STEM: Streaming-based FPGA Acceleration for Large-Scale Compactions in LSM KV

Dongdong Tang
City University of Hong Kong
Hong Kong
dtang8-c@my.cityu.edu.hk

Weilan Wang*
City University of Hong Kong
Hong Kong
weilawang2-c@my.cityu.edu.hk

Yu Mao*
City University of Hong Kong
Hong Kong
yumao7-c@my.cityu.edu.hk

Jinghuan YU
City University of Hong Kong
Hong Kong
jinghuayu2-c@my.cityu.edu.hk

Tei-Wei Kuo
National Taiwan University,
and Mohamed bin Zayed University of Artificial Intelligence
ktw@csie.ntu.edu.tw

Chun Jason Xue
Mohamed bin Zayed University of Artificial Intelligence
jason.xue@mbzuai.ac.ae

***Abstract*—Log-Structured-Merge-tree (LSM-tree) has been extensively adopted because of its exceptional write efficiency and high space utilization. Compaction is invoked periodically in LSM-tree based key-value(LSM KV) systems to maintain good system performance. As the size of LSM-KV grows, large-scale compaction is now frequently seen. Compaction throughput significantly degrades with larger inputs, leading to frequent write stalls and decrement in overall write throughput. This paper proposes STEM, a stream-based compaction framework with FPGA to address this issue. A clean-cut algorithm is introduced to enable streaming-based compaction for large-scale data. With a multi-unit pipeline and dynamic pipeline schedule, STEM can handle large-scale compaction tasks efficiently. Based on the experiment result, the compaction throughput of STEM can achieve $27\times$ on average and up to $35\times$ improvement compared with the current RocksDB compaction, $2.09\times$ to $2.27\times$ improvement compared with the state-of-the-art FPGA accelerator.**

***Index Terms*—FPGA, LSM KV, Compaction**

I. Introduction

LSM-tree has been widely adopted in modern NoSQL systems such as LevelDB [1], HBase [2], AsterixDB [3] and Cassandra [4]. LSM-tree based key-value stores (LSM KVs) buffer all writes in memory before flushing them to disk. By converting random writes to sequential writes, LSM KVs have exceptional write efficiency. Compaction is an operation in LSM KVs that merges files and removes deleted or overwritten records. It reorganizes data to improve query efficiency and reduce storage space utilization. It is critical in LSM KVs because it ensures optimal system performance. However, with the increasing size of database systems, large-scale compaction poses great challenges to LSM KVs. As the scale of compaction increases, the compaction performance can be significantly reduced, leading to a degradation in system performance.

Over the years, numerous research efforts have been devoted to optimizing the system performance of LSM KVs. Software optimization methods for LSM KVs include the modification to the compaction policy and the design of schedulers [5], [9], [10]. These methods focus on the optimization of the LSM KV system, instead of the compaction operation. The optimization methods that involve compaction policy modification [7], [11] can lead to negative impacts such as increased read amplification. Hardware acceleration for compaction is also widely explored [6], [8], [12] and these methods can effectively improve compaction efficiency. The accelerated compaction significantly enhances the overall performance of the LSM KV system.

Nevertheless, these FPGA-based methods face limitations when dealing with large-scale compaction due to the limitations of FPGA's hardware resources. The challenges of large-scale compaction include the large file size and the number of compaction files. Current FPGA accelerators [6], [8], [12] cannot handle compaction that exceeds the capacity of FPGA DRAM buffers. Moreover, previous work on FPGA acceleration is limited by the number of input files they can handle [6]. Handling a larger number of input files in parallel consumes additional hardware resources, which exceed the FPGA capacity, particularly the LUT resource.

To support the compaction for large-scale input, this paper proposes a novel compaction framework STEM. STEM enables streaming-based compaction for large-scale data, which is further refined by a dynamic pipeline schedule based on careful tuning of the FPGA buffer size. The challenges of streaming-based compaction for large-scale compaction lie in guaranteeing the correctness of compaction results while achieving high efficiency in large-scale compaction within the constraints of hardware resources. A naive method of simply performing compaction in groups without accounting for key range overlaps cannot guarantee the correctness of the results. In addition, splitting large-scale compaction into several merging tasks of variable sizes can introduce pipeline bubbles to a statically pipelined system, which degrades system performance.

This paper introduces a novel method to overcome the limitations of hardware resources, applying a clean-cut al-

*Weilan Wang and Yu Mao share equal corresponding authorship

gorithm to facilitate streaming-based large-scale compaction. The cut boundary is carefully decided to guarantee the correct compaction results. We consider the key ranges of all files and maintain a min-heap to manage them. The initially determined boundary is fine-tuned to ensure the selected blocks fit in the optimal buffer size. The split of compaction files creates opportunities for the parallel processing of multiple inputs. In STEM, a multi-unit pipeline is implemented to fully utilize FPGA's parallelism and support the concurrent processing of different merging tasks. To further enhance the system performance, STEM introduces a dynamic pipeline scheduler to minimize pipeline bubbles. The proposed design starts with a comprehensive analysis of various scenarios, including the full pipeline, pipeline bubbles caused by limited FPGA computing units, and those caused by limited buffer allocation on FPGA. We enumerate parameters that affect the pipeline schedule and identify buffer size as the critical factor for pipeline performance. We determine the mathematical conditions for reducing pipeline bubbles and establish the formula for optimal buffer size. By tuning the buffer size, we effectively improve system efficiency. Moreover, an FPGA compaction engine is implemented in this work to achieve good large-scale compaction efficiency. According to the experiment results, the proposed STEM compaction framework achieves $27\times$ on average and up to $35\times$ improvement, compared with the compaction of vanilla RocksDB. It is $2.09\times$ to $2.27\times$ faster compared with the state-of-the-art FPGA accelerator. In general, we make the following contributions:

- We propose a stream-based processing of large-scale compaction tasks to remove the limitation of compaction input size. We propose a clean-cut algorithm to split the huge size input into streams of data with non-overlapping key range;
- We propose a multi-unit pipeline to fully utilize the parallelism of FPGA;
- We minimize the pipeline bubbles by dynamic pipeline schedule to further enhance the compaction efficiency;
- We implement a compaction engine on FPGA with careful pipeline design. Based on our experiment result, the compaction engine achieves significant improvement compared to the compaction of vanilla RocksDB;
- The performance of STEM is evaluated to demonstrate the improvement of our design compared with RocksDB. We also compare STEM with state-of-the-art related work to verify the effectiveness of handling large-scale compactions.

The rest of the paper is organized as follows. Section II introduces the background of LSM KVs. Section III presents the challenges of large-scale compactions and explains the motivations. The design details of STEM, including the clean-cut algorithm, multi-unit pipeline and dynamic pipeline schedule, are introduced in detail in Section IV. The implementation of the compaction engine on FPGA is presented in Section V. Section VI shows the evaluation result of STEM. Section VII introduces some related work on optimizing compaction and the conclusion is made in Section VIII.

II. Background

A. LSM-tree based Key-Value Store

LSM KVs use log-structured merge tree to efficiently store and retrieve key-value pairs. LSM KVs have exceptional write performance and are widely used in NoSQL storage systems [3], [4], [13]. LSM KVs achieve high write efficiency because they replace random writes with sequential writes. To achieve sequential writes, LSM KVs perform out-of-place updates, which buffers new input in memory instead of overwriting the old entries. When the data buffered in memory exceeds a certain capacity, the in-memory data is flushed to disk sequentially, which improves the write efficiency.

As shown in Figure 1, LSM KVs perform out-of-place updates by holding incoming data in the memtable, an in-memory data structure that stores the most recent data in the database. The buffered data will be then flushed to SST (Sorted Sequence Table) files in the disk when the memtable is full. The read requests to the database first query memtable in memory, and then the SST files on disk. SST files are persistent files storing data in sorted order, enabling binary search for keys. The files are structured to different levels (L_0,L_1,L_2,...), with higher levels having larger sizes. This organization strikes a balance between read amplification and write amplification. Compaction is triggered in LSM KVs when the capacity of a level (L_i) exceeds a certain threshold. It merges files in L_i with files in the deeper level L_{i+1}. Compaction ensures optimal system performance by maintaining the structure of LSM KVs.

