FPGA-based Compaction Engine for Accelerating LSM-tree Key-Value Stores

Xuan Sun, Jinghuan Yu, Zimeng Zhou and Chun Jason Xue
Department of Computer Science
City University of Hong Kong
{xuansun-c, jinghuayu2-c, zmzhou4-c}@my.cityu.edu.hk, jasonxue@cityu.edu.hk

***Abstract*—With the rapid growth of big data, LSM-tree based key-value stores are widely applied due to its high efficiency in write performance. Compaction plays a critical role in LSM-tree, which merges old data and could significantly reduce the overall throughput of the whole system especially for write-intensive workloads. Hardware acceleration for database is a popular trend in recent years. In this paper, we design and implement an FPGA-based compaction engine to accelerate compaction in LSM-tree based key-value stores. To take full advantage of the pipeline mechanism on FPGA, the key-value separation and index-data block separation strategies are proposed. In order to improve the compaction performance, the bandwidth of FPGA-chip is fully utilized. In addition, the proposed acceleration engine is integrated with a classic LSM-tree based store without modifications on the original storage format. The experimental results demonstrate that the proposed FPGA-based compaction engine can achieve up to 92.0x acceleration ratio compared with CPU baseline, and achieve up to 6.4x improvement on the throughput of random writes.**

***Index Terms*—compaction, LSM-tree, key-value, FPGA**

I. Introduction

In the era of big data, Log-Structured-Merge-tree (LSM-tree) based key-value stores are widely applied in NoSQL systems, such as Cassandra [1], BigTable [2], HBase [3], and RocksDB [4]. In comparison with RDBMS (Relational Database Management System), LSM-tree based key-value stores show better performance for applications with write-intensive workloads. The traditional RDBMS is more suitable for workloads dominated by read requests. The reason of high efficiency in write performance for LSM-tree based key-value stores is that, it converts random writes to sequential writes. This mechanism significantly improves the write throughput at the penalty of storing redundant data. Compaction is in charge of sort-merge these old data, which is similar to garbage collection. Compaction is often handled by background threads. Under heavy write workloads, system jam may occur, as flushing new data to disk is hindered by frequent compaction. This paper proposes an FPGA-based acceleration engine to alleviate the compaction impact.

Recent researches have been carried out to improve the performance of LSM-tree based key-value stores, but they often target to reduce the compaction frequency or delay the write pause [5]–[7]. The impact of compaction has not been totally removed. Lazy compaction is widely used in highly optimized LSM system. It allows key range overlap within certain tables or levels. Although the write throughput has been improved when compaction takes place in upper levels, it causes more severe performance degradation when compaction moves to lower levels. In addition, it also results in larger read amplifications. Few works focus on accelerating the compaction operation itself, without modifying the structure of database systems.

Database acceleration using FPGA is a popular trend in the recent years. Many works are performed on utilizing FPGA to improve the performance of RDBMS. Sukhwani et al. [8] applies FPGA to assist CPU on selection and decompression work. Casper et al. [9] utilizes FPGA to speed up join query. Woods et al. [10] chooses to insert an FPGA card between the disk and host machine, so that most of unwanted data are pre-filtered. In order to accelerate the processing of join queries, Zhou et al. [11] proposes a hardware-accelerated solution to speed up merge part for sort-merge join queries, especially for low-selectivity join queries. In this paper, we apply FPGA in LSM-tree based key-value stores to improve its compaction performance.

An FPGA accelerator for compaction faces the following challenges. Firstly, since CPU has an order of higher frequency than FPGA, it requires careful design of the compaction unit on FPGA to ensure better performance than CPU. Secondly, the pipeline design should be FPGA-optimized to take advantage of FPGA. Thirdly, to realize seamless integration with real LSM-tree database, the communication interface between software and hardware should be carefully designed with the consideration of data corruption. Finally, from the host side, how and when to invoke this hardware acceleration module to improve the overall throughput should also be investigated.

With consideration of these challenges, this paper proposes an FPGA-based compaction engine to accelerate the compaction procedure. The main contributions of this work are presented as follows:

- A software-hardware co-designed LSM-tree based architecture. FPGA plays the role of co-processor for accelerating the compaction process;
- A compaction engine on FPGA to accelerate the offloaded compaction work, including Decoder, Comparer and Encoder modules. The compaction speed is optimized by adopting index-data block separation, key-value separation, and full utilization of the FPGA bandwidth;
- Software integration with LevelDB. Uniform input and output memory interfaces are designed to improve com-

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00113

munication in this heterogeneous architecture. Compaction tasks scheduler at the host side is FPGA-optimized to further improve the write throughput of LevelDB;
- Comprehensive experimental results, which verify the improved performance of the proposed FPGA-based compaction acceleration engine (up to 92.0x, compared to CPU) and its contribution to the LSM-tree write throughput (up to 6.4x, compared to original LevelDB).

The rest of the paper is organized as follows. Section II introduces background on LSM-based key-value stores. Section III presents previous work on LSM-tree performance optimization. Section IV presents a system overview of the proposed solution. Section V illustrates the detail design of the proposed compaction engine on FPGA. Section VI shows the software integration with hardware. Section VII presents experimental results and related discussions. Section VIII presents the conclusion.

II. Background

In this section, LSM-tree based key-value stores is introduced first, followed by detail description on compaction and the storage format in LSM.

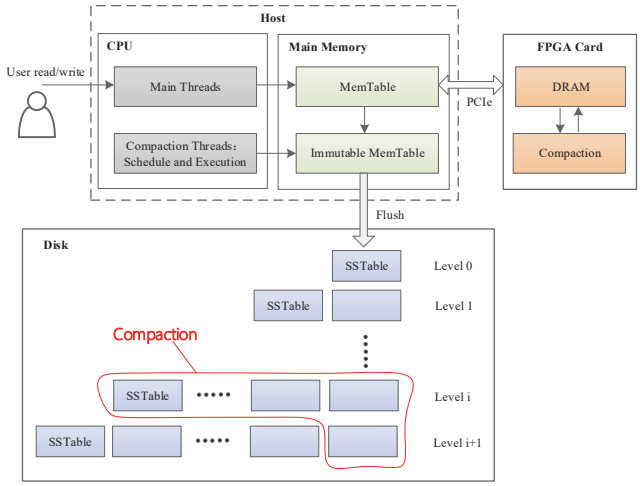

Fig. 1. System architecture of an LSM-tree based key-value store equipped with FPGA accelerator.

As shown in Fig. 1, the conventional LSM-tree based key-value stores are composed of two types of tasks. One is the main threads, which is in charge of handling write and read requests from users. The other is background threads, which is responsible for scheduling and executing compaction tasks. As a result, when the CPU is overloaded with heavy compactions, the main threads could be slowed down, which leads to longer response latency for users.

A. Compaction in LSM-tree Based Key-Value Stores

There are two types of compaction processes in LSM-tree. One is for storage format conversion. LSM achieves high write throughput by converting random writes into sequential writes, which is different from B-tree. This means the new data inserted by users are all appended to the old data. Even for deleted data, the delete flag is used to indicate its unavailability. As shown in Fig. 1, the newest data are stored in the **MemTable** in main memory using skiplists [12]. When the size of the MemTable reaches a threshold, it is marked as **Immutable MemTable**, and dumped into disk later. However, the data are stored on the disk in the format of **SSTable**, which consists of a series of sorted key-value pairs (details are introduced in Section II-B). The transformation of MemTable in main memory to SSTable on the disk is the first type of compaction.

