large-scale systems. The I/O interface between device-side storage and host-side memory is always the bottleneck in high-performance systems, which are evenly aggravated by data movement from storage to memory. Currently, a new notion of moving computation to data source called near-data processing (i.e., NDP) raises concerns from academia and industry. Computation can be placed in the memory level [21][22] or storage level [23][24][25]. The former is also set as processing in memory (i.e., PIM), and the latter one is referred to as in-storage computing as well.

In NAND Flash memory, the NDP model improves the NAND Flash-based storage system because of the advent of high performance inside the flash memory. The NDP framework that is available in NAND Flash memory relies on the following two reasons: (1) The bandwidth of the flash memory is much higher than the I/Os between host and device. (2) Inside a flash memory-based device, the computation controller, which mostly runs a lightweight flash translation layer, can be extended to perform computing-intensive tasks. In big-data workloads, the storage-level NDP (or in-storage computing) becomes increasingly popular in high-performance systems.

In this manuscript, we focus on the storage-level NDP to optimize the compaction performance in the LSM-tree based key-value stores by the host- and device-side collaboration. See more details in Section IV.

III. MOTIVATION

Motivation 1. According to the sub-processes in compaction, SSTables at a high level are independent and do not have overlapped key ranges. It motivates us to preprocess upper SSTables and divide the host-side compaction into several steps.
sub-processes, which are independent of each other. An NDP-based device provides the computing capability needed to perform a portion of compaction tasks in the device. Thus, the compaction sub-tasks are dynamically sent to the device and the host to perform compaction in parallel by using their different compaction processing capability. In addition, an on-demand-based compaction task offloading scheduling is designed on the basis of computing capability on NDP-based storage systems. The overload of data movement between the two sides reduces; thereby, improving the overall performance. Similar to pipelined [10], we can optimize steps in compaction (see Section II-A) in parallel:

Step 1 The host-side scheduling performs the compaction process and calculates all SSTables in this process.

Step 2 The host sends the information of this compaction, e.g., the metadata set of the SSTables, the ratio of compaction processing capability between the host and the device.

Step 3 When it receives information, the device creates a task queue which consists of compaction subtasks. The host- and device-side task dispatching threads fetch tasks from the queue and send them to the compaction executor in two-side subsystems, respectively.

Step 4 The device receives the compaction subtasks.

Step 5 The device transfers the SSTables involved in compaction to its memory through an internal interface in compaction.

Step 6 The key-values in SSTables are sorted to new SSTables in the host.

Step 7 The ordered key-value pairs are sorted into new SSTables in the device.

Step 8 The device sends a metadata set of new SSTables produced by all compaction subtasks into the host.

Step 9 The host performs three types of operation including integrating the metadata, deleting the expired files, and updating manifest and log files.

We can see that Steps 4 and 5, 6 and 7, and 7 perform in parallel. The frequent compaction enlarges the load on the host-side CPU as the volume of data movement increases. We employ the computing capability of the near-data processing model (i.e., NDP) into the optimization of a key-value store. Using the NDP, we proposed an on-demand scheduling to automatically offload the compaction tasks into the host- and device-side subsystems according to their various compaction processing capability; thus, the two-side subsystems can collaboratively perform compaction sub-tasks in parallel. The burden on the host-side CPU and the interface are relieved, thereby optimizing the overall performance of key-value stores.

Motivation 2. We find that NDP-based devices have computation resources to perform compaction tasks. The static scheduling of offloading compaction called Co-KV in our prior work [26] fixes the ratio of compaction sub-tasks offloading into the host- and device-side subsystems. Compared to LevelDB, the host-side CPU overload and the data traffic from the device is reduced; however, the static offloading scheduling of Co-KV rarely considers the heterogeneity compaction processing capability in the host and the device. The static scheduling of offloading compaction can allocate much compaction tasks than the counterpart. The overall system performance is largely dependent on the longest time of the compaction sub-task. In FIGURE 2, the execution duration of the host and the device are inconsistent, which results in a long latency.

Because the execution time in the host and the device is different, the static scheduling of compaction offloading cannot fully utilize the computation capability on the host and device sides. Therefore, it is unreasonable to statically offload the tasks into both sides according to the fixed ratio of offloading compaction tasks.

Our work focuses on the improvement of compaction in LSM-tree based key-value stores under the random-write workloads. We aim to propose a dynamic offloading scheduling for the compaction by using the compaction processing capability in the host- and the device-side subsystems. The expected scheduling is targeted at collaboratively improving compaction between two-side systems and optimizing compaction performance (see FIGURE 2).

IV. DSTORE DESIGN AND IMPLEMENTATION
In this section, we introduce the on-demand scheduling for the collaborative improvement of compaction sub-tasks based on the different computation resources in the host and the device. The implementation of the DStore prototype system based on this scheduling is presented in this section.

A. DSTORE OVERVIEW
The host-side DStore mainly completes the following functions. (1) Receiving the user read/write requests from the upper application. (2) Triggering the compaction process. (3) Managing the metadata consistency in the compaction processing. (4) Sending the message in the compaction process. (5) Receiving and processing host-side compaction tasks. (6) Redirecting storage I/O operations.
A compaction contains multiple subtasks, each of which contains several SSTables to perform compaction. The device-side DStore handles the following operations. (1) Analyzing the information of compaction. (2) Establishing the dual-end compaction task queue and creating and maintaining the distribution thread of compaction subtask. (4) Executing the storage access redirection operations. (5) Receiving and processing the device-side compaction subtasks.

The runtime system is responsible for the storage I/O redirection and manages the information interaction of compaction subtasks between the host- and device-side subsystems. Then the reliability of the on-demand based key-value storage system is guaranteed by the runtime system.

1) Host-side DStore System
The host-side DStore system provides native APIs which are compatible with those in LevelDB. Application developers can seamlessly migrate their applications to the DStore. This subsystem is responsible for receiving application requests from the upper levels to the key-value store. It determines whether the compaction process is triggered when processing these requests. When the compaction occurs, the host-side DStore calculates the number of levels, SSTable metadata, and the ratio of compaction processing capability between the host- and device-side DStore. These messages are packaged into interactive information through the runtime system. In this paper, the information format is defined [packetHeader, packetLength, TaskOffload Ratio, CompactionInfo, Checksum]. The interaction information is sent to the runtime system in the device-side DStore through the host interface.

The device-side runtime system receives and parses the message for compaction. Then, the double-ended queue for the compaction is created by using all SSTables in this compaction and based on the ratio of the processing capability between the host and the device. Meanwhile, two threads exclusively distribute the subtasks of compaction from the head (or tail) of the queue to the host-side (or device-side) subsystem in DStore. The compaction execution module on the host (or the device) is named H-Compaction (or D-Compaction) module.

When sending the information of a compaction to the device via the host interface, the host-side DStore system recurrently receives the compaction subtasks sent by the dispatch thread of subtask in the device-side system. When the host-side system receives the task information, the H-Compaction module analyzes the task information and determines whether the received information is the compaction subtask or the completion information for the compaction process.

(1) If the host-side DStore subsystem receives the compaction subtasks, the H-Compaction module processes the received compaction subtasks and records the data volume of all SSTables in the subtask \((DV_{hi}, \text{ data volume in the host})\) and completion time for this compaction \((T_{hi})\). The H-Compaction module evaluates the capability of processing for the compaction in the host-side DStore according to \((DV_{hi}/T_{hi}, 1 \leq i \leq N, N\) is the number of compaction subtasks in the host-side compaction process).

(2) If the host-side DStore subsystem receives the completion information for the compaction process, the H-Compaction module calculates the average processing capability for all the compaction subtasks \((\text{Perf}_{hi} = (\sum_{i=1}^{N} DV_{hi}/T_{hi})/N)\) in the entire compaction process. The value is used to evaluate the capability of the compaction process in this subsystem.

Note that all I/O accesses in the host-side DStore subsystem must be configured as the interactive information according to the following format [packetHeader, packetLength, FileOpCmdMapTable, Checksum]. These operations from the host are redirected to the device-side system by the APIs in the runtime system. The redirection operation is performed by the runtime system module of the DStore device-side system, and the operation result is transmitted to the host-side DStore system.

In addition, the host-side DStore system is also responsible for the consistency management of the metadata during the compaction.

After all the host-side compaction subtasks in the queue are completed, the host receives the metadata of new SSTables and the average value of processing capability for the device-side compaction. Then, the host-side system integrates the metadata of new SSTables from the host and the device, deletes the expired files, and updates the manifest and the log file.

Then, the host-side DStore begins to perform the on-demand decision. The host calculates the ratio of the compaction processing capability between the host and the device for the next compaction by using the current average value of the processing capability in the two-sided systems. When the new compaction process is triggered, the host-side system sends the ratio and the interaction message of the compaction
to the device-side DStore system. This information is employed to build a double-ended task queue for the on-demand scheduling to optimize compaction.