The SST files include the following components [14]:

- **Data Block** The sorted key/value pairs in SST files are partitioned into a sequence of data blocks. These blocks are stored at the beginning of the files and optionally compressed.
- **Meta Block** The data blocks are followed by meta blocks, containing meta data of different types. The Index block keeps the offset and length for each data block, which is referred to as BlockHandle in RocksDB. With the BlockHandle, the data block containing a lookup key in SST files can be located. Besides, the 'Compression Dictionary', 'Properties' meta block, 'Range Deletion' meta block, and 'Filter' meta block contain other important information that can be used for file access and processing.
- **Metaindex Block** Metaindex Block contains the BlockHandle for each meta block.
- **Footer** The footer is fixed-length data stored at the end of an SST file. It contains the BlockHandle for the metaindex block and index block. At the end of footer, a magic number is appended to identify the file format and ensure the validity of the files.

RocksDB is a storage engine developed by Facebook and it enjoys great popularity not only in industry but also in academic research [15]–[17]. It provides highly flexible configuration settings for tuning and allows for implementation on

different storage such as SSDs, hard disks, or remote storage [14].

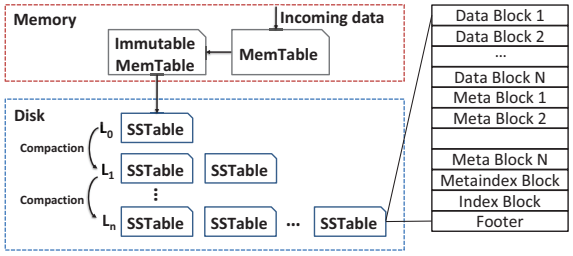


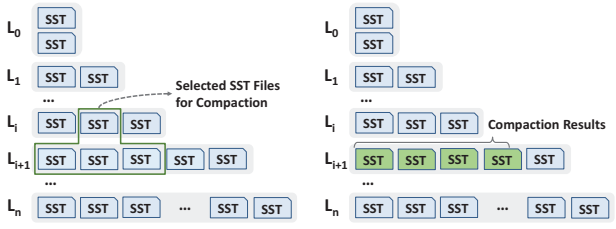
Fig. 1: LSM-tree based key-value store.

B. Compaction in RocksDB

As inserted records in LSM KVs accumulate over time, the obsolete records remain in the system, leading to increased storage space utilization. The query performance drops down since more components are to be examined. To address this issue, compaction is necessary to maintain the optimal performance of system. [14]

In RocksDB, compaction is triggered based on the number of files and size in each level. In L_0 of RocksDB, the compaction is triggered when the number of files is larger than a certain threshold. All the other levels in RocksDB have a target size, which is exponentially increasing for deeper levels. The compaction of the other levels is triggered if the capacity of a level exceeds the target size. As shown in Figure 2a, compaction is triggered in L_i since the file size in this level exceeds the target size. The file of L_i is merged with the files of L_{i+1} and the result is stored in L_{i+1}, as it is shown in Figure 2b. The file size of L_{i+1} increases and compaction will be triggered at this level, merging the files to deeper levels if the file size exceeds the threshold.

The compaction operation is essential for maintaining the key-value store structure and ensures high write efficiency. However, heavy compaction tasks bring some challenges. LSM KVs have problems in processing incoming write requests from users under a heavy compaction workload. This can greatly degrade the performance of the key-value store system and significantly increase the latency for each request response.



(a) Compaction triggered in L_i. (b) Files of L_i merged to L_{i+1}.

Fig. 2: Compaction of SST files in LSM KVs.

III. Motivation

To demonstrate the impact of large input for RocksDB, a preliminary study is conducted to observe how the system's performance changes with increasing compaction input size. The benchmark tool *db_bench* [18] performs *fillrandom* workload with the increasing number of random insert operations as shown in Figure I. Each record consists of a 16-byte key and a 1000-byte value. The experiment is performed on Amazon EC2 F1 instance. We analyze all the compaction and choose the top-5 largest compaction tasks under each input scale. Table I presents the relationship between the number of input records, the size of top-5 largest compaction tasks, and the corresponding system write throughput. Figure 3 illustrates the compaction throughput and system stalls with different input scales. Based on the experiment results, we have the following observations.

TABLE I: Compaction size with increasing input scale.

Input Size (Million Records)	2.5	5	10	25	50	100
Top-5 Compaction Size (GB)	0.56	1.2	1.92	2.58	3.26	4.1
Write Throughput of System (Kops/s)	21.2	17.9	14.2	12.3	7.6	5.8

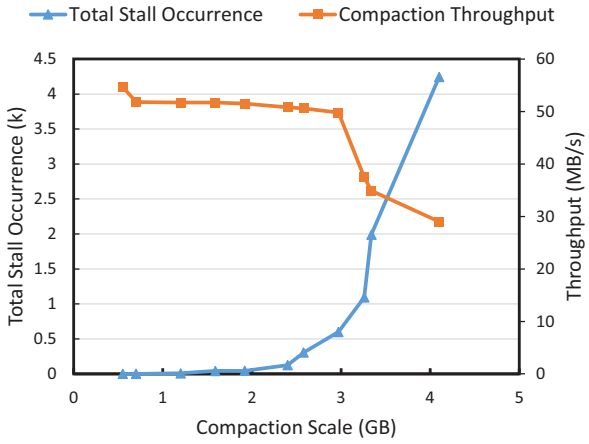


Fig. 3: Compaction throughput and system stall occurrence with increasing number of input records. The number of input records and the corresponding compaction scale is listed in Table I.

Observation 1: Increasing compaction input size will cause significant write stalls.

We observe that increasing compaction input size can lead to significant system write stalls. The write stall refers to the sudden drop in system throughput [15], [17]. Our observation is based on the following experiment results. As shown in Table I, the increasing input scale results in larger compaction tasks. Figure 3 shows that the compaction throughput drops greatly from 53.2MB/s to 29MB/s with the increase of compaction scale. The degradation of compaction efficiency can hinder the write throughput of the system. Table I illustrates

that the system write throughput decreases from 21.2 Kops/s to 5.8 Kops/s. As a result, we can observe significant growth of stall occurrence in Figure 3 from 0 to $4.2k$. In summary, as the number of input records increases, large-scale compaction tasks are triggered and the corresponding compaction efficiency has notable drops. With lower compaction efficiency, the SST files cannot be merged to higher level in a timely manner. The flushing of in-memory immutable memtable is blocked which finally leads to write stalls of the system.

Observation 2: FPGA accelerators face challenges when handling large-scale compaction.

Current FPGA accelerators [6], [8], [12] cannot handle large-scale compaction due to the limited hardware resources. The challenges of large-scale compaction include the file size and number of input files. The compaction input files are first loaded to the DRAM on FPGA before being processed. However, previous works handle all input files in a single compaction, making it impossible to handle compaction tasks if the size of input files exceeds the available DRAM space of FPGA. As shown in Table I, the scale of compaction tasks grows significantly when the input size increases. Therefore, with a large input scale, the size of compaction input can exceed the capacity of FPGA DRAM buffer. The FPGA-based compaction methods face difficulties in handling compaction tasks that are larger than the size of FPGA DRAM buffer.

Moreover, the number of compaction input files presents another challenge for FPGA accelerators. The FPGA accelerators can only process a limited number of input files, due to the restriction of hardware resources. Previous FPGA accelerators [6], [8], [12] use parallel comparers to facilitate efficient key merging. However, when handling more input in parallel, the LUT resource consumption of these accelerators significantly increases. For example, FCAE [6] cannot process the compaction tasks when the number of compaction input files is larger than 9, as the LUT resource consumption exceeds 100%. In FCAE [6], compaction tasks with more than 9 input files are handled by software, which is much slower than FPGA-based methods. As an example, considering the experiment scheme with 100 million input records, more than 14% of the compaction tasks involve more than 9 input files, exceeding the input number limitation of prior studies [6].

In conclusion, large-scale compaction will significantly decrease the system throughput by increasing the occurrence of write stalls. However, conventional FPGA accelerators are unable to process such large-scale compactions.