The other compaction process merges data stored in different SSTables. These SSTables may be scattered throughout multiple levels. Each level contains limited number of SSTables with certain size. Once $Level\ i$ reaches the limit, the background threads collect relevant SSTables from $Level\ i$ and $Level\ i+1$, and then performs data sort-merge operations to generate new SSTables in $Level\ i+1$. Therefore, SSTables in all levels ($i > 0$) can be guaranteed in order, which means there is no key range overlap in these levels. Since SSTables in $Level\ 0$ are directly flushed from main memory, it becomes a special case and brings additional overhead to this type of compaction process.

In this paper, we mainly focus on the second type of compaction for data merging, which is often the main bottleneck of system performance.

B. SSTable Format Details

In order to improve the read performance, every SSTable is equipped with an index block at the end of the real key-value data blocks. The index block is also composed of a set of key-value pairs, where the key is the separation between two adjacent data blocks, while the value records the block size and offset. The size of each block is around a certain value (e.g., 4096 bytes). In other words, the key-value pairs in an SSTable are not global successively. After finishing merging a data block, it is required to stop scanning and fetch meta data of the next data block from index block, and then come back to continue. This handling mechanism is acceptable for CPU. However, it is not optimized for FPGA. The proposed FPGA-based compaction engine will solve this problem.

III. Related Work

Recently, performance optimization for LSM-tree based key-value stores has received a lot of research attention. In this section, relevant works are discussed from the perspective of compaction [13], including modifications on compaction policy, reducing compaction frequency, and with consideration of hardware.

For the common compaction mechanism, compaction takes place at the adjacent levels. Skip-tree [14] aims at merging keys from new $Level\ i$ and old $Level\ i+k$ directly (no more occurrence between these two levels). As a result, system can avoid unnecessary compactions. In a conventional LSM-tree, the key and value are always stored together. Wisckey [15]

stores the key and value separately, so that the write amplification and compaction cost can be decreased dramatically, but the overall performance is still impacted by the garbage collection of values. To solve this problem, Hashkv [16] hashes values into different groups according to the key and updates independently.

Several works try to alleviate the influence of compaction by means of reducing compaction frequency, which is also called lazy compaction. Pebblesdb [17] benefits from a special structure *guard* inspired by skiplist. It allows unsorted data in the same level. Tables from vertical levels are grouped under the control of guards. Dcompaction [5] introduces the notion of virtual SSTables. And the virtual compaction only changes the pointer of inputs without actual I/O operations. Several virtual compactions lead to a real compaction, so that the compaction is delayed. In Monkey [6] and Dostoevsky [7], SSTables are level-tiered in the upper levels and rigidly sorted in the lower levels. The number of threshold level can be adaptively varied. Although the above approaches can push off or reduce the compaction, the write pause phenomenon cannot be avoided under write-intensive workloads.

Some other works attempt to optimize LSM-tree by utilizing modern hardware. NoveLSM [18] takes advantages of high capacity and low latency of non-volatile memory (NVM) by extending MemTable and in-place updates on the NVM. The write throughput of LSM-tree can be increased in normal cases, and the compaction granularity still exits. SLM-DB [19] combines the advantages of B^+-tree and LSM-tree together. The B^+-tree are built in PM (persistent memory) for quick key search, while LSM-tree buffers are also kept in PM to ensure high write throughput. FlashKV [20] implements the LSM-tree based key-value stores in open-channel SSD, which reduces the impact from file system and FTL (flash translation layer) in normal SSD. The performance improvements come from direct data management and I/O parallelism through independent channels in SSD.

All the above works are devoted to improving the performance of the overall database system, rather than the compaction operation itself. Zhang et al. [21] divide the compaction process into several phases and arrange them in pipeline fashion for CPU handling. Alibaba introduces a write-optimized storage engine called X-Engine [22] for large-scale E-commerce Transaction Processing. They accelerate the write process from both software and hardware. For software optimization, they adopt data reuse strategy to reduce I/O numbers. They also take advantage of FPGA to accelerate compaction processing, while the details of hardware design are not disclosed.

In this paper, we propose a compaction engine designed with FPGA to improve the write performance of LSM-tree based key-value stores.

IV. System Overview

This section presents the overall system architecture of the proposed LSM-tree based key-value stores with FPGA accelerator. The overall architecture is illustrated in Fig. 1. At the host side, the CPU contains main threads and background compaction threads, which is similar to conventional LSM-tree based key-value stores. The main threads handle read and write requests from users. In the proposed scheme, background compaction threads take charge of scheduling compaction tasks, and offloading the compaction execution tasks to the FPGA card (through PCIe bus). The FPGA card is PCIe-attached to the host machine, and it is composed of an FPGA chip and off-chip DRAM. The details of the proposed FPGA-based compaction engine design will be illustrated in Section V.

When the number of SSTables in $Level\ i$ reaches the threshold, the background threads trigger a set of execution tasks described as follows:

1) CPU collects the meta data (e.g., name) of SSTables in $Level\ i$ and $Level\ i+1$ which need to be compacted;
2) Based on the level number, the background compaction threads calculate the number of inputs for the merging purpose. For $Level\ 0$, the key range may overlap with each other in several SSTables. Therefore, it is necessary to read keys from all SSTables to find the smallest key. In other words, the number of input for $Level\ 0$ is equal to the number of SSTables involved. For the other levels, the SSTables are all sorted. The smallest key must be in the first file. Therefore, the set of involved SSTables can be concatenated as a big SSTable, and the number of input is one;
3) For each input, CPU reads SSTable files from disk to a continuous memory block according to the key order and allocates space in memory for newly generated SSTables that will be returned by FPGA later;
4) The inputs data in memory are transmitted to DRAM on FPGA card through PCIe, in DMA mode;
5) Once all the input data arrive at DRAM, the hardware compaction engine starts its process, and read data from DRAM on FPGA card;
6) Partial compaction results are flushed to DRAM on FPGA card when the size of output data on FPGA chip reaches the storage limit;
7) When all the key-value pairs from inputs have been processed, the CPU receives an end signal, and fetches newly generated SSTables data to main memory;
8) CPU writes back these SSTables to the disk, and performs compaction post processing jobs (e.g. recording key range).

After describing the workflow of the overall architecture, we will present the implementation of compaction acceleration engine in detail from FPGA and host side separately. Notations of all the related parameters are listed in TABLE I.

V. Hardware Implementation

In this section, we first introduce the basic pipeline design of the proposed compaction engine. Then we present several important optimization modules to improve the compaction process based on FPGA's characteristics.

TABLE I
DEFINITION OF PARAMETERS.

Symbol	Definition
R_{req}	#Read request in decoding a key-value pair
L_{key}	Key length
L_{value}	Value length
V	Data transfer width on FPGA
N	Number of inputs allowed on FPGA
S_i	#involved SSTable in $Level\ i$
S_{index}	#key-value pairs in an index block
S_{data}	#key-value pairs in a data block
W_{in}	DRAM read width for Data Block
W_{out}	DRAM write width for Data Block

A. Basic Pipeline of Compaction

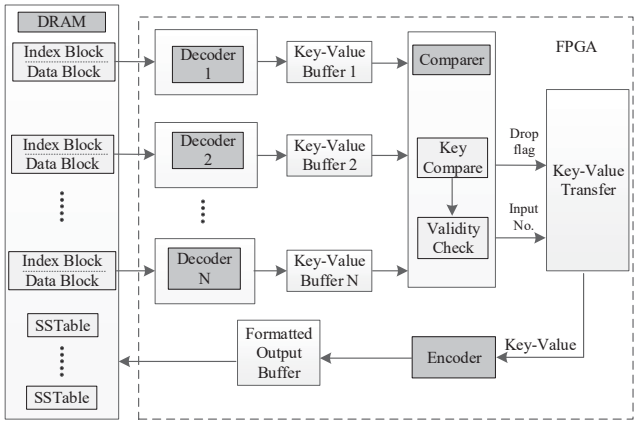

Fig. 2. Basic pipeline of compaction.