2) Device-side DStore System

The device-side DStore system periodically receives the interaction information of the compaction sent by the host-side DStore. At the same time, the runtime system calculates the number of levels, file metadata set, and the ratio of compaction processing capability in this compaction.

Based on this information, the device-side DStore system creates a double-ended task queue for the compaction. In our prototype, the task distribution thread on the host side fetches subtasks from the head to the tail of the queue and sends the subtask information to the host-side DStore through the host interface. The H-Compaction module in the host-side system executes compaction. Another task distribution thread on the device side fetches the subtasks from the opposite side and sends the subtasks to the D-Compaction module through shared memory. The two threads take the compaction subtasks from the queue until it is empty.

The method of handling the compaction subtasks in D-Compaction differs from that in H-Compaction.

When the H-Compaction processes the SSTables in compaction, it needs to redirect the storage accesses to the device-side DStore through the host- and device-side runtime system, and perform the corresponding file operations. When D-Compaction stores the SSTables in the compaction, the POSIX API interface provided by device-side-DStore operating system can be used directly rather than forwarding the system through interactive interaction at both sides.

When D-Compaction receives the compaction subtasks, it directly executes compaction on the device. When a compaction subtask is processed, the data volume of all SSTables in the subtask \(D_{V_D}\) and the completion time \(T_D\) of this compaction subtask are recorded. \(D_{V_D}/T_D\), \(1 \leq i \leq M\), where \(M\) is the number of sub-tasks for the device-side compaction, define the processing capability of this compaction subtask in the device-side system.

When all the compaction subtasks in the task queue finish, the D-Compaction calculates the averages value of the processing capability for all the compaction subtasks. Subsequently, the device-side system handles the processing capability draw value \(Perf_D = \frac{\sum_{i=1}^{M} D_{V_D}/T_D}{M}\), together with all the metadata of SSTables produced by the compaction as new information according to the interactive information format [packetHeader, packetLength, Perf_D, NewMetaDataSet, Checksum]). This information is sent to the host through the two-sided runtime system. The host-side system is responsible for the consistency management of metadata and the ratio of compaction processing capability between the host- and device-side subsystems.

3) Runtime System

The implementation of the device-side system largely depends on its runtime system, which has two functions: (1) Executing a set of redirection commands of storage access from the host-side system; (2) Transferring host-side compaction subtask information to the H-Compaction and the results of the device-side compaction.

(1) Execute redirection of storage access. The file operations in the host-side system need to be redirected to in the device-side system through the host-side runtime; then, the device-side runtime parses and executes the file operations. The results of the file operations are sent to the host-side system through the host interface.

The file operation redirection is the basis for the collaborative processing of compaction on both sides. Since the host- and device-side systems are independent subsystems, the data transmission on both sides should fulfill the data protocol format defined by the runtime and send it through the host interface. The database file is not stored on the host side. The file operations of the database are sent to the device-side system by means of rewritten POSIX API interfaces. Afterward, the device-side system parses and executes the corresponding file operations through the middleware, and returns the result to the host-side subsystem.

(2) Transferring the compaction information on the host side and the results of device-side compaction. When compaction is triggered on the host side, the device receives the compaction metadata from the host (e.g., level number, file metadata, ratios of processing capability between the host and device).

In the compaction on both sides, the dispatching thread of compaction for the host side fetches the compaction subtasks and configures interaction information by the format [packetHeader, packetLength, FinishFlag, SubtaskInfo, Checksum]). The data packet is sent to the host-side system through the host interface by the device-side runtime system.

When all the compaction subtasks in the task queue complete, the device-side system will send the metadata set of all SSTables generated by the compaction process and the value of compaction processing capability \(Perf_D\) for the device’s compaction task. The runtime system encapsulates the interactive information, i.e., [packetHeader, packetLength, Perf_D, NewMetaDataSet, Checksum] and sends it to the host-side system through the host interface. The host system completes the data consistency management of the storage system.

B. ON-DEMAND COMPACTION QUEUE

Note that when a compaction process in the DStore is triggered, an SSTable at level \(L_i\) has overlapping key ranges with the \(n\) number of SSTables at level \(L_{i+1}\). We suppose that the ratio of compaction processing capability is configured as \(X\).

The compaction is entirely performed on the device when \(n\) is no greater than \(X\). Otherwise, we have \(m = n\% (1 + X)\) and \(d = n/(1 + X)\), where \(m\) is the remaining number of SSTables when the compaction subtasks are finally offloaded. There are three types of values for the \(m\):

Type 1. \(m = 0\); Type 2. \(m = 1\); Type 3. \(1 < m < X + 1\);
FIGURE 4: On-Demand Compaction Queue Creation (FIGURE 4(a)) and Three types of Task Queue (FIGURE 4(b)). Note that there is $k = X + d$ and $j = n - X + 1$.

The total number of compaction subtasks is expressed as

$$t = 2 \cdot d + (m > 1) ? 2 : m$$

The challenge is how to build a compaction subtasks queue for the host- and device-side compaction. We construct a double-ended task queue.

When the first compaction is triggered, these subtasks in the compaction are dispatched to the host- and device-side compaction execution modules (i.e., H-Compaction and D-Compaction) in a ratio of 1:1.

In the compaction, the data volume of SSTables in the host and device ($DV_H$ and $DV_D$) and the completion time ($T_H$ and $T_D$) on both sides are recorded. By using $Perf = DV/T$, we can evaluate the processing capability of compaction tasks on both sides.

We have \(\frac{Perf_D}{Perf_H} = X\), meaning that the compaction processing capability of the device is $X$ times that of the host. When the following compaction is triggered, the value of $X$ is different from that at the initial stage. The compaction processing capability on both sides is also re-calculated as the average value of the processing capabilities of all the compaction subtasks, respectively. In the $K^{th}$ ($K > = 2$) compaction, the host-side system selects an SSTable at level $L_i$ (such as $SST_{0}$) as the participating SSTable and compares its key range of the $SST_{0}$ file with all ones at level $L_{i+1}$. It also finds all overlapped key ranges of $SST_{0}$, such as $SST_{1}$, $SST_{2}$, $SST_{3}$, ..., $SST_{n}$.

The compaction task queue is established by using the ratio of $1:X$ of the $n$ SSTables at level $L_{i+1}$, where “1” means that a single compaction subtask contains one SSTable and $X$ represents $X$ SSTables in a single compaction subtask.

The finding of the task queue that divides the SSTables into multiple sub-tasks according to $1:X$ (see FIGURE 4(a) and 4(b)) is according to the principle of from two ends to the middle. If the remaining number of unassigned SSTables is $M$ at the last split, the task queue format has three types:

**Type 1:** $M = 0$ (i.e., $n \% (1 + X) = 0$). The compaction task format can be expressed as (see (1) in FIGURE 4(b)):

$$T_1 : T_2 : T_3 : \cdots : T_{\frac{n}{1 + X}} : T_{\frac{n}{1 + X} + 1} : \cdots : T_{\frac{2n}{1 + X}}$$

**Type 2:** $M = 1$, the compaction task format can be listed as (see (2) in FIGURE 4(b)):

$$T_1 : T_2 : \cdots : T_{\frac{n}{1 + X} + 1} : T_{\frac{n}{1 + X} + 2} : \cdots : T_{\frac{2n}{1 + X} + 1}$$

$$T_i = \left\{ \begin{array}{ll} 1, & 1 \leq i \leq \frac{n}{1 + X} + 1 \\ X, & \frac{n}{1 + X} + 2 \leq i \leq \frac{2n}{1 + X} + 1 \end{array} \right.$$  \hspace{1em} (2)

**Type 3:** $1 < M < X + 1$, the compaction task queue format is (see (3) in FIGURE 4(b)):

$$T_1 : T_2 : \cdots : T_{\frac{n}{1 + X} + 1} : T_{\frac{n}{1 + X} + 2} : \cdots : T_{\frac{2n}{1 + X} + 2}$$

$$T_i = \left\{ \begin{array}{ll} 1, & 1 \leq i \leq \frac{n}{1 + X} + 1 \\ X, & \frac{n}{1 + X} + 2 \leq i \leq \frac{2n}{1 + X} + 2 \end{array} \right.$$  \hspace{1em} (3)

where let $T_i$ be the number of SSTables in the $i^{th}$ compaction subtask.

Meanwhile, when the task queue is established, the maximum key of all the SSTables in each compaction subtasks at level $L_{i+1}$ is used as a boundary to split the $SST_0$ SSTable at level $L_i$.