The challenges posed by the size and number of compaction files, make it infeasible to load all compaction input data into FPGA accelerators. To address these issues, we propose STEM which enables the streaming-based compaction for large-scale data via a clean-cut algorithm. The multi-unit pipeline design further enhances the processing throughput, while a dynamic pipeline schedule minimizes pipeline bubbles to further improve the compaction efficiency.

IV. STEM Design Details

This section introduces the design details of our streaming-based compaction framework STEM. Design principles (Section IV-A) introduce our ideas to tackle the problems observed in Section III. System overview (Section IV-B) demonstrates the architecture and workflow of each module. Then we present our optimization methods in detail. The clean-cut algorithm (Section IV-C) enables the streaming-based processing of large compaction tasks by dividing the input into non-overlapping data blocks with reduced size. The well-designed multi-unit pipeline (Section IV-D) and dynamic pipeline schedule (Section IV-E) improve the compaction efficiency.

A. Design Principles

The principles of our design focus on addressing the issues identified in our observations in Section III. To handle large-scale compaction, our design incorporates a streaming-based processing approach. It breaks the size limit of input files while bounding the memory usage of compaction tasks. Then we maximize the parallelism and minimize the pipeline bubbles of our design to improve the compaction efficiency. In STEM, we have the following goals to achieve in detail:

Streaming-based Processing As mentioned in Section III, FPGA accelerators have difficulties in handling large-scale compaction due to the restriction of hardware resources. The streaming-based processing of compaction is designed to overcome these limitations on compaction input size and number of files by selectively choosing blocks for compaction, instead of the entire SST files. A straightforward streaming of the large compaction input cannot get the correct compaction results. When there are key range overlaps between the streaming groups, the results are invalid. The streaming-based processing of STEM is based on the clean-cut algorithm, which ensures the correctness of each merging loop.

Maximizing the Parallelism of Compaction In a system with large-scale compaction, the compaction efficiency will significantly degrade, resulting in severe write stalls. This can be attributed to the SST files not being merged to higher levels in a timely manner, which blocks the flushing of in-memory immutable memtable [15], [17]. To improve the compaction efficiency, a multi-unit pipeline is proposed in STEM to utilize the parallelism of FPGA.

Minimizing the Pipeline Bubble To further improve the compaction efficiency of STEM, it is crucial to minimize pipeline bubbles during scheduling. To achieve the fully pipelined compaction shown in Figure 5a, the system should avoid pipeline bubbles shown in Figure 5b and 5c. Figure 5b illustrates a pipeline bubble occurring when new blocks are transferred to buffer 1 while it is still occupied by the write operation of previous blocks. This can be attributed to the limited number of buffers. Similarly, in Figure 5c, a pipeline bubble occurs when new blocks are assigned to FPGA computing unit 1 while it is still processing the previous blocks. The limited number of FPGA computing units causes the pipeline bubbles. Adjusting the buffer size on FPGA can

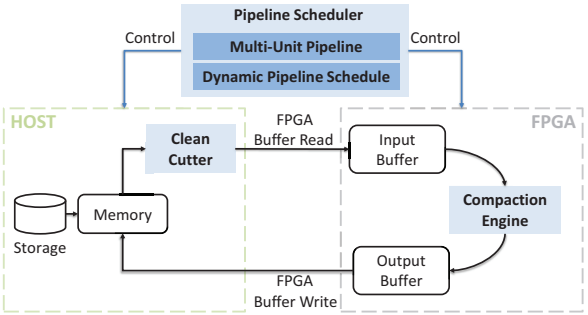


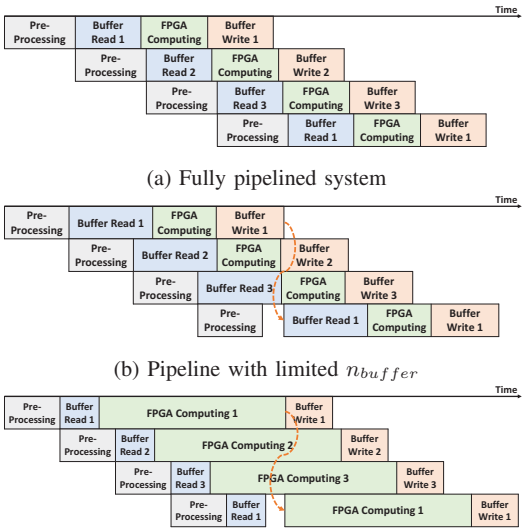
Fig. 4: System overview of STEM. The Clean Cutter is introduced in Section IV-C. The Pipeline Schedule includes the multi-unit pipeline introduced in Section IV-D and dynamic pipeline schedule introduced in Section IV-E. The Compaction Engine is introduced in Section V.

modify the time consumption of each stage and subsequently reduce the pipeline bubbles. However, a pre-defined buffer size cannot effectively address this issue. The pipeline design of large-scale compaction presents challenges due to the varying time consumption of each stage for different merging tasks. To achieve higher compaction efficiency, a dynamic pipeline schedule is essential to minimize pipeline bubbles.

With these design principles, STEM can efficiently handle large-scale compaction. Current FPGA accelerators struggle with large compaction because their designs face the limitation of hardware resources [6], [8], [12]. In contrast, STEM adopts streaming-based processing to overcome resource restrictions. The maximization of parallelism and minimization of pipeline bubbles improve the efficiency of streaming-based processing.

B. System Overview

Figure 4 provides an overview of the FPGA-based compaction system. The SST files are read from disk and Clean Cutter divides these files into non-overlapping blocks. The Clean Cutter algorithm addresses the issue of key range overlap of SST files to ensure the correct compaction results. With the pre-processing of Clean Cutter, FPGA can handle large-scale compaction with limited hardware resources. The selected data blocks are transferred to the input buffer through the PCIe interface and then forwarded to the compaction engine on FPGA. The compaction engine provides high-efficiency compaction and reduces CPU utilization. The compaction results are stored in the output buffer and will be transferred to host memory after the compaction is finished. The data transfer and computing unit execution of the system is coordinated by a well-designed pipeline scheduler. The Multi-Unit Pipeline maximizes compaction efficiency by fully utilizing FPGA parallelism. Additionally, the Dynamic Pipeline Schedule adjusts the size of each merging task to minimize the pipeline bubbles.



(a) Fully pipelined system

(b) Pipeline with limited n_{buffer}

(c) Pipeline with limited n_{unit}

Fig. 5: Pipeline schedule design. The pipeline of STEM comprises pre-processing, buffer read/write, and FPGA computing unit execution. Buffer Read refers to the data transfer from the host to FPGA memory, while Buffer Write refers to the data transfer from FPGA memory to the host. Full pipeline execution is ideal for optimized system performance(see 5a), but limited FPGA buffers (see 5b) or computing units (see 5c) can introduce pipeline bubbles, which degrade the efficiency.

C. Clean-Cut Algorithm

As previously discussed, the increasing scale of compaction brings two challenges. First, the number of files to be processed may surpass the maximum input number that our FPGA engine can handle. Second, the file size may exceed the available space of one buffer of FPGA, which is 4GB in our selected FPGA card Xilinx Virtex Plus UltraScale XCVU9P. Given these hardware resource constraints, compaction becomes a significant challenge.

Simply dividing the large compaction task into portions for compaction cannot guarantee the correctness of the results. A naive approach could cause the files selected for compaction to overlap in key ranges with the remaining files. If the output of one merging loop overlaps with another, the merging results are invalid. To ensure the correctness of streaming-based compaction, the key selection step should be carefully designed in the algorithm.

We propose a clean-cut algorithm to address the issue of key range overlap. As shown by Figure 6, the clean-cut algorithm cuts the input files at the block level, and the large-scale files are split into non-overlapping blocks with limited sizes. It enables handling large-scale compaction on FPGA and ensures the correctness of results.

The clean-cut begins by obtaining the key range of each SST file. The largest keys of each file are selected and maintained by a min-heap $heap_k$. As it is shown in Algorithm 1, the value

Algorithm 1 Clean-Cut Algorithm.