As shown in Fig. 2, the basic FPGA-based compaction engine is made up of three main modules: *Decoder*, *Comparer* and *Encoder*. Each original SSTable input is first decoded by its corresponding Decoder, and then the decoded key-value pairs will be compared at the Comparer. The smallest valid key-value pair will be selected and encoded in the Encoder.

Decoder is used to extract key-value pairs from SSTable. There are N Decoders in the compaction engine, which can process N inputs at a time. Each input contains one or several SSTables. Each SSTable has an index block and a set of data blocks, as mentioned in Section II-B. Algorithm 1 shows the workflow of the Decoder, which consists of three loops. The first loop is the reading procedure. The read pointer is first set to the first index block address in an SSTable, and incremented throughout the index block addresses. The second loop is the parsing procedure, in which the value in index block is parsed to obtain the size and offset of a data block. The third loop is the decoding procedure, each key-value pair in the data block will be decoded. Since keys in an SSTable are ordered, the Snappy compression [23] method is often applied to save storage space. As a result, decompression is needed in Decoder. After one data block has finished processing, the read pointer goes back to the index block for the meta data of the next data block. Similarly, when one SSTable has finished processing, the read pointer moves to the next SSTable, until the last one is processed. All the parsed key-values are stored in **Key-Value Buffers** and wait for the Comparer for further processing.

Algorithm 1 Mechanism of Decoder.

Input: DRAM read pointer;
Output: Decoded key-value streams;
1: **for** $x = 0$ TO $x < S_i$ **do**
2: Set read pointer to address of index block x;
3: **for** $y = 0$ TO $y < S_{\text{index}}$ **do**
4: Set read pointer to address of data block y;
5: **for** $z = 0$ TO $z < S_{\text{data}}$ **do**
6: Decoding key-value pair z using snappy method;
7: **end for**
8: **end for**
9: **end for**

Comparer module is in charge of selecting the smallest key and checking its validity. Comparer includes **Key Compare** module and **Validity Check** module. Key Compare module compares key one by one from each key-value buffers until the smallest one is found. However, this selected key may not always be valid. SSTable stores real key plus mark fields, which is treated as a whole in Decoder and Encoder. It is necessary to check the mark fields of the chosen key in Validity Check module. For example, if the *Delete* flag is set, this key-value should be considered invalid. After Comparer module finishes, the **Drop flag** is sent to **Key-Value Transfer** module. If the current smallest key-value is valid, it will be transmitted to Encoder module as an element of the newly generated SSTable. Otherwise, it should be dropped. Moreover, for Key-Value Transfer module to acquire which input possess the current smallest key, the **Input No.** should be sent to Key-Value Transfer module as well.

Encoder is designed to encode key-value pairs for an SSTable. The newly created SSTable needs to be reformed into the uniform format for better performance. Hence, the selected keys are compressed using *snappy* compression. When the size of a data block reaches a threshold (e.g., 4KB), the formatted data block are flushed into DRAM, and the meta data of this block is added into the index block. The index block is realized by BRAM (FPGA on-chip RAM). Similarly, the size of an SSTable also has a threshold (e.g., 2MB). When the accumulated size of data blocks becomes larger than this threshold, it indicates that this SSTable is completed. The index block can be written back to DRAM, and the Encoder gets reset for the next SSTable, until the last key-value. The smallest and the largest key of each SSTable are also recorded. After the entire encoding task finishes, these values are returned to the host side together with the newly produced SSTables.

B. Optimization for Decoder/Encoder

Based on the characteristics of FPGA, optimizations are carried out on the basic decoding and encoding modules, as shown in Fig. 3.

1) Decoder Separation: In order to improve the throughput of key-value decoding task, each Decoder module is split into **Index Block Decoder** and **Data Block Decoder**. The optimization is motivated by the following reasons.

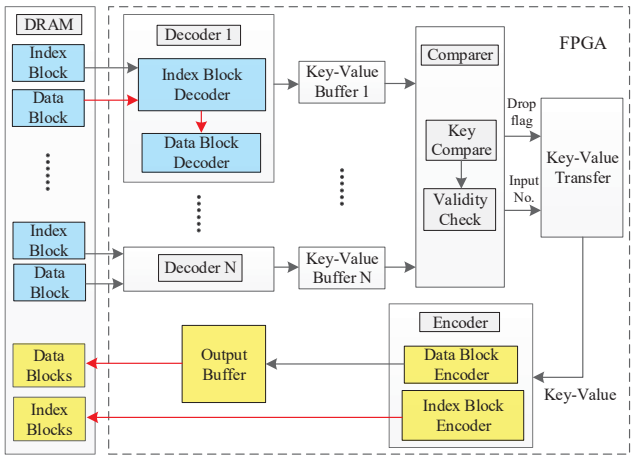

Fig. 3. Optimized pipeline of compaction: index block and data block are separated.

- From the Decoder in the basic design, it is found that the read pointer needs to be switched between index block and data block. When the Decoder process for a data block is completed, the process of generating key-values will pause, until meta data is acquired from index block again. If we have two read pointers for each SSTable, the index block decoding task and data block decoding task can be arranged in pipeline fashion, so that the index block decoding time could be hidden;
- The read/write latency of DRAM on FPGA card need to be considered. FPGA usually runs at 200-300 MHz. Under this condition, the read latency of memory on FPGA is 1 cycle while the read latency of DRAM is 7-8 cycles. It is more efficient to issue one large DRAM read request then multiple smaller DRAM read requests. In this optimized design, after Index Block Decoder parsing the size and offset of a data block, the whole data block is streamed into FPGA, which is implemented by FIFOs (First in, first out, the read latency is 1 cycle). As a result, the read latency of decoding key-value in data block can be reduced to $(1 \cdot R_{\text{req}})$ cycles.

2) Encoder Separation: Similarly, the Encoder module is also split into **Index Block Encoder** and **Data Block Encoder**. In the basic design, for each standard SSTable, the index block is always placed at the end of several data blocks. The whole index block will be buffered in the BRAM, until all data blocks finish encoding, which would both increase the formatted output SSTable transfer time and BRAM resource usage. In this optimized design, when separation is adopted, once a data block is generated, the corresponding index block can be transmitted to DRAM immediately, and thus index block transfer time and BRAM consumption accumulation can be avoided.

The host side also needs relevant modifications. Before compaction starts, the pointers of index block and first data block in all input SSTables should be transferred to FPGA. After FPGA finishes, the host is in charge of combining data blocks with index blocks into new formatted SSTables.

C. Optimization on Key-Value Separation

In the conventional LSM-tree based key-value stores, a key-value pair is always treated as a unity and stored together. Considering the characteristic difference between key and value, the proposed FPGA-based compaction engine handles key and value separately. Fig. 4 illustrates this optimization.

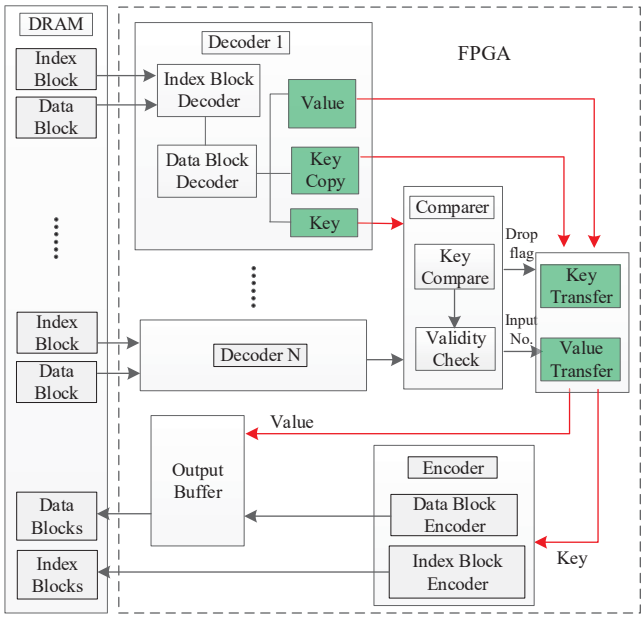

Fig. 4. Optimized pipeline of compaction: key and value are separated.