It assumes that $X = 2$, $n = 6$, and the task queue is $(1, 1, 2, 2)$, see FIGURE 5. Take the large key of $S_{0}$, $S_{1}$, $S_{3}$, and $S_{5}$ as the splitting boundary. The $S_{ki}$ is divided into four separated files and each one is bonded to the corresponding task blocks in the task queue. When a dispatching thread for the host- and device-side compaction fetches subtasks from the task queue, these four splitting files corresponding to the task blocks are integrated together and are sent to the compaction execution modules on both sides.

When the following compaction process is triggered again, the above processes are performed as well.

FIGURE 5: Dispatching threads for the host- and device-side subsystems.

**C. ON-DEMAND COMPACTION TASK SCHEDULING**

When the task queue is created, the device-side system generates two dispatching threads for host- and device-side...
compaction subtasks. These two threads exclusively take the compaction subtasks from the task queue and send them to the compaction processing modules on the host and the device.

The host- and device-side systems consider the task as the minimum execution unit of the compaction subtask, and one task may have one or more SSTables. After the completion on both sides, a new task from the task queue is selected to execute until the task queue is empty. In addition, the on-demand task scheduling varies with the platform.

(a) Homogeneous Platform for DStore

On the same platform, the computing and storage capability on the platform is fixed. Therefore, the ratio of compaction processing capability between the host and the device-side system tends to be stable. However, during the runtime, the resource consumption of the platform such as computation, storage, and bandwidth will change with the execution process in the compaction. Therefore, the average value of the compaction processing capability in the host and device may appear to be a variety. The ratio of the compaction processing capability (i.e., X) between the host and device will change in a certain interval.

To handle this issue, the DStore calculates the value of a compaction subtask processing capability in the host and the device, respectively. Then, the average processing capability during the compaction can be obtained. To realize a tactical task scheduling that dynamically adjusts the value of X when the usage of system resources changes between the host and the device in compaction, the DStore can optimally configure offloading tasks on both sides.

After creating the task queue of the compaction process according to the value of X, the host- and device-side task dispatching threads fetch the compaction subtasks and send them to the task execution modules on both sides, respectively.

Although the computing power of the host and equipment is stable, the latency time of a subtask processing may be fluctuated because of the CompactMemtable operations in host side and the resource occupations during the workload. We divide the following three cases according to the processing latency of subtask in the host- and device-side subsystems ($L_H$ and $L_D$). We analyze the processing of three kinds of task queues in each case.

(A) The latency time of the subtask on the host side is similar to that on the device side: $L_H = L_D$.

According to the definition of the compaction processing performance, the number of compaction subtasks in the host is equal to that in the device (see Equ. 1 and Equ. 3) because the compaction performance on both sides has the same values. However, there is the $(M-1)$ number of SSTables in the last compaction subtask as shown in Equ. 3.

(B) The latency time of the subtask on the host side is greater than that on the device side: $L_H > L_D$.

In the host-side compaction, the number of compaction subtasks acquired by the host-side dispatching thread is greater than that taken by the device-side dispatching thread.

In Equ. 1, there is at least one subtask with the X number of SSTables, which is sent to the host. In Equ. 2, there is a subtask with an SSTable and no less than zero subtasks with the X number of SSTables. There is a subtask having the $(M-1)$ number of SSTables and no less than zero subtasks including the X number of SSTables in Equ. 3.

(C) The latency time of subtask on host side is less than that on the device side: $L_H > L_D$.

In the device-side compaction, the number of subtasks acquired by the device-side dispatching thread is greater than that by the host-side thread in Equ. 1, Equ. 2, and Equ. 3. No less than one subtask with an SSTable is sent to the device-side compaction execution module. Note that the device dispatches a subtask with the $(M-1)$ number of SSTables to the device-side compaction in Equ. 3.

In the same platform, the on-demand compaction scheduling can dynamically adjust a variety of subtasks on both sides to effectively deal with the different performance of host- and device-side compaction tasks.

The compaction dispatching threads can select the optimal number of tasks to be processed on both sides according to compaction processing capability on both sides. This solution can effectively avoid the long latency in handling the subtasks and improves the overall time, which enhances the efficiency of the collaborating compaction process in the overall system.

(b) Heterogeneous Platform for DStore

In the same platform, the change in the values of X is relatively small and is mainly concentrated in a smaller interval in FIGURE 6. The X is mostly three for the OED (Open Ethernet Driver) platform [27] in most cases. However, due to different platforms, there are varying proportions of compaction offloading tasks because of the computing capability, the memory size, the interface bandwidth between the host and the device, the computing capability of the device, the size of the memory and the internal storage bandwidth in the device. For example, the ratio is mostly 7 in 1:4(S) (see Table 4). The on-demand scheduling can dynamically adjust the X value according to the compaction processing capability of the host and the device on different hardware platforms. Therefore, the overall system can achieve the optimal ratio of compaction processing capability, which is useful to improve the compaction performance on both sides.

![FIGURE 6: The ratio of compaction processing capability (i.e., X) under 20GB-fillrandom workload](image)
The on-demand scheduling method aims to optimize the performance of compaction using the NDP model. DStore allocates computing and storage resources on the NDP-enabled device to dynamically offload compaction subtasks to both sides according to their processing capability. This method reduces the host-side CPU and memory cost and reduces the I/O time of the host interface. The existing static task scheduling method cannot cope with task offloading on diverse hardware platforms, which can be solved by on-demand scheduling.

On a hardware platform, the on-demand scheduling can dynamically build task queue based on the compaction processing capability on both sides. Meanwhile, this method avoids much more compaction subtask offloaded into the host or the device, which leads to a large latency in task processing at one side of the system and cannot efficiently cooperate with compaction tasks. Instead, it reduces the performance of the overall system.

D. ON-DEMAND ALGORITHM IN DSTORE

In this section, we describe the on-demand compaction offload scheduling in DStore. The algorithm of the scheduling method in the host and device works as follows.

**Algorithm 1** Compaction Process on the Host Side

1: *procedure DOCOMPACTWORK_H*
2: X: the ratio of compaction processing capability
3: if TrigCompaction == TRUE then
4: if FirstDoCompaction == TRUE then
5: X=1
6: end if
7: SEND(Device, MetaDataSet, X)
8: vector<FileMetaData> HostMeta
9: while TRUE do
10: RECV(Device, Subtasks, FinishFlag)
11: if FinishFlag == TRUE then
12: break
13: end if
14: HostMeta ← CompactionSubtask()
15: end while
16: Compute the Perf_H
17: RECV(Device, Perf_D, NewMetaDataSet)
18: X ← Perf_H/Perf_D
19: Integrate MetaDataSet and Update Files
20: end if
21: *end procedure*

**Algorithm 2** Compaction Process on the Device Side

1: *procedure DOCOMPACTWORK_D*
2: RECV(Host, MetaDataSet, X)
3: dequeue<FileMetaData> Deque=CREATE(CompactionTaskQueue)
4: PARTITION_FILE(input_0_file)
5: Create two exclusive threads to get subtasks in Deque in opposite direction
6: CREATE_THREAD(Thread_H, Deque, SEND(Host, Subtasks))
7: CREATE THREAD(Thread_D, Deque, CompactionSubtask())
8: FinishFlag ← TRUE
9: SEND(Host, FinishFlag)
10: Compute the value of Perf_D
11: SEND(Host, Perf_D, NewMetaDataSet)
12: Destroy Thread_H and Thread_D
13: *end procedure*

(1) **Host-side Algorithm in DStore.** Based on LevelDB, we design on-demand scheduling modules (such as the decision, runtime system, and the compaction task cooperative processing) in DStore. The host-side system still uses strategy in LevelDB to handle requests from the upper applications and the judgment mechanism for the compaction. When the compaction is triggered, the original method that handles compaction by means of the resources within the host-side system is employed in LevelDB. In DStore, the relevant information of some compaction tasks is configured as a package according to the defined interactive information format by the runtime system. Through the host interface (Note that an ethernet interface is used as the host interface in this experiment), this compaction information based on these interactions is sent to the device-side system.

The device-side system creates and manages a compaction task queue according to the information and the ratio of the compaction processing capabilities (The ratio is one in the first compaction). The process of creating a task queue is shown in FIGURE 4(a). After the task being created, the device-side system generates two mutual threads. By the compaction processing capability on both sides, these two threads can fetch the compaction subtasks from the queue and integrates compaction information as a package through the runtime system; then, the packet is sent to the host.

In compaction, the host-side system continually receives the subtask information from the dispatching thread in the device. After the host-side system receives the package, it first parses the packet message according to the packet format (i.e., [packetHeader, packetLength, FinishFlag, TaskInfo, Checksum]) and makes a decision whether the received interactive information is a subtask or computation identification information for a compaction process.

If FinishFlag is false, it indicates that there are still remaining subtasks in the task queue. Then, the host-side subsystem performs the host-side compaction subtask and records the processing capability value for this compaction. Then, the host continues to receive the interaction information sent by the host-side dispatching thread.