Input: n SST input files;
Output: blocks within a certain key range;

1: Initialize buffer size B_0 to $\frac{1}{n}$ of the maximum buffer size $\frac{1}{n}B_{max}$;
2: Obtain the key range of each SST file , build a min-heap $heap_k$ for the largest keys;
3: **while** file list is empty **do**
4: Obtain the top element k_{min} from min-heap $heap_k$;
5: Set the boundary of clean-cut $k_{bound} = k_{min}$;
6: Adjust the boundary k_{bound} according to the optimal buffer size B_i;
7: Do compaction for blocks within the boundary;
8: Update the optimal buffer size, derive B_{i+1} using Algorithm 2 ;
9: **end while**

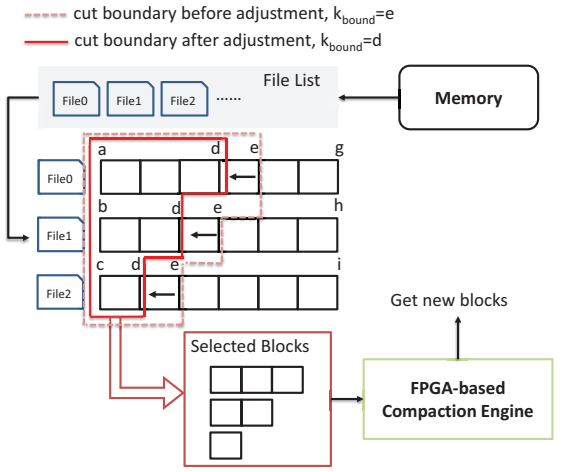


Fig. 6: Clean-cut algorithm. The cut boundary is adjusted according to the buffer size

of clean-cut boundary k_{bound} is set to the top element of $heap_k$ k_{min} and the blocks within the boundary are selected as the compaction input. With clean-cut at block level, the keys in selected blocks are less than k_{bound}, while the remaining keys are larger than k_{bound}. The compaction result will not have an overlapping key range with the rest of the input, which guarantees compaction correctness.

While determining the clean-cut boundary using the min-heap ensures the correctness of compaction, the dynamic adjustment of k_{bound} after obtaining it from the top element of the min-heap improves the overall efficiency of STEM. The boundary is adjusted to ensure the size of the merging loop does not exceed the optimal buffer size B. The steps of determining the optimal buffer size are introduced in Section IV-E. As shown in Figure 6, the boundary is adjusted from k_{min} =e to k_{min} =d to reduce the size of selected blocks.

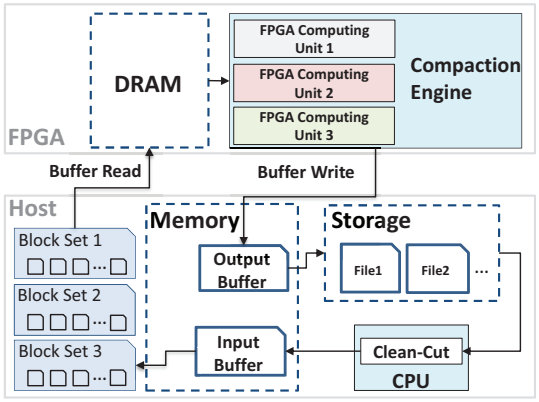


Fig. 7: System with pipeline design.

D. Multi-Unit Pipeline Design

STEM enhances the compaction efficiency by a delicate multi-unit pipeline design. The complete workflow starts with the clean-cut job on the CPU, which selects a portion of the compaction input. The selected blocks are sent from the host memory to the DRAM of FPGA. A memory buffer of DRAM is allocated for each SST file that contains selected blocks. As shown in Figure 7, the blocks selected by one clean-cut form a 'Block Set', which will be transferred to the input buffer on FPGA DRAM by buffer read.

After the data transmission between host memory and FPGA's DRAM, the computing units on FPGA start to merge these selected blocks and place the output in another allocated buffer of the DRAM (named the output buffer).

To enable pipeline execution, the input buffers for n 'Block Set' are allocated for the results of n clean-cut operations. With more input buffers, the compaction for n 'Block Set' can be pipelined. Ideally, a fully-pipelined merging loop should follow the manner as shown in Figure 5a, after transferring all data in the output buffer to the host (the end of Buffer Write 1), a new round of clean-cut is already finished and start to transfer new input data to buffer on FPGA (the start of next Buffer Read 1).

The multi-unit pipeline design generally coordinates the clean-cut on CPU, data transfer, and computing unit execution on FPGA to improve the system performance. To utilize the parallelism of FPGA, multiple FPGA computing units are applied to enable the concurrent execution of compaction. Each FPGA computing unit handles the compaction of one 'Block Set' and the concurrent execution of multiple FPGA computing units further enhances the efficiency.

E. Dynamic Pipeline Schedule Design

The pipeline design of large-scale compaction is challenging due to the varying time consumption of each stage. A dynamic pipeline schedule is essential for higher compaction efficiency. To achieve the fully-pipelined system in Figure 5a, the input size of each merging loop, which is controlled by the buffer size of each clean-cut, should be carefully determined. In

some cases, the pipeline will stall in some stages, as shown in Figure 5b and 5c, degrading the system efficiency. Take the pipeline design of Figure 5b as an example. Buffer Read 1 can not start data transfer until Buffer Write 1 is finished. The pipeline bubbles occur due to the buffer locking that prevents data transfer conflicts. In contrast, when the buffer size becomes larger and the computing unit execution on FPGA takes longer, the stall occurs as shown in Figure 5c. Pipeline bubbles appear when the data blocks are cut and transferred to the FPGA buffer, FPGA Computing Unit 1 cannot handle the input until the compaction of previous data blocks is finished. In addition to the pipeline design itself, the data distribution will also influence the execution of clean-cut algorithm and processing on FPGA computing unit. For example, when the input files are heavily overlapped with each other, the clean-cut algorithm will adjust boundaries more frequently. At the same time, the FPGA computing units need more time to sort and merge the incoming data. This makes it critical to dynamically adjust the pipeline on each merging loop.

TABLE II: Definition of parameters.

Parameter	Definition
n_{unit}	Number of available computing units on FPGA
n_{buffer}	Number of available buffers on FPGA
t_{cut}	Time consumption of clean-cut
t_{unit}[i]	Time consumption of FPGA execution for Block Set i
t_{read}[i]	Time consumption of buffer read of Block Set i
t_{write}[i]	Time consumption of buffer write of Block Set i
T_{read}	Throughput of buffer read
T_{write}	Throughput of buffer write
T_{unit}	Throughput of FPGA computing unit
B	Buffer Size of Each Compaction
S	Total size of Compaction

$$n_{buffer} \times t_{cut} \geq t_{unit} + t_{read} + t_{write} \tag{1}$$

$$n_{unit} \times t_{cut} \geq t_{unit} \tag{2}$$

$$t_{unit} = \frac{B}{T_{unit}}, t_{read} = \frac{B}{T_{read}}, t_{write} = \frac{B}{T_{write}} \tag{3}$$

Algorithm 2 Dynamic Buffer Size.

Input: Optimal buffer size B_i
Output: Updated optimal buffer size B_{i+1}
1: Obtain time consumption t_{cut},t_{unit},t_{read},t_{write};
2: $b_1 = \frac{n_{buffer} \cdot t_{cut}}{\frac{t_{read}}{B_i} + \frac{t_{write}}{B_i} + \frac{t_{unit}}{B_i}}$;
3: $b_2 = n_{unit} \cdot t_{cut} \cdot \frac{B_i}{t_{unit}}$;
4: $B_{i+1} \rightarrow min(b_1, b_2)$

The buffer size of the clean-cut algorithm determines the input size of each merging loop. In STEM, we tune the buffer size for better merging efficiency. For each iteration, we choose the buffer size according to the following rules: 1) it should be as large as possible to reduce the transmission round, and 2) the duration of the next pipeline bubbles should be as short as possible to avoid low resource utilization.