It is worth noted that only in decoding and encoding process, the value needs to be transmitted, while it is totally ignored in the comparing procedure. While it is acceptable to transmit key and value together for CPU handling as it works with high frequency, it is too expensive for FPGA to conduct in the same manner. Longer value length may lead to increases of total execution time. Considering the data value is not needed in Comparer module, the key-value separation strategy is adopted in the proposed compaction engine. Relevant optimizations have been made as follows.

1) The decoded key-value stream processed by the proposed Decoder module is separated into three streams: value stream, the original key steam and a copy of this key stream. The original key stream is sent to the Key

Compare module to obtain the smallest key. The value stream and the copy key stream are transmitted to the Key-Value Transfer module, and wait to be selected by the result from Comparer. The reason of utilizing key copy stream is that the element in FIFO can be used only once. The reason for FIFO chosen to store the key and value instead of BRAM is that FIFO is easier to be synchronized;

2) Since the key value stream are separated in Key-Value Transfer, the validity check flag can be used to select the key stream and value stream at the same time;
3) The valid key stream from Key-Value Transfer is delivered to Encoder for snappy compression, while the valid value stream is sent to output buffer directly.

In conclusion, during the compaction process on FPGA, the proposed solution can perform better by cutting down the unnecessary value stream transmissions.

D. Optimization for Data Transmission

In this subsection, we present the optimized design on improving the bandwidth of value transmission and input/output of the FPGA chip. Fig. 5 demonstrates the optimized version of our proposed FPGA-based compaction engine.

1) Bandwidth of Value Transmission: After optimization with key-value separation, the approximate period of each module in pipeline are listed in TABLE II. Considering the Index Block Decoder and Encoder have low duty cycle, their timing cost are not listed.

TABLE II
PERIOD OF MAIN MODULES IN PIPELINE.

Module Name	Cycles	Calculation Method
Data Block Decoder	$L_{\text{key}} + L_{\text{value}}$	decoding key + value read
Comparer	$(2+\lceil \log_2 N \rceil) \times L_{\text{key}}$	key read + key compare + check key if existing
Key-Value Transfer	$Max(L_{\text{key}}, L_{\text{value}})$	longer time between key and value transfer
Data Block Encoder	L_{key}	encoding key

The module with the longest cycles determines the average execution time in a pipeline system. The value length is often much larger than key length. Therefore, the bottleneck is often the value stream transmission.

One of the most important feature of FPGA is high parallelism available. Several bytes of data can be transmitted at the same time. If we set the data width to V, the value transmission time will be reduced to $\frac{L_{\text{value}}-1}{V}+1$. For simplicity, the approximation $\frac{L_{\text{value}}}{V}$ is adopted in this work.

TABLE III gives the optimized execution time. After optimization, the longest cycles is either Data Block Decoder or Comparer. If $L_{\text{key}} + \frac{L_{\text{value}}}{V} > (2 + \lceil \log_2 N \rceil) \times L_{\text{key}}$, which can be further simplified as $L_{\text{key}} < \frac{L_{\text{value}}}{(1+\lceil \log_2 N \rceil) \times V}$, the longest cycles is Data Block Decoder. Otherwise, the bottleneck of execution time is changed to Comparer module.

TABLE III
OPTIMIZED PERIOD OF MAIN MODULES IN PIPELINE.

Module Name	Cycles
Data Block Decoder	$L_{\text{key}} + \frac{L_{\text{value}}}{V}$
Comparer	$(2+\lceil \log_2 N \rceil) \times L_{\text{key}}$
Key-Value Transfer	$Max(L_{\text{key}}, \frac{L_{\text{value}}}{V})$
Data Block Encoder	L_{key}

2) Bandwidth of FPGA Chip Input/Output: The FPGA chip reads/writes data on DRAM through Advanced eXtensible Interface (AXI) protocol. It allows a maximum of 512-bit (64-byte) data transferred per cycle. When the FPGA resource is sufficient, in order to take full advantage of bandwidth, the data blocks of each input on DRAM are fetched to the FPGA chip in the form of 64-byte/cycle ($W_{\text{in}} = 64$) and the same for output buffer writing back to DRAM ($W_{\text{out}} = 64$). There is no need to make this modification for index block, since it has low read request frequency.

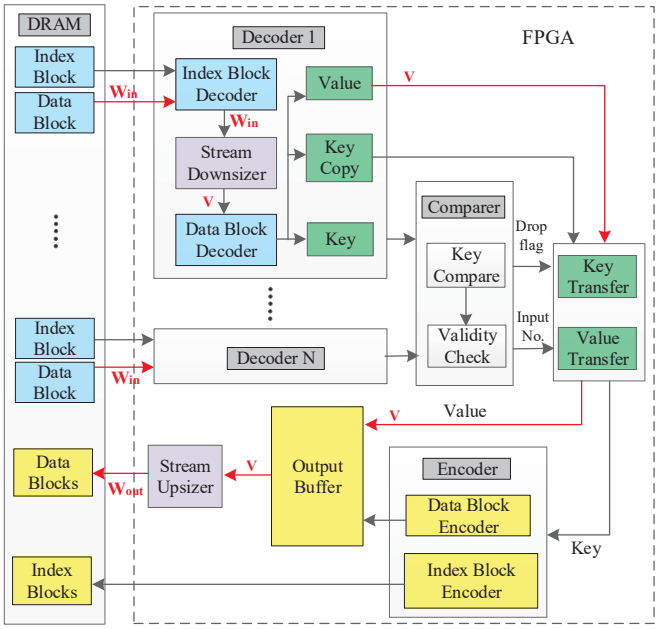

Fig. 5. Optimized pipeline of compaction: increase data transmission bandwidth.

The rate of AXI read is W_{in}-byte/cycle, while the consuming rate of data block decoding is V-byte/cycle ($V <= W_{\text{in}}$). As a result, a **Stream Downsizer** is inserted before each Data Block Decoder to narrow down the input stream. Similarly, a **Stream Upsizer** is added after output buffer, so that AXI write rate can be increased to W_{out}-byte/cycle. Here, we only expand the bandwidth for Data Block Encoder, as Index Block Encoder is invoked only when each data block finishes and the size of each entry is small.

VI. Software Integration with Hardware Compaction Engine

The proposed FPGA-based compaction engine plays the role of execution engine of compaction thread for LSM-tree based key-value stores, which requires seamless integration with the host side. The proposed engine is implemented with LevelDB [24], which is one of the most popular LSM-tree based key-stores designed by Google. Many other databases are developed based on LevelDB (e.g., RocksDB [4], IndexedDB [25]). The proposed compaction engine is also applicable to other similar key-value stores with a few modifications.

In this section, the procedure of compaction thread is introduced first. And then the memory interface between software and hardware are presented.

A. Workflow of Compaction Thread

Ideally, the compaction task schedule is processed by the host side, while the execution is to be offloaded to FPGA. However, due to the resource limitations of FPGA, it supports up to N inputs compaction. Therefore, when the number of involved SSTable in $Level\ 0$ (S_0) is larger than $N-1$, the compaction task will be processed completely by the software, as indicated in Fig. 6 **SW Compaction** module.