If FinishFlag is true, which indicates that all compaction subtasks in the queue are completed, and the average processing capability is calculated; then the result of device-side compaction is obtained (such as metadata of the new SSTables and Perf_D). After that, metadata of the new SSTables and Perf_D is parsed from the packet message on device-side.

Last, the host system performs the data consistency management, such as integrating the of metadata sets on both sides, updating the log and inventory files, and deleting the related expired files. After completing the consistency management, it calculates the ratio of the compaction processing capability of both sides. When the next compaction process
is triggered, the ratio is sent to the device-side system. At this point, the two sides handle the compaction process, respectively.

(2) **Device-side Algorithm in DStore.** The runtime system in the device receives and processes the file-redirection operations from the host-side system in real time. It receives the compaction task information from the host-side system. When the runtime system receives this information, it immediately calculates the level numbers in the compaction, the metadata set of the SSTables, and the compaction processing ratio. The device-side system uses this information to establish the compaction task queue for this compaction process. According to the maximum key of each task in the queue, an SSTable of the upper level ($L_i$) in the compaction process is divided into a lot of subfiles corresponding to each task at level $L_{i+1}$, and each subfile is bound to the corresponding task as a compaction subtask. The device-side system creates two mutually exclusive threads, one of which fetches the compaction subtasks from the queue head (tail) and sends them to the host-side (device-side) compaction task processing module.

When the task processing module on the device side receives the subtask, it immediately completes the compaction process. Then, the value of the compaction processing capability is calculated and the device continues to send the request of subtask from the queue until all the sub-tasks in the task queue complete. When the tasks in the queue are completed, the value of $\text{Per}_{FD}$ is calculated, and the new metadata set of SSTables in this compaction is encapsulated into interactive information and sent to the host system.

The host-side system is responsible for metadata consistency management and calculates the ratio of the compaction processing capability.

Then, the compaction tasks on both sides cooperatively complete, but the device-side system still processes the operation of the database file from the host-side system by means of the runtime system, which aims to complete the operations from the upper-level application in the storage system.

### E. QUANTITATIVE ANALYSIS OF DSTORE

In this section, we conduct quantitative study of DStore's performance from two aspects, namely, compaction performance and write amplification in the compaction.

1) **Performance Analysis**

To facilitate the comparison of the performance relationships of LevelDB, Co-KV, and DStore, we assume that the ability of the host and device to handle the compaction task is a constant value of $\text{Per}_{FH}$ and $\text{Per}_{FD}$. Furthermore, we have $DV = \sum_{i=0}^{n} \text{Size}(\text{SST}_i)$, where $\text{SST}_0$ and $\text{SST}_i$ ($1 < i \leq n$) represents the SSTables in the compaction, and the size of SSTable can be defined as $\text{Size}(\text{SST}_i)$.

First, the compaction is done only on the host side for LevelDB, thus, the performance is expressed as:

$$T_{LevelDB} = \frac{DV}{\text{Per}_{LevelDB}} = \frac{DV}{\text{Per}_{FH}}$$

In Co-KV, the compaction task is divided into the device- and the host-side compaction tasks according to a fixed ratio. Therefore, the performance of the compaction task is determined by both the host- and the device-side performance because NDP aims to solve the data-intensive applications, and the latency of data processing from the device to the host is greater than the latency of processing data on the disk. Thus, we assume that $\text{Per}_{FD} \geq \text{Per}_{FH}$, that is $X = (\frac{\text{Per}_{FD}}{\text{Per}_{FH}}) \geq 1$; then, there is $\text{Per}_{FD} \geq \text{Per}_{FH}$.

$$T_{Co-KV} = \frac{DV}{\text{Per}_{Co-KV}}$$

$$= \max(DV/(2 \times \text{Per}_{FH}), DV/(2 \times \text{Per}_{FD}))$$

(5)

In a compaction, the host- and device-side performance runs in parallel, but the overall performance depends on the completion time of the submodule of the completion compaction task once completed. In DStore, the compaction process task is split into several compaction subtasks based on the compaction processing ratio. That is, the time to complete the compaction process is $T_{DStore}$. There are three possibilities for handling compaction performance on both sides during the process of the compaction subtask:

**(No.1)** When the latency time (processing speed) of compaction subtasks at both ends is the same ($L_H = L_D$): The amount of data in compaction subtasks on the host side and the device side is accurately divided according to the task ratio. The amount of data completed in the host is $DV/(1 + X)$, and the device has $(X \times DV)/(1 + X)$.

$$T_{DStore} = \frac{DV}{\text{Per}_{FD}}$$

$$= \max(\frac{DV}{(1 + X) \times \text{Per}_{FH}}, \frac{X \times DV}{(1 + X) \times \text{Per}_{FD}})$$

(6)

The value of $\text{Per}_{FH}$ and $\text{Per}_{FD}$ is consistent, and we set $X$ by the value of $\text{Per}_{FH}$ and $\text{Per}_{FD}$. Therefore, we have $DV/(1 + X)^2 \times \text{Per}_{FD} = (X \times DV)/(1 + X) \times \text{Per}_{FD}$ and then, $T_{DStore} = DV/(1 + X) \times \text{Per}_{FH}$

we have $(1 + X) \geq 2$, then,

$$T_{DStore} \leq T_{Co-KV} < T_{LevelDB}$$

**(No.2)** $L_H > L_D$

the database in host-side is $DV/(1 + X) - \delta$, because there is $X \geq 1, \delta > 0$, and then, $DV/(1 + X) - \delta < DV/2$

Since tasks in both sides are completed at the same time, So,

$$T_{DStore} = \frac{DV/(1 + X) - \delta}{\text{Per}_{FH}} < \frac{DV}{2 \times \text{Per}_{FH}} = T_{Co-KV}$$

(No.3) $L_H < L_D$

Data Volume in the Host-side: $DV/(1 + X) + \delta$.

Data Volume in the Device-side: $(X \times DV)/(1 + X) - \delta$

**(a)** $(X \times DV)/(1 + X) - \delta = DV/2$,

$$T_{DStore} = \frac{DV}{2 \times \text{Per}_{FD}}$$

**(b)** $(X \times DV)/(1 + X) - \delta > DV/2$,

$$DV/(1 + X) + \delta < DV/2,$$

$$T_{DStore} < T_{Co-KV}$$

**(c)** $(X \times DV)/(1 + X) - \delta < DV/2$,

$$T_{DStore} = \frac{DV/(1 + X) - \delta}{\text{Per}_{FD}} < \frac{DV}{(2 \times \text{Per}_{FH})} = T_{Co-KV}$$

Thus, we can obtain $T_{DStore} \leq T_{Co-KV} < T_{LevelDB}$.
The data volume of the SSTables involved in the triggering of the compaction is $DV$, and DStore completes the compaction with less time than Co-KV and LevelDB, so the write throughput of the DStore is better than that in Co-KV and LevelDB.

2) Write Amplification
We compare the three schedulings for the write amplification caused by compaction. Although both Co-KV and DStore have optimized write amplification in LevelDB, due to the different strategies, the optimization results are also different.

We assume that the upper level file participating in the compaction operation is $SST_0$ and the lower level files are $SST_1$, $SST_2$, $\cdots$, and $SST_n$. $WA_{LevelDB}$, $WA_{Co-KV}$, and $WA_{DStore}$ respectively indicating the write amplification of LevelDB, Co-KV, and DStore in compaction.

We set $DataSize$ as the total size of the SSTables in compaction. $DV = \sum_{i=0}^{n} Size(SST_i)$, where $SST_i$ is a file at level $L_i$ or $L_{i+1}$. $Size(SST_i)$ indicates the size of the $i^{th}$ SSTable. In Eq. 7, the write amplification of LevelDB should be the ratio of the size of the SSTable file in the compaction to the size of the $SST_0$ file. Co-KV directly offloads half of the compaction tasks to the device. The size of the SSTable file in the host-side system performs the compaction task actually involves: $DV/2$; therefore, the Co-KV-induced write amplification is performed by Eq. 8.

$$WA_{LevelDB} = \frac{DV}{Size(SST_0)}$$

(7)

$$WA_{Co-KV} = \frac{Data_{Host\_Co-KV}}{Size(SST_0)} = \frac{(DV/2)}{Size(SST_0)}$$

(8)

The on-demand scheduling in the DStore dynamically adjusts the value of $X$ based on the compaction processing capability of the host and the device. Similarly, the write amplification of DStore, meaning that the ratio of the total size of the SSTables in the host $(Data_{Host\_DStore})$ to the size of $SST_0$, can be expressed as

$$WA_{DStore} = \frac{Data_{Host\_DStore}}{Size(SST_0)}$$

(9)

The remainder number of SSTables is: $M = n \% (1 + X)$, where $X$ is the compaction processing capability ratio.