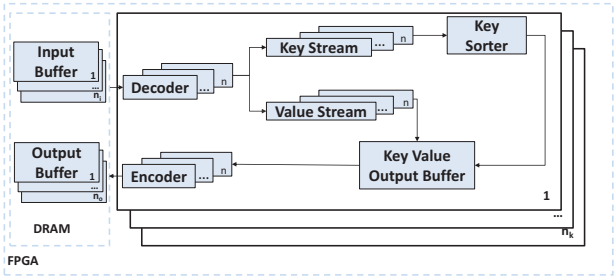


Fig. 8: Implementation of STEM compaction engine.

We quantify the pipeline bubbles using the terms defined in Table II. The buffer size should satisfy the condition in Equation 1 to avoid the conflicts between buffers shown in Figure 5b and the condition in Equation 2 to avoid the conflicts between FPGA computing units shown in Figure 5c. From Equation 1,2 and 3, we derive the condition of buffer size that can ensure the fully pipelined design of system in Equation 4. Equation 4 shows that to achieve the ideal pipeline, the buffer size B has an upper limit.

In our design, the buffer size is dynamically adjusted for each compaction using Equation 4. The buffer size is set to approach the upper limit for optimal performance. It's important to ensure that the buffer size for each compaction is larger than a certain threshold to ensure the high performance of data transfer. Additionally, the total time cost for a fully pipelined system can be approximated by the count of clean-cut multiplied by the time cost of each cut operation $N \times t_{cut} = \frac{S}{B} \times t_{cut}$. To reduce the time consumption, the buffer size B should be as large as possible.

$$B \leq \min(\frac{n_{buffer} t_{cut}}{\frac{1}{T_{unit}} + \frac{1}{T_{read}} + \frac{1}{T_{write}}}, n_{unit} t_{cut} T_{unit}) \tag{4}$$

In conclusion, the dynamic pipeline schedule of STEM is implemented as follows. The initial buffer size will be determined by empirical studies. The impact of different initial buffer size is shown in Section VI-D. The number of buffers and FPGA computing units will be allocated as many as possible according to the available hardware resources. After each merging loop, the pipeline schedule will be adjusted by reassigning a new buffer size according to Algorithm 2. The next merging loop continues to merge input blocks with the new buffer size.

V. Implementation

In this section, we present the implementation details of the STEM compaction engine. As shown in Figure8, the compaction engine of STEM breaks down the sort and merge process into multiple computation stages. These stages are pipelined using data flow optimization to ensure high efficiency. The large-scale compaction input is split into non-

overlapping blocks using the clean-cut algorithm for stream-based processing. Every two streams of blocks are combined into one. There are eight streams of merging process, which is the maximum number of parallelism supported in one FPGA computing unit. The degree of parallelism is limited by the hardware resource. When more inputs are being processed in parallel, more hardware resources are required on FPGA to support the related operations, as shown in SectionVI-E. The size of the input buffer is dynamically adjusted to reduce the pipeline bubbles. The input is selected by a clean-cut algorithm the size of the blocks fits in the buffer available on FPGA. Compared with the related work, STEM can perform large compaction tasks with limited hardware resources.

Each stream has three major modules: Decoder, Key Sorter, and Encoder. The design of each module is presented in the following.

Decoder The large-scale compaction input is split into non-overlapping blocks, which are sent to the input buffer on FPGA through the PCIe interface. As shown in Figure 8, the compaction engine starts with the Decoder extracting key and value from input blocks. The read of key and value pairs from data streams and decoding can be executed in parallel by using ping-pong buffers in the design. The Decoder pre-fetches one entry from data streams and keeps it in $buffer_a$. Then the entry in $buffer_a$ is decoded to key and value, while the new entry is read and kept in $buffer_b$. Similarly, the read of the new entry and decoding of the entry in $buffer_b$ are run in parallel in the next step.

The key and value are separated into two streams after decoding for parallel processing. To fully utilize the parallelism, each input is handled independently by its own Decoder.

Key Sorter The separated key streams from the decoder are sent to the Key Sorter module and each Key Sorter module works independently. The keys from streams are compared to select the smaller one and flags indicating the choice of keys are sent to the value output buffer. The corresponding value indicated by the flag is sent to the output.

Encoder The Encoder module encodes the merged key and value records from the output buffer to data blocks. Similarly, ping-pong buffers are applied to improve the encoding efficiency. The Encoder pre-fetches one key-value pair from data streams to Buffer_1. When the key/value pair in Buffer_1 is encoded, another key-value pair is read to Buffer_2 simultaneously. After sorting, the separated key/value pairs are combined into data blocks following RocksDB's format.

After processing the Decoder, Key Sorter, and Encoder, the key-value pairs in the split data blocks are sorted and kept in the output buffer of FPGA DRAM. The sorted output is sent to the memory of the host machine through the PCIe interface.

VI. Evaluation

This section first presents the experiment setup, then evaluates STEM's compaction throughput, compared with the related works and the vanilla RocksDB. We also evaluate the overall system performance of RocksDB integrated with STEM, compared with the vanilla RocksDB. Sensitivity studies are included to discuss varying key length and value length, different parallelism on FPGA, and different initial buffer sizes in dynamic pipeline schedules. At the end of this section, we present the utilization of hardware resources.

A. Experiment Setup

The proposed compaction framework STEM is evaluated on Amazon EC2 F1 instance, a cloud server provided by Amazon Web Service. The evaluation targets the Xilinx Virtex Plus UltraScale XCVU9P platform, with a working frequency of 200 MHz. This board has a 64GB DDR4 DRAM and PCIe gen3 x 16 interface. The proposed STEM is developed on the Vitis Unified Platform (2021.2). The CPU of the Amazon cloud server is an Intel Xeon E5-2686 processor (2.3 GHz). We implement STEM on RocksDB v7.10.0 and use its embedded *db_bench* to evaluate the system.

B. Evaluation on STEM

This subsection presents the performance evaluation of STEM. The compaction throughput of STEM is compared with the related works and vanilla RocksDB. The key length is set to 16 byte and the value length is set to 1024 byte for evaluation. The performance gains from implementing a dynamic pipeline are assessed by comparing pipeline bubble duration with that of a static pipeline.

Performance Comparison. The compaction throughput of STEM is compared with the state-of-the-art works and vanilla RocksDB. FCAE [6] is chosen as our comparison target since it provides hardware design details. We made our best efforts to reproduce FCAE for our comparison. The comparison of STEM and FCAE is based on experimental results obtained under the same execution environment. The comparison with Zhang et al. [12] is based on the results inferred from the paper. We believe that FCAE and Zhang et al. [12] have comparable performance as they share similar design principles. Since Zhang et al. [12]'s approach is implemented on the X-engine [8] with an optimized FPGA compaction engine, we regard X-engine as a component of Zhang et al. [12].

As shown in Figure 9, the compaction throughput of STEM is compared with one of the state-of-the-art works FCAE [6] and vanilla RocksDB. We compare the throughput with FCAE only when the total input size is smaller than 4GB. Due to the limited size of the FPGA DRAM buffer, FCAE has an Out-of-memory (OOM) problem with larger inputs. For compaction input larger than 4GB, the performance of STEM is compared with the compaction of the vanilla RocksDB. By comparison, we achieve $2.09\times$ to $2.27\times$ higher compaction throughput compared with FCAE [6], and up to $35\times$ faster than vanilla RocksDB at any compaction input size.