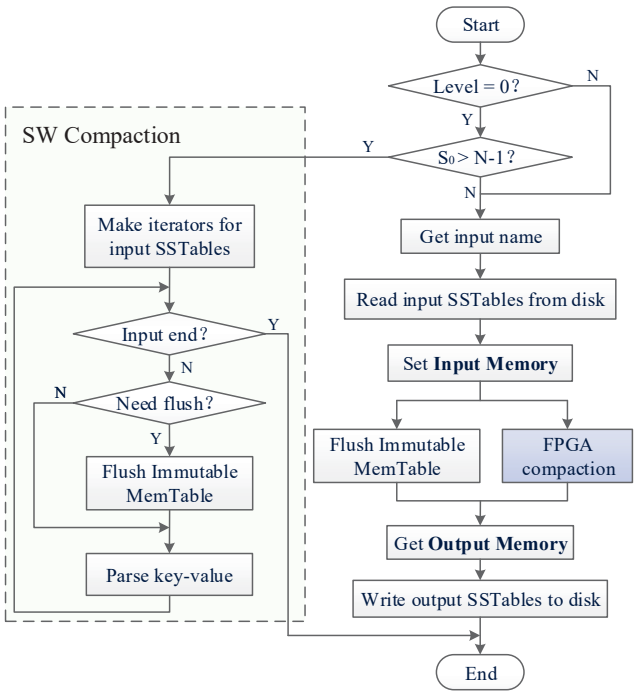

Fig. 6. Workflow of compaction thread.

As mentioned in Section II-A, the first type of compaction that dumps Immutable MemTable to $Level\ 0$, has higher priority than the second compaction type (FPGA focused). When performing software compaction, it is required to check whether Immutable MemTable dump is needed before parsing key-value. If it is needed, the second type of compaction should be paused. This default schedule is not efficient. With the proposed scheme, when the second type of compaction is pushed down to FPGA, the flush operation at the host side can be executed at the same time. In this way, the write throughput can be increased, which is another benefit of the proposed scheme.

B. Memory Interface

As shown in Fig. 6, before FPGA starts, all the involved input SSTables should be read from disk. And after FPGA finishes, the new SSTables from output should be written back to disk. For this heterogeneous architecture, the data should be unified in input and output memory interface.

Considering the separated Index and Data block Decoder/Encoder on FPGA, the data blocks and index blocks should be stored in different memory locations for both input and output. Fig. 7 presents the format of **Index Block Memory** and **Data Block Memory**.

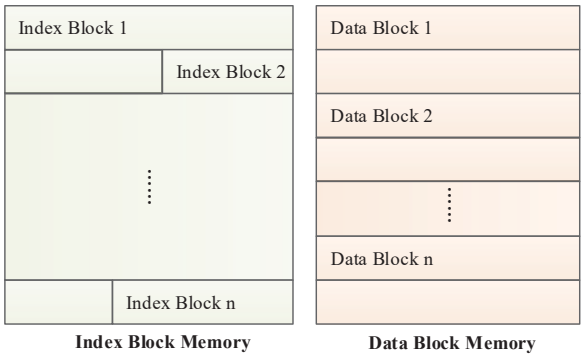

Fig. 7. Block Memory layout.

Index Block Memory: For each input and generated output, the index block of each SSTable can be placed continuously.

Data Block Memory: As indicated in Section V-D, the data block is transferred at W_{in}-byte/cycle for reading. Therefore, the data blocks of input SSTables should be stored in W_{in}-byte aligned. Similarly, the newly generated SSTables are saved in series with W_{out}-byte alignment.

Besides the SSTable data, several additional information should be provided to the FPGA compaction engine, as illustrated in Fig. 8.

MetaIn Memory. A meta block is required for each input. It stores the number of SSTables and the offset of index block and first data block in their corresponding memory region.

MetaOut Memory. To improve the read performance of LSM-tree based key-value stores, the smallest and the largest key of each SSTable are maintained. These keys are returned to the host side. In addition, the number of output SSTables and the size of each are needed.

The **Input Memory** in Fig. 6 consists of Index block, Data block and MetaIn Memory, while the **Output Memory** includes Index block, Data block and MetaOut Memory.

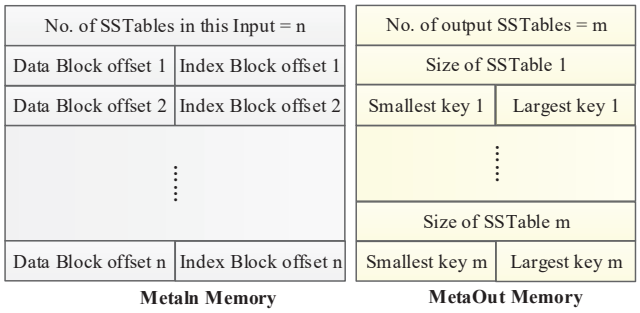

Fig. 8. Meta Memory layout.

VII. Experiments

In this section, we conduct a series of experiments to evaluate the performance improvement of the proposed FPGA-based Compaction Acceleration Engine (named FCAE). We use *LevelDB* to represent the original database, and *LevelDB-FCAE* to represent the proposed FPGA-based compaction engine.

We first introduce the experimental setup of the experiments. Then, we investigate the performance of FCAE with different input N, aiming to evaluate if the hardware engine can support compaction with key range overlap inputs. In both cases, we compare the compaction speed between single CPU thread and FCAE. For end-to-end evaluation, we compare the write performance between the original LevelDB and LevelDB-FCAE, considering different data size, data transfer bandwidth on FPGA (optimized parameter V mentioned in Section V-D) and intrinsic parameters of LSM-tree based key-value stores.

A. Experimental Setup

The proposed FPGA-based compaction engine is implemented on Xilinx Kintex UltraScale FPGA KCU1500 card. This board is equipped with 16GB off-chip DRAM, and communicates with host machine through PCIe gen3 $\times$ 16. The hardware compaction engine runs at 200 MHz.

For the host side, the computer has a 12-core i7-8700K CPU (@3.7GHz), with Ubuntu 16.04 operating system. We use LevelDB v1.1 as the baseline of our modification and integrate it with the proposed FPGA-based compaction engine. For fair comparison, LevelDB runs with 2 CPU cores, while LevelDB-FCAE runs with 1 CPU core + FPGA card.

To evaluate the write performance of LevelDB with compaction interference, the built-in benchmark of LevelDB, **db_bench** and **YCSB** [26] benchmark are used. The settings of LevelDB are presented in TABLE IV.

B. Evaluation on 2-input FCAE

When $N = 2$, for LevelDB, all levels except for $Level\ 0$ can be handled by FCAE. In this case, the resource of FPGA is sufficient, and W_{in} and W_{out} are both set to 64 for high performance. The bandwidth of value data transfer V can be tuned. In this subsection, we investigate the impact of V variation on the acceleration ratio.

TABLE IV
LevelDB settings.

Parameter	Default	Range
Key length (Bytes)	16	[16, 256]
Value length (Bytes)	128	[64, 2048]
Leveling ratio	10	[4, 16]
Data block size (KB)	4	[2, 1024]

1) Compaction Speed: In this part, we compare the compaction speed of FCAE and single CPU thread. The compaction speed is defined as *Size of input SSTables / Kernel compaction time*. The kernel time is measured assuming that all input and output memory are already set and PCIe transfer time is not considered. To investigate the relation between V and the compaction speed. V is varied from 8 to 64. In each test case, the length of value increases from 64 bytes to 2048 bytes. TABLE V shows the compaction speed with respect to different parameter sets, and their corresponding acceleration ratio results are given in Fig. 9.

TABLE V
Compaction speed with different value length and V.