Suppose that $Perf_H$ is fixed and the maximum number of tasks which can be obtained by the dispatching thread is expressed as: $B = n/(1 + X) + M$

Let $Size(SST)$ have the same size.

$$WA_{DStore} = \sum_{i=0}^{n} \frac{Size(SST_i)}{Size(SST_0)} = B * \frac{Size(SST)}{Size(SST_0)} + B$$

(10)

$$WA_{Co-KV} = \frac{(DV/2)}{Size(SST_0)} = \frac{n}{2}$$

(11)

$$WA_{DStore} = \frac{[n/2]}{[n/(1 + X)] + M}$$

(12)

Because $X \geq 1, M \geq 1$ (see Type 1 and Type 2 in Section IV-B); then, when $X = 1$, we have,

$$WA_{DStore} (max) = \frac{n}{2} + 1$$

(13)

and

$$WA_{Co-KV} = \frac{n}{2} + 1$$

(14)

Then, $WA_{DStore} = WA_{Co-KV}$.

Finally, we have,

$$WA_{LevelDB} > WA_{Co-KV} \geq WA_{DStore}$$

Therefore, we can conclude that DStore outperforms LevelDB and Co-KV in the term of write amplification.

V. PERFORMANCE EVALUATION

We employ macro-benchmark and micro-benchmark to measure the performance of the on-demand scheduling in DStore. The experimental results of DStore, LevelDB, and Co-KV under this workload demonstrate the advantages of on-demand scheduling in DStore.

A. EXPERIMENT SETUP

To realistically simulate the NDP platform, our testbed is composed of host- and device-side subsystems. The host-side system runs on a computer configured with four core Intel Core(TM), i5-6500 CPU, 8GB DRAM. The operating system employs Ubuntu 16.04 LTS. The device-side system uses an ARM-based development board with six-core CPU, 4GB RAM, and 1TB SATA HDD as the storage device. Because two independent hardware systems do not have direct physical data transmission paths, the data exchange between the two systems must be set up through a runtime system to encapsulate and unpack the interaction information, and then transfer them through the host interface.

To ensure the credibility of the experiment, we set the basic parameters in the three key-value stores to the same value, such as 4MB-size MemTable, 2MB-size SSTable, and 4KB-size datablock. We chose sequential write load and random write load in a micro-benchmark to measure the three key-value stores. The key size is 16bytes, the value size is 1KB, and the tested data set is set to 10GB, 20GB, 30GB, and 40GB, respectively. In addition, we configure the YCSB benchmarking [28] tool to generate a comprehensive load to test the on-demand scheduling. YCSB is a widely used benchmark for performance evaluation of NoSQL databases. YCSB provides a default load type that tends to read-intensive workloads. However, the frequent compaction process is mostly triggered by the random-write load, which causes the performance degradation of the key-value store.

DStore aims to solve the problem of low compaction performance caused by write-intensive workloads. We configure three different load types in YCSB with varying database sizes, record sizes, and read/write ratios to compare the performance of on-demand scheduling of DStore, Co-KV, and LevelDB.

TABLE 1: DStore EVALUATION PLATFORM

<table>
<thead>
<tr>
<th>Hardware Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host CPU</td>
<td>Four Intel(R) Core(TM) i5-6500 @3.20GHz</td>
</tr>
<tr>
<td>Host DRAM</td>
<td>8GB DDR4</td>
</tr>
<tr>
<td>Protocol</td>
<td>DStore protocol based on TCP/IP</td>
</tr>
<tr>
<td>NDP Processors</td>
<td>Two Cortex A72 and Four Cortex A53</td>
</tr>
<tr>
<td>NDP DRAM</td>
<td>4GB DDR3</td>
</tr>
<tr>
<td>NDP Storage Medium</td>
<td>1TB Western Digital SATA HDD, 126MB/s(read) and 115MB/s(write)</td>
</tr>
<tr>
<td>Host-and-Device Interface</td>
<td>Gigabit Ethernet Interface, 16MB/s</td>
</tr>
</tbody>
</table>

To ensure the credibility of the experiment, we set the basic parameters in the three key-value stores to the same value, such as 4MB-size MemTable, 2MB-size SSTable, and 4KB-size datablock. We chose sequential write load and random write load in a micro-benchmark to measure the three key-value stores. The key size is 16bytes, the value size is 1KB, and the tested data set is set to 10GB, 20GB, 30GB, and 40GB, respectively. In addition, we configure the YCSB benchmarking [28] tool to generate a comprehensive load to test the on-demand scheduling. YCSB is a widely used benchmark for performance evaluation of NoSQL databases. YCSB provides a default load type that tends to read-intensive workloads. However, the frequent compaction process is mostly triggered by the random-write load, which causes the performance degradation of the key-value store.

DStore aims to solve the problem of low compaction performance caused by write-intensive workloads. We configure three different load types in YCSB with varying database sizes, record sizes, and read/write ratios to compare the performance of on-demand scheduling of DStore, Co-KV, and LevelDB.
TABLE 2: Parameters in DStore, Co-KV, and LevelDB

<table>
<thead>
<tr>
<th>Type</th>
<th>MemTable</th>
<th>SSTable</th>
<th>Data Block</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>4MB</td>
<td>2MB</td>
<td>4KB</td>
<td>16B</td>
<td>1KB</td>
</tr>
</tbody>
</table>

TABLE 3: Workload Characteristics

<table>
<thead>
<tr>
<th>db_bench - Workload in LevelDB</th>
<th>Type</th>
<th>Key-Value Items (10,000,000)</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>db_bench_1</td>
<td>fillrandom</td>
<td>1x, 2x, 3x, 4x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>db_bench_2</td>
<td>fillseq</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B. MICRO-BENCHMARK EVALUATION

We use db_bench to test the throughput, write amplification, and execution time of DStore, Co-KV, and LevelDB in the random-write workload db_bench_1 and sequential-write workload db_bench_2.

1) Throughput (MB/s and micros/op)

The time consumption (for example micro-seconds) per each operation is recorded as well as the throughput. The much lower the value is, the higher the throughput of the key-value store.

In LevelDB, the compaction is placed on the host side and consumes the computing resources and bandwidth on the host side, which makes the requests latency for upper-level application and reduces the system throughput. In FIGURE 7(a) and 7(b), the throughput decreases as the size of the data sets increases, while the DStore with on-demand scheduling can create a compaction task queue based on the real-time processing capability of compaction on both sides.

The host- and device-side dispatching thread fetches the compaction subtasks from the task queue according to the compaction processing capability. This process of subtask execution on both sides is collaboratively performed dynamically. DStore fully utilizes the computing capability on both sides and the device-side internal storage bandwidth. Thus, the completion time for the compaction shortens, and the response time for the upper level application achieves improvement. In FIGURE 7(a) and 7(b), the throughput in DStore is higher than that in LevelDB. The throughput in DStore reaches the maximum performance when the size of data set is 10GB, i.e., 0.7MB/s, 1479micros/op. Comparing with LevelDB, the throughput has 3.5x MB/s and 3.7x micros/op improvement.

Similar to DStore, Co-KV performs compaction in parallel between the host and the device. Co-KV uses a static offloading ratio (0.5) to partition the compaction subtasks into the host- and device-side systems. It also optimizes the computing and storage in the device to achieve the process compaction task in parallel. Therefore, it reduces the duration of the compaction and improves the throughput of the overall system. In FIGURE 7(a) and 7(b), Co-KV has better throughput than that in LevelDB under a random-write workload. When the tested data set reaches 10GB, the Co-KV throughput achieves the highest value, which is 1.9x (micros/op) and 2.3x (MB/s) throughput improvement compared to LevelDB.

Due to the increment of ARM-based processors and low cost, the computing capability and storage bandwidth of a device can effectively deal with the offloading tasks in the host-side system. However, there may be many more compaction subtasks being offloaded on the host, which can cause many CompactMemTable operations and cause the host to consume much time to complete the compaction subtasks. Then, the compaction processing efficiency of the storage system is reduced. The throughput in DStore is much higher than that in Co-KV.

The random-write workload (i.e., db_bench_1) frequently triggers the compaction in LSM-tree based key-value stores. In sequential-write workloads, because the key-value is put in the device in an orderly manner and the key-value store does not have SSTables with overlapping key ranges, the compaction is rarely triggered such as in db_bench_2. The goal of on-demand scheduling based DStore and Co-KV to improve the compaction performance in LevelDB. The on-demand and Co-KV optimized scheduling will not be invoked under a sequential write workload which cannot trigger a compaction process. In FIGURE 7(a) and 7(b), the throughput of DStore, Co-KV, and LevelDB has approximately the same values.

2) Write Amplification

The random-write workload frequently triggers the compaction process and enlarges the write amplification, thereby reducing the performance of the key-value store. Write amplification is a key metric affecting the performance of the key-value store.