The comparison with the Zhang et al. [12] is shown in Table III. STEM and related works operated at a working frequency of 200 MHz. The compaction throughput of STEM can achieve 1.27 to 1.61 GB/s, outperforming other works. FCAE [6] and Zhang et al. [12] cannot handle large compaction tasks due to the limitations of hardware resources, as

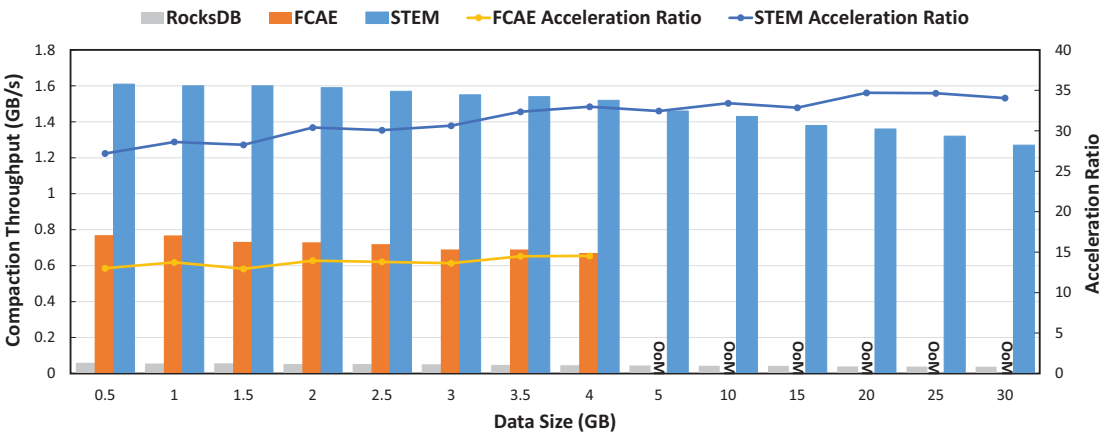


Fig. 9: Compaction throughput comparison. The compaction throughput of STEM is compared with the vanilla RocksDB and the state-of-the-art FPGA accelerator FCAE [6].

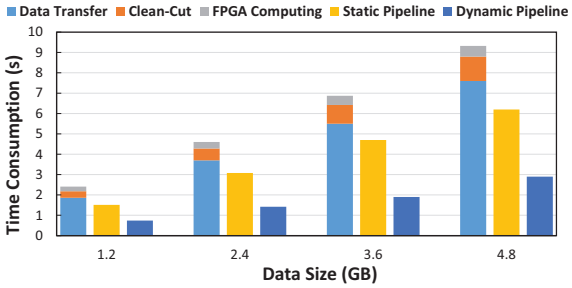
TABLE III: STEM Comparison with Related Works.

	STEM	FCAE [6]	Zhang et al. [12]
Frequency	200 MHz	200 MHz	200 MHz
Compaction Throughput	1.27-1.61 GB/s	0.67-0.77 GB/s	0.85 GB/s
Maximum Compaction Size	Not limited	Limited by hardware resource	Limited by hardware resource
BRAM Resource Utilization	39%	25%	38%
LUT Resource Utilization	65%	84%	26%
Flip-Flop Resource Utilization	28%	14%	10%

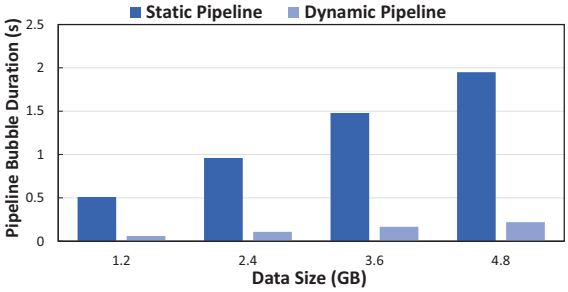
discussed in Observation 2 of Section III. With streaming-based processing and a well-designed pipeline scheduler, STEM can efficiently handle large compaction without resource limitation. As shown in Figure 9, STEM can handle compaction tasks as large as 30GB, while FCAE cannot handle compaction over 4GB due to the limited FPGA buffer size. STEM and FCAE use the same FPGA Xilinx Virtex Plus UltraScale XCVU9P.

Improvement Breakdown. Pipeline schedule design improves the efficiency of the FPGA-based compaction. The contribution of the pipeline schedule design is evaluated by removing the pipeline from STEM. The time consumption breakdown of the STEM without pipeline is shown in Figure 10a. The time consumption is measured for three main modules, including the clean-cut, data transfer, and FPGA computing. Without a pipeline schedule design, these modules are running in sequential order. The time consumption of compaction equals the sum of these modules' time consumption. The size of compaction input varies from 1.2GB to 4.8GB. The experiment results show that data transfer takes the dominant portion of time consumption. It takes 1.86s at 1.2GB and 7.6s at 4.8GB to transfer the data between the host and FPGA. The compaction efficiency of STEM with a static pipeline can achieve 1.46× to 1.6× improvement compared with the STEM without any pipeline design.

To evaluate the improvement brought by the dynamic



(a) Compaction time of different pipeline schedule design



(b) Pipeline bubble duration of static and dynamic pipeline

Fig. 10: Benefits of dynamic pipeline design

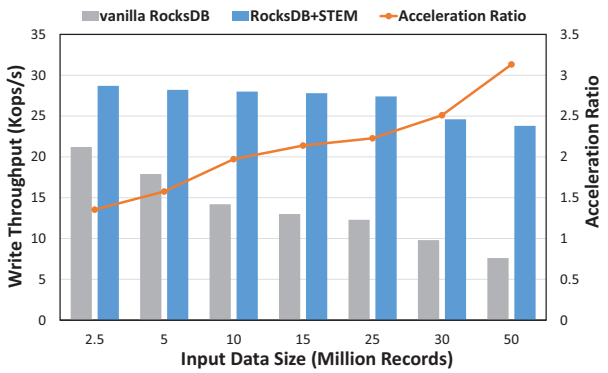


Fig. 11: Comparison of system write throughput.

pipeline, we compare the compaction efficiency of STEM with dynamic and static pipeline respectively. As shown in Figure 10a, STEM with a dynamic pipeline schedule achieves 2.04× to 2.47× improvement over the static pipeline version. The improvement can be attributed to reduced pipeline bubbles. As shown in Figure 10b, with a dynamic pipeline schedule, the pipeline bubble duration is reduced by around 89%.

C. Overall System Performance

System Write Throughput. Figure 11 illustrates system performance evaluations, with write throughput as the metric. The experimental setup is the same as the motivation section. The experiment examines how the write throughput of RocksDB with STEM, changes with increasing input scale compared to the vanilla RocksDB. The write throughput of vanilla RocksDB experiences significant degradation with increasing input scale. As mentioned in the **Observation 1** of the motivation section, the degradation can be attributed to the SST files not being merged to higher levels in a timely manner when the compaction efficiency drops under large-scale input. As a result, the flushing of in-memory immutable memtable is blocked, finally leading to system write stalls. The RocksDB with STEM has a less significant decrease in write throughput than the vanilla RocksDB. The write throughput of RocksDB with STEM is 1.35× to 3.13× faster than the vanilla RocksDB. This improvement benefits from the high-efficiency compaction provided by STEM, which can effectively resolve the bottleneck. We observe a notable rise in the acceleration ratio as the input scale increases. The issue posed by heavy compaction tasks is less critical at smaller scales. However, the problem becomes more severe with the increase of input scale, and the advantage of RocksDB with STEM over the vanilla RocksDB becomes more substantial.

System Resource Utilization. Figure 12 presents the evaluation of CPU utilization. With a large input scale, vanilla RocksDB experiences heavy compaction, which is indicated by CPU utilization approaching 100% when the input size is larger than 25 million records. In comparison, the CPU utilization of RocksDB with STEM is much lower than vanilla RocksDB. The offloading of heavy compaction tasks to FPGA

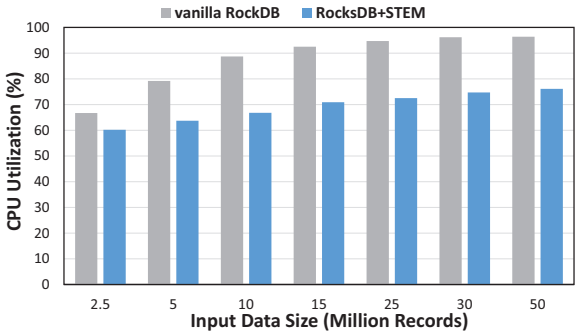


Fig. 12: CPU utilization.

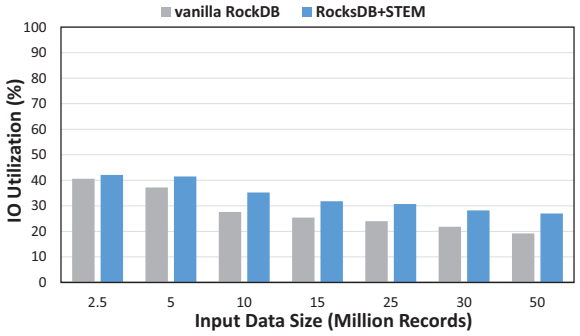


Fig. 13: I/O utilization.

effectively mitigates the challenges brought by compactions, improving efficiency with lower CPU utilization.