L_{value} (bytes)	Compaction speed (MB/s)				
	CPU	$V = 8$	$V = 16$	$V = 32$	$V = 64$
64	5.3	178.5	164.5	181.8	175.8
128	6.9	260.1	312.1	311.8	291.7
256	9.0	343.9	451.6	510.7	524.9
512	12.2	446.9	627.9	672.8	745.4
1024	14.8	448.5	739.5	896.7	1026.3
2048	13.3	506.3	709.0	1077.4	1205.6

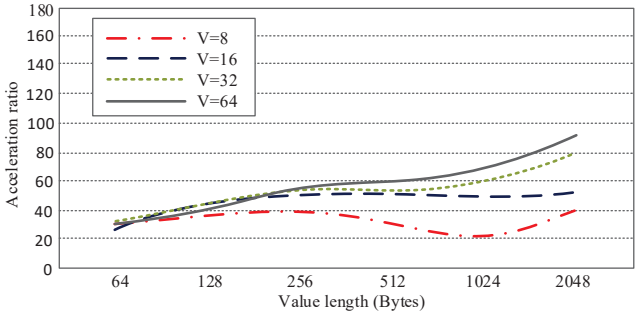

Fig. 9. Acceleration ratio of FCAE compaction speed with different value length and V.

The result indicates that the compaction speed of CPU and FCAE both increase when length becomes larger, while FCAE grows faster. This increasing acceleration ratio benefits from the hardware optimization strategy: key-value separation and bandwidth increase of data transmission. Value streams are not needed during the decoding and encoding process, hence the

transmission can be reduced. Furthermore, the bandwidth of FPGA chip is fully utilized.

The variation of V also has influence on the compaction speed. The increase of V leads to the growth of compaction acceleration ratio in general. The special case is when value length is 1024 bytes, the acceleration ratio shows a little bit reduction, with $V = 8$ and $V = 16$. As mentioned in Section V-D, when $L_{\text{key}} < \frac{L_{\text{value}}}{2V}$, the bottleneck is Data Block Decoder, otherwise is Comparer. As a result, when $V = 8, L_{\text{value}} = 1024$, and $V = 16, L_{\text{value}} = 1024$, the longest cycles become 152 and 88 ($L_{\text{key}} + \frac{L_{\text{value}}}{V}$), not original 72 ($3 \times L_{\text{key}}$)[1]. This decline does not happen when value length is 2048, because the compaction speed of CPU drops off.

2) Evaluation on Write Throughput: In this subsection, the write throughput performance of LevelDB is evaluated based on **db_bench**. It is an end-to-end assessment, including PCIe transfer time.

a) Data Size Variation: The write traffic is one of the key factors to hinder the write performance of LSM-tree based key-vale stores. Hence, we vary the data size of workloads from 0.2 GB to 2 GB, with fixed factors $L_{\text{value}} = 512, V = 16$.

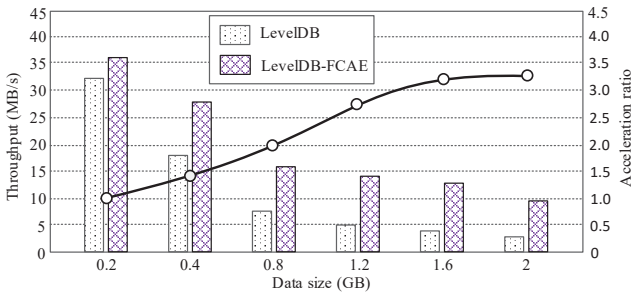

Fig. 10. Write throughput with different data size.

As shown in Fig. 10, the write throughput of LevelDB decreases dramatically when data size increases. However, the throughput of LevelDB-FCAE degrades gently. The reason is that, when data size is small, the pressure from compaction thread is small, and it happens in low frequency. When a large amount of data burst in, the situation becomes more serious due to resource contention. The compaction thread have to stop flushing Immutable MemTable to disk, and merges the SSTables much more often. For LevelDB-FCAE, FPGA compaction engine alleviates the SSTable compaction task execution for CPU and thus achieves more stable performance.

b) Value Length Variation: In Section VII-B1, it has been illustrated that value length and data transfer width V have effect on compaction speed. Therefore, we also need to evaluate how the above two factors affect write throughput of real database.

TABLE VI and Fig. 11 reveal that FCAE improves the write throughput of LevelDB, and the acceleration ratio gets a steady increase when value length increases. For the original LevelDB, the write throughput decreases a little bit when

[1]Here, $L_{\text{key}} = 16$ (real key length) $+ 8$ (mark fields)

TABLE VI
WRITE THROUGHPUT WITH DIFFERENT VALUE LENGTH AND V.

L_{value} (bytes)	Throughput (MB/s)				
	LevelDB	$V = 8$	$V = 16$	$V = 32$	$V = 64$
64	2.4	5.6	5.4	5.6	5.4
128	2.9	6.5	7.7	7.6	7.6
256	2.5	5.8	7.1	7.2	7.2
512	2.8	6.0	9.1	9.6	9.3
1024	2.3	6.7	9.8	11.0	11.6
2048	2.3	10.9	12.3	14.1	14.4

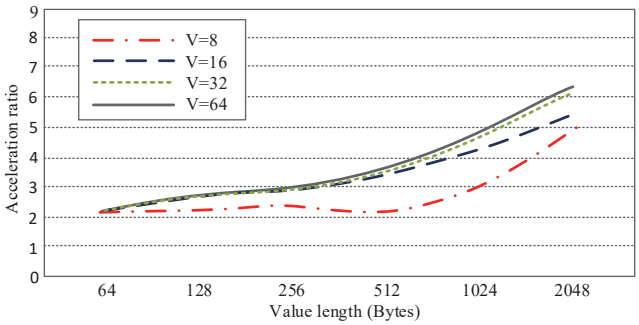

Fig. 11. Acceleration ratio of LevelDB-FCAE throughput with different value length and V.

value length rises. The reason is that even for CPU with high working frequency, the value data movement also degrade the compaction performance.

When the value length is fixed, variation of V also has different influence. For small value length, the acceleration ratio of FCAE has tiny changes since the processing bottleneck is Comparer, which has a stable period $3 \times L_{\text{key}}$. However, for long value length, the obstacle is Data Block Decoder with period $L_{\text{key}} + \frac{L_{\text{value}}}{V}$. It is obvious that this value reduces when V increases.

C. Evaluation on Multi-input FCAE

In modern write-optimized LSM-tree based key-value stores, partitioned tiering merge is adopted such as SifrDB [27] or PebblesDB [17], which may allow key range overlap in some levels (except for $Level\ 0$), $N = 2$ is not enough for handling these cases. Hence, we extend the maximum input allowance N to adapt FCAE to lazy compaction.

When it comes to $Level\ 0$ compaction in LevelDB, eight SSTables on $Level\ 0$ and $Level\ 1$ are involved in compaction process in most cases, which means $N = 9$. For other lazy compaction adopted LSM key-value stores, when the input number is not larger than nine, the compaction tasks would be pushed down to FPGA, otherwise it is handled by CPU.

1) FPGA Configurations: When $N = 9$, if W_{in} and W_{out} are still set to maximum 64, the resource of FPGA would run out. Considering that the output of compaction process is single, while input comes from multiple paths, we keep

$W_{\text{out}} = 64$ and decrease W_{in} and V. TABLE VII lists the FPGA resource utilization under different configurations of N, W_{in} and V. It is noted that the exact same configuration as $N = 2$ is far from acceptable. The reason is that the Stream Downsizer module on FPGA consume considerable LUT resource, and the added Decoder would occupy all of them. Due to the limited resource of the FPGA platform used in the experiments, $W_{\text{in}} = 8$ and $V = 8$ are picked for 9-input FCAE. We also compare the compaction speed with 2-input FCAE as value length changes from 64 to 2048 bytes.

TABLE VII
RESOURCE UTILIZATION FOR DIFFERENT FPGA CONFIGURATIONS.