Since the sequential-write workload does not trigger the compaction in the key-value store, write amplification in the three key-value stores is mainly caused by the update of the write head log and the manifest files; meanwhile, the offloading scheduling for the compaction optimization module in the on-demand based DStore and Co-KV is not called. Therefore, these three key-value stores have the same write amplification. In FIGURE 7(c), under sequential-write workloads, the write amplification value in these key-value stores is approximately two.

However, the key-value store performs compaction frequently under random-write workloads. With the increment of the testing data set, the frequency of compaction becomes higher. In FIGURE 7(c), write amplification in LevelDB, Co-KV, and DStore shows a gradual upward trend. The reason is that the increment of the data volume triggers more compaction; thus, write amplification becomes higher. When the size of data set reaches 40GB, the write amplification in DStore, Co-KV, and LevelDB reaches 2.9, 8.9, and 15.4, respectively. In FIGURE 7(c), DStore has a much smaller write...
amplification than Co-KV under random-write workloads, which is also validated by our quantitative analysis.

3) Execution Time under Workloads

In this section, we study the execution time of three key-value stores on the host-side system under db_bench because DStore allows the compaction subtasks from the queue to be dispatched to the host and device according to various compaction processing capabilities. The partial tasks in the compaction process are offloaded into the device-side system. It reduces the amount of compaction tasks performed by the host-side system; thereby, the computation overload on the host-side system is reduced and the execution time of the host-side system is effectively reduced.

In FIGURE 8, the execution time in DStore is less than that in LevelDB and Co-KV under random-write workload. When the tested data set is 40GB, DStore decreases the execution time by 24.1% compared to LevelDB, and presents 16.3% lower than that in Co-KV. Because Co-KV uses static scheduling to offload half of the compaction task device-side system, it reduces the amount of compaction tasks on the host-side system. The execution time for the operation of SSTables and merge sorts is reduced. Therefore, the execution time in Co-KV is slightly lower than that in LevelDB.

The key values generated by the sequential-write workload are ordered. Therefore, there is no compaction triggered under sequential-write workloads; thus, the host-side execution times of LevelDB, Co-KV, and DStore are almost the same.

C. YCSB

1) Database Size

In this experiment, we set the workload with 100%-writes and 1KB-record size to test the performance impact of different database sizes on DStore, Co-KV, and LevelDB.

We first use the "Load" command to create a 20GB dataset, and then use the "Run" command to perform this workload. In FIGURE 9(a), the throughput of the three key-value stores gradually decreases as the dataset set increases. When the dataset is set to 20GB, the performance of DStore, Co-KV, and LevelDB, reaches (minimum) 813 ops/sec, 336 ops/sec, and 186 ops/sec, respectively.

The 100%-write workload in the Zipf distribution is a random operation; the frequency of compaction significantly increases as the size of dataset becomes much bigger. At the same time, the read/write operations in the merge sort for SSTables consume much more host-side computing and bandwidth, which delays the response time of the request in the upper-level application. Thereby, the overall system throughput reduces.

Write amplification in LevelDB increases because the increment of dataset causes more compactions. When the testing dataset reaches 20GB, the write amplification of LevelDB is 11.4. DStore and Co-KV offload a portion of host-side compaction tasks to the device, which reduces the amount of data movement between the host and the device. Therefore, the write amplification in DStore and Co-KV is much smaller than that in LevelDB. DStore dynamically offloads compaction subtasks to both sides; therefore, the on-demand scheduling-based DStore improves the system performance compared to that in Co-KV with static scheduling.

In the condition of a fixed amount of data volume, the throughput is inversely proportional to the latency. In FIGURE 9(c), the update latency gradually increases when the size of the dataset increases. Under 100%-random-write workloads, the compaction is much frequently triggered as the size of tested dataset increases. The compaction increases the latency of the requests from the upper level application. Since DStore and Co-KV perform the compaction in parallel and shorten its duration, the latency in these two key-value stores is lower than that in LevelDB. However, there are many more compaction subtasks offloaded to the host in Co-KV, which causes much higher latency for compaction in the host
than that in the device. Then its update latency is higher than that in DStore.

2) Record Size

Different record size in workload affects the read and writes performance. Therefore, it is extremely necessary to study the effect of different record sizes on the performance of DStore, Co-KV, and LevelDB. In this set of experiments, the parameters of the workload are 100% write, Zipf distribution, and four types of record size (i.e., 1KB, 4KB, 16KB, and 64KB). The 10GB dataset is inserted into key-value stores by "Load", and "Run" 10GB workload on the system.

In FIGURE 10(a), the throughput in (ops/sec) of DStore, Co-KV, and LevelDB drops sharply with the increment of record size. When the record size is 1KB, the throughput of DStore, Co-KV, and LevelDB is 786ops/sec, 376ops/sec and 198ops/sec, respectively. When the record size increases to 64KB, the throughput of the three key-value stores reaches (minimum) 18ops/sec, 13ops/sec, and 10ops/sec, respectively. The increment of the record size results in a longer latency. The completion time compaction on the three key-value stores increases and the throughput decreases. DStore and Co-KV make a full utilization on the resources of the device to reduce data movement from the host and shorten the completion time of compaction. DStore and Co-KV have a better performance than LevelDB. Compared with Co-KV, DStore avoids offloading half of the compaction subtasks to the host as the record size increases. The high-overhead data transmission to the host is reduced. Therefore, DStore outperforms Co-KV on throughput significantly.

In FIGURE 10(b), the write amplification increases with the increment of record size. When the record size is 1KB, the write amplification in LevelDB is 9.9. When the record size is 64KB, the value of write amplification increases to around 12.1 in LevelDB. Co-KV offloads half of the compaction tasks to the device, and the amount of data transmitted to the host system reduces; then, its write amplification decreases. When the record size is 64KB, the write amplification in Co-KV is 6.8, which has a 43% reduction compared to that in LevelDB. When the record size increases, the latency for the host-side SSTable increases and the device has the ability to effectively handle the compaction tasks. A large number of compaction tasks are completed on the device and the host-side load is reduced. Therefore, write amplification on DStore drops by 62% compared to Co-KV when the record size is 64KB.

From the above analysis, it can be concluded that a bigger record size leads to an increase in the delay of SSTables read and write; thereby, increasing the overhead of the compaction processing in the overall system. In FIGURE 10(c), the update latency is much worse with bigger record size. When the record size is 64KB, the update latencies of DStore, Co-KV, and LevelDB reach (maximum) 54,827 us, 76,727 us, and 99,280 us, respectively. Based on the near data processing model, DStore employs on-demand scheduling avoid offloading more compaction tasks into the host at one time, which realizes the dynamically cooperative compaction process in the host and device. Therefore, DStore has a lower update latency advantage than that in other ones.

3) Write Ratio

In this section we evaluate the impact of the different ratios of write operations on the performance. We load a 20GB test dataset, configure record size as 1KB, and employ Zipf distribution. There are four write ratios in the workload, i.e., 10%, 40%, 70%, and 90%, respectively. DStore and Co-KV are based on the NDP model to design different scheduling for compaction tasks. Both of them do not consider the optimization of read operations. The process of their read operations is the same as that in LevelDB. When the upper application initiates a read request, the MemTable in memory is first searched. If the MemTable hits, the result is returned, else it will search the metadata in each level to find the targeted file; meanwhile, the number of unnecessary I/O operations can reduce by means of Bloomfilter.

In experiments, the file operation in the host should be redirected to the device side by the two-side runtime systems. These file operations are handled by the execution module in the device. Therefore, the read operations consume much time than batch write operations, that is, meaning that read throughput is lower than write. In FIGURE 11, when the write ratio is 90%, the throughputs of DStore, Co-KV, and LevelDB are 486 ops/sec, 288 ops/sec, and 170 ops/sec, respectively. However, as the write ratio of the workload reduces, the throughput of the three key-value stores gradually decline. When the ratio reduces to 10%, the throughput of the
VI. SENSITIVITY STUDY

In the sensitivity study, we perform experiments to validate the availability of on-demand scheduling in DStore. This test aims to find high-throughput performance and low write amplification of DStore in the condition of different ratios of the internal and external bandwidth in NDP-based devices, different computing capability, and a real-world NDP platform.

There are two important types of bandwidth in the NDP-based device: (1) the external bandwidth between the host and the NDP-based device; (2) the internal bandwidth between the memory and the storage medium in the NDP-based device. In Section V-A, our NDP-based platform is implemented based on a development board with an embedded storage device (e.g., HDD and SSD) which emulates the storage medium in a real-world NDP platform. As listed in Table 4, we employ four types of embedded storage devices to configure different internal bandwidths between the memory and storage medium in the NDP-based device.