The results in Figure 13 show that the I/O utilization of vanilla RocksDB drops from 40.6% to 19.2%, while the utilization of RocksDB with STEM drops from 42% to 27%. Since the vanilla RocksDB is CPU-bounded when performing compaction tasks, the I/O bandwidth is not fully utilized. While STEM reduces the CPU utilization of RocksDB, it comes with the trade-off of increased I/O usage.

Figure 14 presents that the memory budget of both vanilla RocksDB and RocksDB with STEM shows a notable increase. This is due to the growing compaction size when the system handles large-scale input, as shown in Table I.

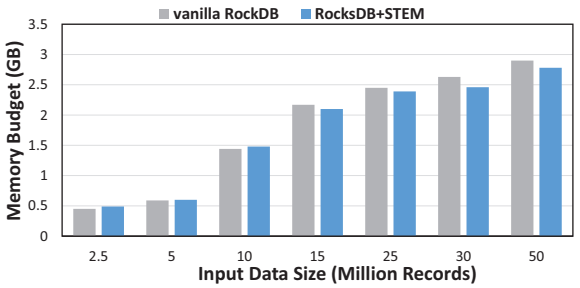


Fig. 14: Memory Budget.

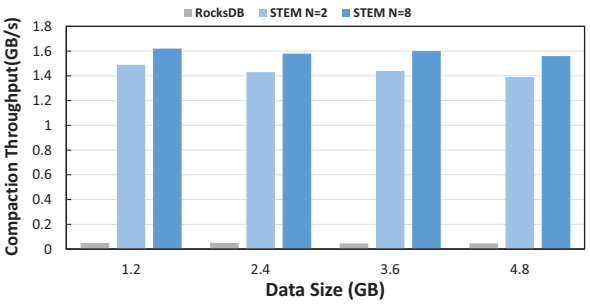


Fig. 15: Compaction performance with different hardware configurations. N refers to the degree of parallelism of FPGA

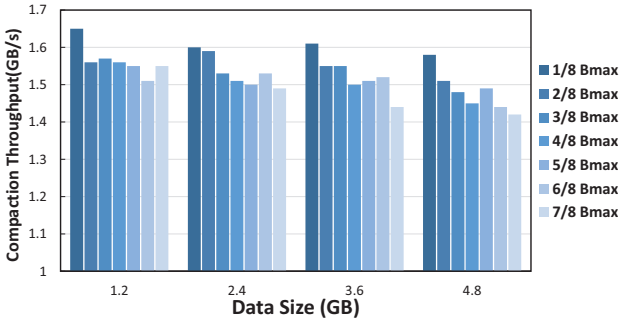


Fig. 16: Impact of different initial buffer size. B_{max} refers to the maximum buffer size available on FPGA

D. Sensitivity Study

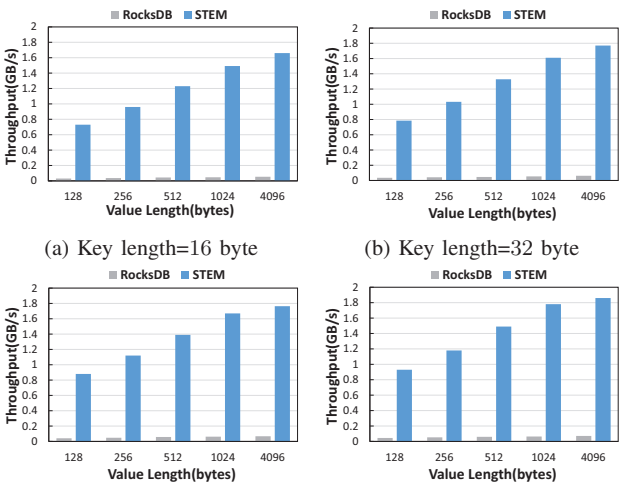
Performance with Different Key and Value Size. We evaluate STEM's compaction throughput across different key and value lengths. In sub-figures of Figure 17, the key length is set to 16 byte, 32 byte, 64 byte and 128 byte, respectively. Within each sub-figure, we demonstrate how compaction throughput is affected when the value length ranges from 64 bytes to 4096 bytes.

Within each sub-figure of Figure 17, the compaction throughput increases with a larger value length. The improvement can be attributed to the compaction engine processing an increased file size every second. The compaction throughput is equal to the handled file size divided by the time consumed, which includes the time costs of data transfer and computing on the FPGA compaction engine. The bottleneck of the FPGA compaction engine is the key sorter, and the key length determines the latency. Given that the key length is fixed, the number of entries handled per second remains consistent, when the value length increases. The improvement in throughput is evident as the file size processed per second can be estimated by multiplying the length of the entries by their total count.

We also observe the improvement of compaction throughput with the increase in key length. The key sorter has higher latency with a larger key length, and the compaction engine handles fewer entries per second. Despite this, compaction throughput is slightly improved because the impact of the increased entry length is more significant.

Performance with Different Hardware Configurations. The performance of STEM with different hardware configurations is evaluated in this part. As shown in Figure 15, the input number of compaction engine STEM is set to be $N = 2$ and $N = 8$. The performance of STEM is $28.6\times$ to $32\times$ compared with the vanilla RocksDB when $N = 2$. The improvement is $31.6\times$ to $35.5\times$ when $N = 8$. The compaction engine with $N = 8$ has better compaction throughput because it has higher parallelism.

Performance with Different Initial Buffer Size. Figure 16 demonstrates the performance of STEM with different initial buffer sizes. As mentioned in Section IV-E, the optimal buffer size is determined by the time cost of clean-cut, throughput of data transfer, and FPGA computing unit processing. However, at the beginning stage when these parameters are unavailable, the initial buffer size is set to some portion of the maximum buffer size available on FPGA. The initial buffer size is set to $1/8$,$2/8$,$3/8$,$4/8$,$5/8$,$6/8$, $7/8$ B_{max}, where B_{max} is the maximum buffer size on FPGA. The experimental results show that dynamic pipeline schedules with different buffer sizes have similar performance. In general, the change of initial buffer size in dynamic pipeline schedule design does not affect the system performance. As long as the size is not too small, leading to the degradation of PCIe transfer, the buffer size can be adjusted to the optimal value after several iterations.



(a) Key length=16 byte (b) Key length=32 byte

(c) Key length=64 byte (d) Key length=128 byte

Fig. 17: Compaction throughput with different key length and value length

E. Hardware Resource Consumption

The hardware resource consumption of the FPGA compaction engine with different numbers of parallelism is shown in Table IV. The LUT resource has the highest utilization. The usage of LUT achieves 64.9% when the compaction engine on FPGA has input number $N = 8$. When the number of parallelism on FPGA increases, more hardware resources

TABLE IV: Hardware Resource Cost.

Parallelism	BRAM	DSP	FF	LUT
$N = 2$	19.8%	3.2%	13.0%	51.8%
$N = 4$	26.9%	3.9%	23.4%	58.2%
$N = 8$	39.2%	5.2%	28.2%	64.9%

are needed to support the compaction engine on FPGA. To process more input in parallel, more Decoders, Key Sorters, and Encoders are needed on FPGA to support the related operations. The increased number of modules results in higher hardware usage.

F. Performance Improvements

Based on the evaluation of STEM in handling large-scale compaction input, we can observe that STEM significantly improves the compaction of LSM KVs. The evaluation shows that:

1) STEM can achieve up to $35\times$ improvement compared with the compaction of vanilla RocksDB. The compaction throughput of STEM also outperforms the related work. Compared with the state-of-the-art FPGA accelerator [6], STEM achieves $2.09\times$ to $2.27\times$ improvement. (Section VI-B)
2) STEM with dynamic pipeline schedule can achieve $2.04\times$ to $2.47\times$ improvement compared with the static pipeline version. The compaction efficiency improvement can be attributed to the reduced pipeline stall with a dynamic pipeline schedule. (Section VI-B)
3) Compared with vanilla RocksDB, RocksDB with STEM achieves up to $3.1\times$ improvement in write throughput with lower CPU utilization. (Section VI-C)
4) STEM is insensitive to the changes in parameters. It significantly improves compaction regardless of variations in value size, parallelism of FPGA computing units, and initial buffer size. (Section VI-D)

VII. Related Work

LSM KV has been widely applied due to its outstanding performance and has been a subject of intensive study in the past decades [16], [19], [20]. Compaction is a critical issue in LSM KVs for maintaining high performance. The optimization of compaction is widely explored [17], [21]–[23].