N	W_{in}	V	BRAM	FF	LUT
2	64	16	18%	10%	72%
		8	17%	9%	63%
9	64	8	35%	27%	206%
	16	16	30%	18%	125%
	16	8	26%	16%	103%
	8	8	25%	14%	84%

As shown in Fig. 12 and Fig. 13, the 9-input FCAE has compaction speed degradation in comparison with 2-input for small value length (about 70%). However, the performance gap is narrowed down as the length increases. Because the bottleneck moves from Comparer to Data Block Decoder, and the period of latter module is almost the same for $N = 2$ and $N = 9$.

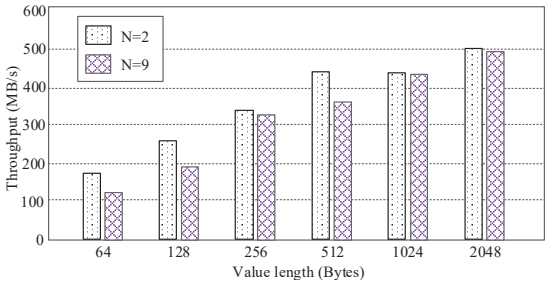

Fig. 12. Compaction speed comparison with different value length.

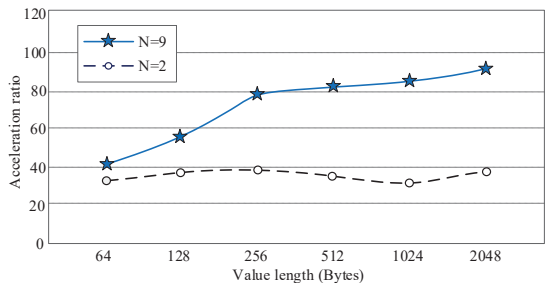

Fig. 13. Acceleration ratio comparison with different value length.

Although the compaction speed reduces for 9-input FCAE, the acceleration ratio becomes even larger in comparison with the CPU baseline. Apart from the above mentioned hardware optimization strategy: key-value separation and data transmission, the increasing acceleration ratio also benefits from the parallel Comparer module on FPGA.

2) Data Size Variation: In this subsection, we evaluate the write throughput of LevelDB-FCAE, as data size grows from 0.2 GB to 1024 GB. The levelDB settings are the same as shown in TABLE IV, except for $L_{\text{value}} = 512$. As data size increases, the PCIe data transfer time become non-negligible. Therefore, we also investigate whether the data transfer time will impact the whole system performance.

The results in Fig. 14 indicate that the write performance of LevelDB and LevelDB-FCAE drop significantly as the data size grows. This is reasonable since the cost of high level compaction is expensive. The write pause may happen as data accumulated. FPGA cannot eliminate but can alleviate this problem. As data increases to extremely large amount, the speedup of LevelDB-FCAE also decreases, and then becomes steady around 2.5x.

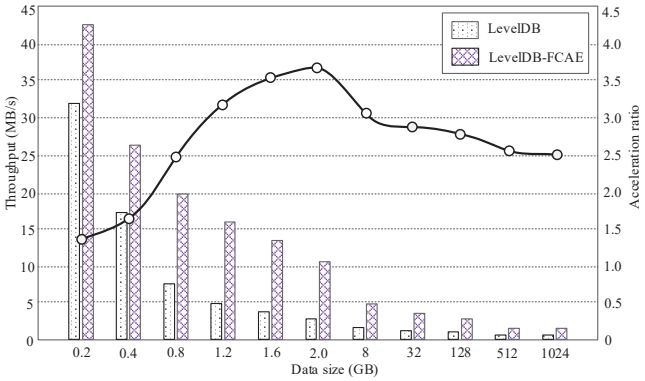

Fig. 14. Write throughput with different data size.

From TABLE VIII, we find that the PCIe data transfer time only occupies a small portion of the whole system execution time. Even for large data size, the time consumption of data transfer is negligible ($< 1\%$).

TABLE VIII
PCIE TRANSFER PERCENTAGE.

Size(GB)	0.2	0.4	0.8	1.2	1.6	2	8	32	128	512	1024
Transfer	9%	7%	8%	8%	6%	6%	3%	2%	1%	<1%	<1%

3) Sensitivity Study on LevelDB Settings: In this subsection, we evaluate the impact of LevelDB settings variation on the LevelDB-FCAE performance, with respect to key length, value length, data block size and leveling ratio. When one parameter varies, the others are set to the default value in TABLE IV. The experimental results are shown in Fig. 15.

a) Key Length Variation: As mentioned in Section V-D, when $L_{\text{key}} < \frac{L_{\text{value}}}{(1+\lceil \log_2 N \rceil) \times V}$, the bottleneck is Data Block De-

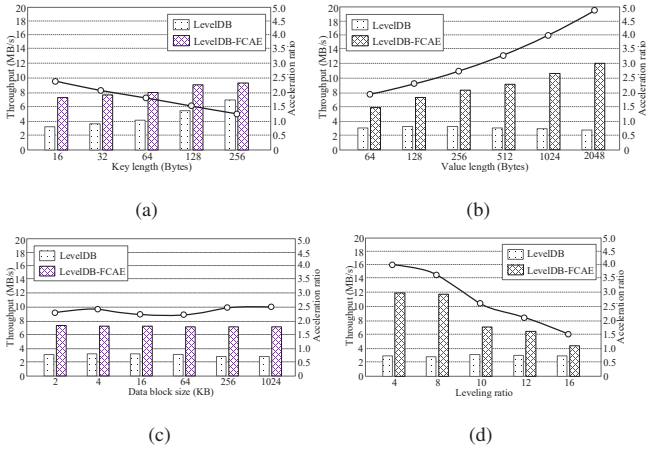

Fig. 15. LevelDB settings variations: (a) Key length, (b) Value length, (c) Data block size, (d) Leveling ratio.

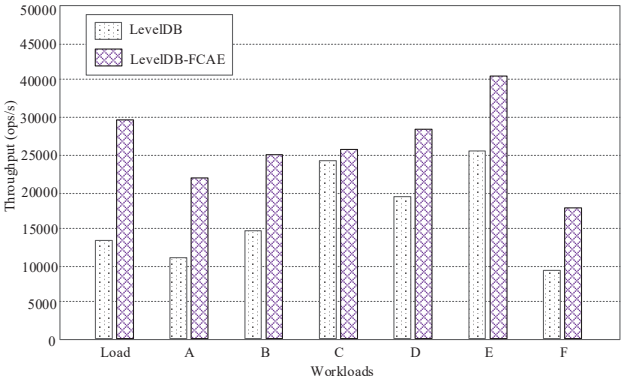

Fig. 16. YCSB throughput for LevelDB and LevelDB-FCAE.

coder, otherwise is Comparer. Using $N = 9, V = 8, L_{\text{value}} = 128$, the equation comes to $L_{\text{key}} < 3.2$. As a result, the Data Block Decoder is always the bottleneck, and the period is $L_{\text{key}} + 16$. The variation trend in Fig. 15(a) also verifies this deduction. The acceleration ratio of LevelDB-FCAE decreases linearly as key length grows from 16 to 256 bytes.

b) Value Length Variation: Similarly, the speedup of LevelDB-FCAE increases as value length grows in Fig. 15(b), which is similar to the trend in Section VII-B2.

c) Data Block Size Variation: From Fig. 15(c), it is revealed that the write throughput of LevelDB and LevelDB-FCAE are both unrelated to data block size, and the acceleration ratio maintains at 2.4x.

d) Leveling Ratio Variation: The leveling ratio is defined as *Size(*$Level\ i + 1$*) / Size(*$Level\ i$*)*. As shown in Fig. 15(d), the speedup of LevelDB decreases as leveling ratio increases. The reason is that the larger the ratio is, the compaction is triggered less frequently. Accordingly, FPGA gets less chance to make a difference.