1) Different internal and the external bandwidths in the NDP-based storage system. Two ratios between the internal bandwidth and external bandwidth are employed in our test, such as 1:2 and 1:4. The difference between 1:2(L) and 1:2(S) is lies on the storage capacity. Taking 1:2(L) case for example, the external bandwidth between the host and the device is about 16MB/s while the optimal internal bandwidth between the device-side memory and its embedded storage medium is 34MB/s. We can see that the ratio of the external and internal bandwidth is around 1:2. Meanwhile, 80GB-HDD-based embedded storage is featured with a large capacity compared to the small 32GB-SSD-based storage medium with the same ratio (1:2) which is defined as 1:2(S).

2) Different number of cores simulates various computing capability of the NDP-based device.

To realize the different computing capability, we respectively deployed four cores and two cores on the platform to run workload, which is targeted at evaluating the effect of various computing capability on system performance of the tested.

3) Using the Open Ethernet Driver, a near-data processing hardware platform.

A real-world NDP-based device is employed in this measurement to study the availability of DStore.

To compare the results of the three key-value stores (LevelDB, Co-KV, and DStore) under YCSB, we performed experiments to validate the availability of on-demand scheduling in DStore. This test aims to find high-throughput performance and low write amplification of DStore in the condition of different ratios of the internal and external bandwidth in NDP-based devices, different computing capability, and a real-world NDP platform.

<table>
<thead>
<tr>
<th>Types</th>
<th>IB : EB</th>
<th>Embedded Storage Devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:2(L)</td>
<td>1:2</td>
<td>80GB SATA HDD, 35MB/s(R) and 35MB/s(W)</td>
</tr>
<tr>
<td>1:4(L)</td>
<td>1:4</td>
<td>25GB SATA HDD, 68MB/s(R) and 68MB/s(W)</td>
</tr>
<tr>
<td>1:2(S)</td>
<td>1:2</td>
<td>32GB SATA SSD, 125MB/s(R) and 38MB/s(W)</td>
</tr>
<tr>
<td>1:4(S)</td>
<td>1:4</td>
<td>64GB SSD, 126MB/s(R) and 66MB/s(W)</td>
</tr>
<tr>
<td>1:7(L)</td>
<td>1:7</td>
<td>1TB SATA HDD, 126MB/s(R) and 115MB/s(W)</td>
</tr>
</tbody>
</table>

Computation power

<table>
<thead>
<tr>
<th>Number of cores</th>
<th>Four Cores and Two Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>OED (Open Ethernet Driver) Platform [27]</td>
<td>Debian 8.7(jessie)</td>
</tr>
<tr>
<td>OS</td>
<td>CPU</td>
</tr>
<tr>
<td>Debian 8.7(jessie)</td>
<td>32bit ARM, 512KB Level 2 Cache</td>
</tr>
</tbody>
</table>
eDB, Co-KV, and DStore), we run db_bench_3 and ycsb_3 in three experimental cases.

In db_bench_3, we use the db_bench in LevelDB and set the workload with fillrandom; the size of the tested data set is 20GB, the size of key-value items, MemTable, SSTable and data block are listed in Table 2. In ycsb_3, the requests in YCSB meets Zipf distribution. The ratio of write is 80%, record size is 4KB, load and run 2.5 million operations.

(C.1) Different Ratios of Internal and External Bandwidth

We test different ratios of the internal to external bandwidth on an NDP-based device, see Table 4.

If the read and write bandwidth decreases, the latency of read/write in compaction on the host and the device side increases. Then, the throughput of the storage system reduces. In FIGURE 12(a) and FIGURE 12(b), it can be seen that the throughput of the LevelDB, Co-KV and DStore gradually increases as the transmission bandwidth of the storage medium raises and the completion time of each operation (micros/op) gradually decreases under db_bench_3. In addition, we can also clearly see that DStore has performance advantages over that in LevelDB and Co-KV (see FIGURE 12(a) and FIGURE 12(b)). When the testbed is featured with 1:2(S), the throughput of LevelDB, Co-KV, and DStore is 0.1MB/s, 0.2MB/s, and 0.4MB/s, respectively. DStore has better performance than the other ones, mainly due to the dynamical on-demand task offloading scheduling.

The change of bandwidth between the host and the device has no effect on write amplification. In FIGURE 12(c), the write amplification in LevelDB keeps the same value, i.e., 13.6, which is independent of the internal and external bandwidth. Co-KV uses a static compaction offloading scheduling and its write amplification is not affected by varying the bandwidth, remaining at about 7.4. However, for the dynamic scheduling in DStore, the bandwidth between the internal and external interfaces affects the capacity of the device to run the compaction task. The higher the internal read/write bandwidth, the higher the capability of performing compaction on the device. The on-demand compaction scheduling offloads more compaction tasks to the device, reducing the number of compaction subtasks on the host. Due to different bandwidth on the two sides, write amplification presents minor fluctuations as shown in FIGURE 12(c), and the value is much better than that in LevelDB and Co-KV.

Under the ycsb_3, we configure different read/write ratios in YCSB to test the impact of different internal and external bandwidth on throughput. In FIGURE 12(d), when the internal bandwidth is small (i.e., 1:2(S)), the throughput of LevelDB, Co-KV, and DStore reaches (minimum) 630ops/sec, 1130ops/sec, and 1490ops/sec, respectively. With the bandwidth increases, the throughput of the three key-value stores gradually increases. Although these three key-value stores have the same operation flow, the DStore optimizes the compaction caused by random-write operation and improves write throughput. Therefore, DStore has better throughput than that in the other two key-value stores. For example, the throughput of the three parties is 810ops/sec, 1610ops/sec, and 2320ops/sec when the storage medium is 1:4(L).

(C.2) Different Computing Power

The computing capability of the NDP-based device determines the ability to encapsulate and unpack in the runtime system and handles the compaction subtasks in the device. We used the sched_setaffinity() provided by Linux to bind the program to four cores or two cores to measure the performance of LevelDB, Co-KV, and DStore with different computing capability. Because the reduction of the number of cores increases the latency of encapsulation in the runtime system and completion of compaction subtasks in the NDP device, the throughput of the three key-value stores reduces. In FIGURE 13(a) and 13(b), the performance of the three key-values gradually decreases with the reduction of cores, but the impact on performance is small. For example, the throughputs of the three systems with 4 cores, the performance in LevelDB, Co-KV, and DStore is 6240 micros/op, 3400 micros/op, and 1751 micros/op, respectively. When there are two cores on the device, the throughput of the three storage systems is 6400 micros/op, 3566 micros/op, and 1820 micros/op, respectively.

Due to the reduction of cores, the change of an NDP device’s computing capability will not influence the write amplifications caused by the compaction. Therefore, the write amplification of LevelDB and Co-KV keep the same value 13.6 and 7.4, respectively, see FIGURE 13(c). In DStore, there are two cores on the device, but it can still efficiently complete the assigned compaction tasks; thus, write amplification remains 2.6 under this workload.

In FIGURE 13(d), we use the ycsb_3 to test the effect of changes in the computational capability of the NDP device on performance under the workload with different write ratios. When the system is bound to four cores, the throughput of LevelDB, Co-KV, and DStore is 740ops/sec, 1350ops/sec, and 1850ops/sec. When the number of cores reduces to two, the throughput decreases by 17.5%, 6.6%, and 5.9% for the three key-value stores, respectively. The smaller number of cores leads to an increment in the latency of processing host-device interaction information and the compaction subtask completion on the device side; therefore the throughput of the three devices reduces. However, DStore can dynamically adjust the offloading scheduling based on the compaction processing capability on both sides. Therefore, the throughput of DStore has a greater improvement over other ones.

(C.3) OED (Open Ethernet Driver) Platform

The OED platform (see Table 4) realizes the computing data near the devices. The platform is configured with a single-core Armv7 processor and 2 GB DRAM. This platform is used to validate the usability of our scheduling and the feasibility of applying it to an NDP-based device.

When the window size of TCP is 8K, the network transmission bandwidth between the OED and the host-side system is 29MB/s, which is 1.8x the network interface transmission bandwidth in Section V. Under db_bench_3, it can be seen that the throughput of LevelDB, Co-KV, and DStore is
much higher than in FIGURE 14(a) and 14(b). This experimental result combined with the tested results under different cores proves that the network interface bandwidth rather than the CPU inside the device limits the performance of the storage system in our existing test platform. Co-KV and DStore offloads a portion of compaction tasks to the device, enabling it to perform compaction tasks in parallel. Therefore, the throughput of both is better than LevelDB. In FIGURE 14(a), on an OED platform under db_bench_3, LevelDB, Co-KV, and DStore have throughputs of 0.4MB/s, 0.6MB/s and 0.8MB/s, respectively. The Co-KV statically offloads half of the compaction tasks to the OED device which has a greater ability to complete the compaction task than that in the host-side system. Therefore, the host-side system cannot complete the task in time, which increases the latency of the parallel compaction subtasks. DStore can dynamically adjust the X value according to the compaction processing capability on both sides (e.g., the X values concentrate around three). In compaction, the two-side task dispatching thread implements the on-demand initiative obtaining the compaction subtask and achieves the best parallel task processing performance. The reason that LSM-tree write amplification has been analyzed above under different bandwidths is the write amplification of the host-side system is not affected by the hardware platform. Therefore, the write amplification of LevelDB and Co-KV remains unchanged in the same under db_bench_3, and the write amplification of both remains 13.6 and 7.4. At this time, the write amplification of DStore is 4.3, which is higher than that of the basic platform, because the ratio of the host-side compaction processing capability is less than that in Section V. Therefore, more tasks are offloaded to the host-side system, when processing compaction tasks.