In conclusion, FPGA compaction engine has much better performance to accelerate compaction in those cases with short keys, long values and leveling ratio not larger than 10.

D. Evaluation on YCSB

We also evaluate the LevelDB-FCAE using YCSB benchmark, which is a popular benchmark for key-value stores. It contains six workloads as described in TABLE IX. We first load 20M records (around 20GB) into the database, where each record consists of 16-byte key and 1024-byte value. The operation number for each workload is 20M. The read requests follow the latest distribution in Workload D, while read requests in other workloads follow the zipfian distribution. In this evaluation, we apply multi-input FCAE, and the settings of LevelDB are default.

As shown in Fig. 16, the proposed LevelDB-FCAE outperforms LevelDB in all workloads. Even for the read-only Workload C, the throughput is not degraded in comparison with original LevelDB, since we make no change on the storage structure of original database. It is shown that, with the increases of write ratio, the acceleration ratio of LevelDB-FCAE increases. For the write-only Load operation, the speedup can achieve the maximum 2.2x.

E. Discussion

The experimental results verify that using FPGA can accelerate compaction for LSM-tree based key-value stores. However, a direct implementation from code running in CPU would not work, as FPGA has only one-tenth frequency of CPU. Therefore, the design optimization for FPGA is critical and challenging. On one hand, we should fully utilize the parallelism of FPGA for acceleration. On the other hand, the limited resource on FPGA restricts the number of available parallel units. In conclusion, the pipeline design for FPGA is the key and it is non-trivial.

In the proposed solution for compaction acceleration, the FPGA plays the role of co-processor in the heterogeneous system, which is similar to the common GPU-CPU architecture. The CPU plays the leader role and all related data should be first read to main memory, and then transmitted to FPGA. This is a general architecture, which is also popular and provided by major cloud vendors.

Another recent trend is near storage computing, which brings computing resources closer to the data source. As an FPGA-based prototype, the FPGA is placed in SSD as an embedded controller. Existing research works mainly focus on RDBMS read optimization, e.g., SmartSSD (ARM-based) [28] and big data analytics, e.g., BlueDBM [29]. In this architecture, FPGA can fully utilize the internal bandwidth of SSD, so that the redundant data transfer is minimized. One challenge is to design a general communication protocol between the host and the FPGA, considering the difficulty in mapping the address of task related data. We are exploiting this direction as a next step.

VIII. Conclusion

This paper proposes an FPGA-acceleration solution for LSM-tree based key-value stores. A compaction engine with

TABLE IX
DESCRIPTION OF YCSB WORKLOADS.

Workload	Load	A	B	C	D	E	F
Description	write	read/update	read/update	read	read/write	range query/write	read/read-modify-write
Percentage (%)	100	50/50	95/5	100	95/5	95/5	50/50

high efficiency is designed and implemented on FPGA. We optimize the index and data block pipeline to reduce the impact of DRAM latency. Moreover, considering the different functionality of key and value in compaction, the above two elements are separated during processing. By leveraging high bandwidth of FPGA chip and parallel value transmission, the compaction performance is significantly improved. Furthermore, this FPGA-based compaction engine is merged with LevelDB, with unified memory interface and judicious task scheduling under heterogeneous architecture. Experimental results demonstrate that the proposed hardware compaction engine can achieve up to 92.0x acceleration ratio compared with the CPU baseline, and improve the write throughput of LevelDB by up to 6.4 times.

ACKNOWLEDGMENT

The work described in this paper was partial supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CityU 11219319). We also thank Xilinx for their generous donation of KCU1500 FPGA board.

REFERENCES

[1] "Cassandra," http://cassandra.apache.org/.
[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, 2008.
[3] "Hbase," https://hbase.apache.org/.
[4] "Roksdb," https://rocksdb.org/.
[5] F. Pan, Y. Yue, and J. Xiong, "dcompaction: Speeding up compaction of the lsm-tree via delayed compaction," *J. Comput. Sci. Technol.*, vol. 32, no. 1, pp. 41–54, 2017.
[6] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal navigable key-value store," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD), Chicago, IL, USA*, 2017, pp. 79–94.
[7] N. Dayan and S. Idreos, "Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging," in *Proceedings of the International Conference on Management of Data (SIGMOD), Houston, TX, USA*, 2018, pp. 505–520.
[8] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. W. Asaad, "Database analytics acceleration using fpgas," in *International Conference on Parallel Architectures and Compilation Techniques (PACT), Minneapolis, MN, USA*, 2012, pp. 411–420.
[9] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *The ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), Monterey, CA, USA*, 2014, pp. 151–160.
[10] L. Woods, Z. István, and G. Alonso, "Ibex - an intelligent storage engine with support for advanced SQL off-loading," *PVLDB*, vol. 7, no. 11, pp. 963–974, 2014.
[11] Z. Zhou, C. Yu, S. Nutanong, Y. Cui, C. Fu, and C. J. Xue, "A hardware-accelerated solution for hierarchical index-based merge-join," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 1, pp. 91–104, 2019.
[12] "Skiplist," https://en.wikipedia.org/wiki/Skip_list.
[13] C. Luo and M. J. Carey, "Lsm-based storage techniques: A survey," *CoRR*, vol. abs/1812.07527, 2018.
[14] Y. Yue, B. He, Y. Li, and W. Wang, "Building an efficient put-intensive key-value store with skip-tree," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 4, pp. 961–973, 2017.
[15] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," *TOS*, vol. 13, no. 1, pp. 5:1–5:28, 2017.
[16] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu, "Hashkv: Enabling efficient updates in KV storage via hashing," in *USENIX Annual Technical Conference (ATC), Boston, MA, USA*, 2018, pp. 1007–1019.
[17] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "Pebblesdb: Building key-value stores using fragmented log-structured merge trees," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China*, 2017, pp. 497–514.
[18] S. Kannan, N. Bhat, A. Gavrilovska, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Redesigning lsms for nonvolatile memory with novelsm," in *USENIX Annual Technical Conference (ATC), Boston, MA, USA*, 2018, pp. 993–1005.
[19] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y. Choi, "SLM-DB: single-level key-value store with persistent memory," in *17th USENIX Conference on File and Storage Technologies (FAST), Boston, MA, USA*, 2019, pp. 191–205.
[20] J. Zhang, Y. Lu, J. Shu, and X. Qin, "Flashkv: Accelerating KV performance with open-channel ssds," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5, pp. 139:1–139:19, 2017.
[21] Z. Zhang, Y. Yue, B. He, J. Xiong, M. Chen, L. Zhang, and N. Sun, "Pipelined compaction for the lsm-tree," in *IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA*, 2014, pp. 777–786.
[22] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li, "X-engine: An optimized storage engine for large-scale e-commerce transaction processing," in *Proceedings of the International Conference on Management of Data (SIGMOD), Amsterdam, The Netherlands*, 2019, pp. 651–665.
[23] "Snappy," https://en.wikipedia.org/wiki/Snappy_(compression).
[24] "Leveldb," https://github.com/google/leveldb.
[25] "Indexeddb," https://en.wikipedia.org/wiki/Indexed_Database_API.
[26] "Ycsb," https://github.com/brianfrankcooper/YCSB.
[27] F. Mei, Q. Cao, H. Jiang, and J. Li, "Sifrdb: A unified solution for write-optimized key-value stores in large datacenter," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC), Carlsbad, CA, USA*, 2018, pp. 477–489.
[28] J. Do, Y. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart ssds: opportunities and challenges," in *Proceedings of the International Conference on Management of Data (SIGMOD), New York, NY, USA*, 2013, pp. 1221–1230.
[29] S. W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind, "Bluedbm: an appliance for big data analytics," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA), Portland, OR, USA*, 2015, pp. 1–13.