Although the read process in the three key-value stores is the same, DStore and Co-KV optimize the compaction caused by write operation. The time to process write operations improves, and the system throughput enhances. Under ycsb_3, DStore shows better throughput, 252ops/sec, 218ops/sec, and 175ops/sec in DStore, Co-KV, and LevelDB, respectively.

VII. RELATED WORK

In this section, we describe related work on LSM-tree based key value store and near-data processing model.

A. LSM-TREE BASED KEY-VALUE STORES

Due to high-write-performance, the log-structured merge-tree (LSM-tree) is widely used in most popular storage systems (e.g., BigTable, HBase, LevelDB, Apache Cassandra [29]) for big-data applications. Recent work focuses on LSM-trees optimization for the compaction process, e.g., LSM-trie [9], WiscKey [30], ForestDB [31], SkipStore [32], DCompaction [33], bLSM [34], cLSM [35], VT-tree [36], FD-trees [37] etc.

Therefore, researchers mostly propose methods to shorten the I/O latency and improve the throughput of LSM-based key-value stores. TRIAD [38] optimizes key-value stores by separating hot and cold keys in memory level. The duplicate write I/Os in the commit log reduces because an optimized compaction process is designed in the storage level. A lightweight compaction tree is used in LWC-store [39] to reduce write amplification. The PebblesDB [40] is proposed to com-
bine the LSM-tree with a skip list, which presents the concept of guards to improve the compaction process. FlashKV [41] eliminates the redundant operations by employing the open-channel SSD. With this platform, FlashKV employs a parallel data layout to exploit the internal parallelism in flash memory devices. Three key technologies (i.e., double compaction threads, a priority-based scheduler, and a compaction-aware cache) are employed to improve the performance of key-value stores. Another LSM-tree based key-value store an on open-channel SSD, LOCs [42] optimizes scheduling and dispatching policies for I/O requests. NVMKV [43] a hashing-based key-value store uses flash translation layer to reduce write amplification and enhance the performance of a key-value store in flash memory. LSBm [44] manages a buffer in the disk to minimize cache access miss caused by compaction in the device. Atlas [45] is a distributed key-value store to handle keys and values on different hard drives.

B. STORAGE-LEVEL NEAR-DATA PROCESSING

In the big data era, large-scale data analysis is increasingly important in data-intensive applications to extract features of data trend, data pattern, and computation models [46]. As the scale of data volume increases, more data moves to the host-side memory from the device-side storage system. When the data analysis completes, the data from store is discarded. Thus, there is a high cost of data movement in data-intensive applications.

Researchers propose a framework that focuses on data processing in the storage level [18] [17] [47] or the memory level [48] [49] to decrease a large amount of data movement and improve performance, such as active disks [50] [51] [52] [53] [54]. In this framework, a portion of an application is conducted on a disk with data-processing capability. In the previous stage of this idea, the prototypes, this framework including programming models, and algorithms has been widely studied but rarely applied to applications, which results from the manufacturing complexity or low power for the embedded computing component. This framework has attracted interest from industry and academics as a response to the trend of powerful embedded computing component (e.g., ARM or FPGA) [55] [56] in devices. Near-data processing [18] or in-storage computing [19] inherits the idea of the active disk. The storage device is treated as a distributed computing platform, in which the embedded processor performs partial program codes inside the storage-level devices. During data processing, it places computing in the data source rather than moving data from the device to the host-side memory.

For ISC model, data which moves to its internal DRAM from flash memory is conducting operations. Once it finishes, the result, the size of which is smaller than rare data, is transmitted to the host-side user applications. This new computing paradigm reduces the cost of data movement for current big-data applications. Our work aims to fully grasp the benefits of the NDP model and its use for compaction improvement in LSM-tree based key-value stores.

In compaction, SSTables with overlapping key ranges moves to the host-side memory from the device-side storage. They perform merge-sort operations. The overlapped-key-range SSTables are sorted into new ones with no-overlapping key ranges and store it in the device.

We find that a compaction process can be divided into sub-steps, a portion of which can be offloaded into the device. Then the compaction is performed on the host and device sides in parallel by using the power of computation resources in the device. Considering the tasks offloading for compaction, the critical challenge is to divide compaction tasks into the two-side subsystem at first. Second, we must confirm how many tasks can be offloaded into the host and the device during the compaction process. In addition, different computing power exists on both sides. The time consumption of compaction tasks on both sides varies. Thus, it motivates us to propose an on-demand computation resource-based compaction optimization scheduling for an LSM-tree based key-value store (i.e., DStore) to optimize compaction by means of the near-data processing model. In this paper, our discussion mainly focuses on the compaction processing capability-aware task division and offloading for LSM-tree based key-value stores.

VIII. CONCLUSION

In LSM-tree based key-value stores, compaction has an adverse effect on the system performance under random-write workloads. Current approaches mostly focus on compaction performance improvement on the host side. In this manuscript, we propose a holistic key-value store named DStore to explore near-data processing and on-demand scheduling for compaction optimization in an LSM-tree key-
value store. In DStore, the device for data storage is featured with computing capability. DStore aims to use the near-data processing model in the device and takes full advantage of the collaborative computation on both the host and device. In NDP-based storage systems, the compaction processing capability varies with the host- and device-side subsystems. To make full use of the both-side computing resources, we design an on-demand scheduling for dynamically partitioning the compaction tasks in the process of compaction and offloading these tasks into the above two-side subsystems by the means of their various compaction processing capability. Then, DStore performs these compaction tasks in parallel on the host and the device sides. In DStore, the NDP-based devices exhibit low-latency and high-bandwidth performance to facilitate key-value stores. DStore cannot only accomplish compaction for LSM-tree based key-value stores but also improve the system performance.

DStore prototype is implemented on a real-world platform and different types of testbeds are employed in our experiment. Extensive results show that the on-demand compaction optimization for the LSM-tree key-value store can dynamically partition compaction tasks and offload them into the host and the device side on the NDP-based platform. In additional DStore outperforms the two scheduling, i.e., LevelDB and static scheduling for partitioning and offloading for compaction tasks. DStore can significantly improve the LSM-tree based key-value storage performance by substantially optimizing the compaction operations in the background.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their work on our paper. This work was sponsored in part by National Natural Science Foundation of China under Grants 61702004 and 61572209, Anhui University Funding for Doctoral Research (J01003214), the National Science Foundation of Anhui Province (1708085QF160), and Key Technology R&D Program of Anhui Province (1704d0802193), 2017 Opening Foundation of Key Laboratory of Intelligent Computing & Signal Processing, Ministry of Education (Anhui University). This work was also supported in part by an U. S. National Science Foundation grant CCF-1563750.

REFERENCES

Hui Sun, received the PhD degree from Huazhong University Science and Technology, in 2012. He is an assistant Professor of computer science with Anhui University. His research interests include computer systems, edge computing, performance evaluation, Non-Volatile Memory-based storage systems, file systems, and I/O architectures.

Wei Liu, born in 1993. He is currently pursuing the M.S. degree in Anhui University. His main research interests include computer systems, edge computing, and key-value storage systems.

Zhi Qiao is a PhD candidate in the Department of Computer Science and Engineering at the University of North Texas. His research interests are Intelligent Storage Devices, File System Resilience and Edge Computing for Augmented Reality. He has been a member of the IEEE and ACM community since 2012.

Song Fu, is an Associate Professor and the Associate Chair for computer engineering at the University of North Texas. He received his PhD from Wayne State University in 2008. His research interests include high-performance computing, distributed and cloud systems, including architecture, resilience, dependability assurance, and resource management. He is an IEEE Senior Member.

Weisong Shi, is a Charles H. Gershenson Distinguished Faculty Fellow and a professor of Computer Science at Wayne State University. His research interests include Edge Computing, Computer Systems, energy-efficiency, and wireless health. He received his BS from Xidian University in 1995, and PhD from the Chinese Academy of Sciences in 2000, both in Computer Engineering. He is a recipient of the National Outstanding PhD dissertation award of China and the NSF CAREER award. He is an IEEE Fellow.