Some optimizations focus on software optimization, including lower compaction frequency and the design of compaction schedule design. Pradeep Shetty et al. propose a novel stitching method on LSM KVs to avoid unnecessary data copy [5]. dCompaction [7] presents a new compaction scheme called delayed compaction for reducing the amount of I/O involved in maintaining an LSM-tree. Block Compaction only reads those data blocks involved by a compaction operation, which reduces the write I/Os and handles the block-cache invalidation problem. [9]. bLSM [10] proposes a merge scheduler that bounds the write latency without degrading the throughput. [28] proposes the pipelined compaction that divides one compaction procedure into multiple sub-tasks and is accelerated by parallel implementation of sub-tasks. [29] proposes light-weight compaction, which divides the tables into segments according to the key range of overlapped tables in the next level. In [30], compaction actions of LSM KVs is triggered by lower-level data, which can reduce the compaction granularity for smaller tail latency and alleviate the write amplification problem.

The distinctive features of some special storage devices make it possible to propose new compaction designs. [31] proposes the SLM-DB on persistent memory (PM) and takes advantage of both the B+ tree and LSM-tree. [32] proposes NVLSM that leverages byte-addressable Non-Volatile Memory to assign pointers to key-value pairs, thereby enabling optimized migration that reduces latency and write amplification for write-intensive workloads. [34] optimizes LevelDB by using the multiple channels of an SSD for higher parallelism. FlashKV [33] is an optimized key-value store for open-channel SSDs that enhances performance by directly controlling raw flash devices at the application layer and exploiting the internal parallelism of the device.

However, these optimizations cannot improve the compaction process itself. Besides, heavy compaction workloads can still occupy the CPU resources. To handle these issues, hardware acceleration can be a promising solution. Hardware acceleration for databases has been widely explored, including GPU-based acceleration [24], [25] and FPGA-based acceleration [26], [27]. [8] introduces a suite of optimizations to improve compaction including a refined LSM-tree data structure, FPGA-accelerated compaction, asynchronous writes in transactions, multi-staged pipelines, and multi-version metadata index. [12] implements a three-stage well-designed pipelined compaction on FPGA with an asynchronous scheduler and integrates it with X-Engine. [6] design a compaction engine on FPGA to accelerate the offloaded compaction work of LevelDB [1].

VIII. Conclusion

This paper identifies the significant degradation of RocksDB compaction under a large write workload, which results in severe system stalls. With lower compaction efficiency, the SST files cannot be merged to higher levels in a timely manner, which blocks the flushing of immutable memtable, leading to write stalls. To deal with these matters, this paper proposes a stream-based compaction framework and implements it on RocksDB. A clean-cut algorithm is applied with the compaction engine to enable streaming-based large-scale compaction with high efficiency. Multi-unit pipeline and dynamic pipeline schedule further improve the working efficiency. The experiment result shows that the proposed STEM compaction framework achieves $27\times$ on average and up to $35\times$ improvement, compared with the compaction of vanilla RocksDB.

IX. Acknowledgement

The work described in this paper was supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CityU 11209122).

REFERENCES

[1] Jeff Dean and Sanjay Ghemawat, LevelDB, https://github.com/google/leveldb, 2021.

[2] The Apache Software Foundation, Apache Hbase, https://hbase.apache.org/, 2023.

[3] Alsubaiee, Sattam and Altowim, Yasser and Altwaijry, Hotham and Behm, Alexander and Borkar, Vinayak and Bu, Yingyi and Carey, Michael and Cetindil, Inci and Cheelangi, Madhusudan and Faraaz, Khurram and others, "AsterixDB: A scalable, open source BDMS", *arXiv preprint arXiv:1407.0454*, 2014

[4] The Apache Software Foundation, Apache Cassandra, https://cassandra.apache.org/, 2023.

[5] Shetty, Pradeep J and Spillane, Richard P and Malpani, Ravikant R and Andrews, Binesh and Seyster, Justin and Zadok, Erez, "Building workload-independent storage with VT-trees", *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pp.17–30, 2013.

[6] Sun, Xuan and Yu, Jinghuan and Zhou, Zimeng and Xue, Chun Jason, "Fpga-based compaction engine for accelerating lsm-tree key-value stores", *2020 36th International Conference on Data Engineering (ICDE)*, pp.1261–1272, 2020.

[7] Pan, Feng-Feng and Yue, Yin-Liang and Xiong, Jin, "dcompaction: Speeding up compaction of the lsm-tree via delayed compaction", *Journal of Computer Science and Technology*, vol.32, pp.41–54, 2017.

[8] Huang, Gui and Cheng, Xuntao and Wang, Jianying and Wang, Yujie and He, Dengcheng and Zhang, Tieying and Li, Feifei and Wang, Sheng and Cao, Wei and Li, Qiang, "X-Engine: An optimized storage engine for large-scale E-commerce transaction processing", *Proceedings of the 2019 International Conference on Management of Data*, pp.651–665, 2019.

[9] Wang, Xiaoliang and Jin, Peiquan and Hua, Bei and Long, Hai and Huang, Wei, "Reducing Write Amplification of LSM-Tree with Block-Grained Compaction", *2022 38th International Conference on Data Engineering (ICDE)*, pp.3119–3131, 2022

[10] Sears, Russell and Ramakrishnan, Raghu, "bLSM: a general purpose log structured merge tree", *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp.217–228, 2012

[11] Dayan, Niv and Idreos, Stratos, "Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging", *Proceedings of the 2018 International Conference on Management of Data*, pp.505–520,2018.

[12] Teng Zhang and Jianying Wang and Xuntao Cheng and Hao Xu and Nanlong Yu and Gui Huang and Tieying Zhang and Dengcheng He and Feifei Li and Wei Cao and Zhongdong Huang and Jianling Sun, "FPGA-Accelerated Compactions for LSM-based Key-Value Store", *FAST*, pp.225–237, 2020

[13] Chang, Fay and Dean, Jeffrey and Ghemawat, Sanjay and Hsieh, Wilson C and Wallach, Deborah A and Burrows, Mike and Chandra, Tushar and Fikes, Andrew and Gruber, Robert E, "Bigtable: A distributed storage system for structured data", *ACM Transactions on Computer Systems (TOCS)*, vol.26, pp.1–26, 2008

[14] Facebook, RocksDB Wiki, https://github.com/facebook/rocksdb/wiki, 2022

[15] Yu, Jinghuan and Noh, Sam H and Choi, Young-ri and Xue, Chun Jason, "ADOC: Automatically Harmonizing Dataflow Between Components in Log-Structured Key-Value Stores for Improved Performance", *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pp.65–80, 2023

[16] Chen, Hao and Ruan, Chaoyi and Li, Cheng and Ma, Xiaosong and Xu, Yinlong, "SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage", *FAST*, vol.21, pp.17–32, 2021

[17] Balmau, Oana and Dinu, Florin and Zwaenepoel, Willy and Gupta, Karan and Chandhiramoorthi, Ravishankar and Didona, Diego, "SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores", *USENIX Annual Technical Conference*, pp.753–766, 2019

[18] Facebook, db bench, https://github.com/facebook/rocksdb/blob/main/tools/db_bench_tool.cc, 2023

[19] Sarkar, Subhadeep and Athanassoulis, Manos, "Dissecting, Designing, and Optimizing LSM-based Data Stores", *Proceedings of the 2022 International Conference on Management of Data*, pp.2489–2497, 2022

[20] Mei, Fei and Cao, Qiang and Jiang, Hong and Tintri, Lei Tian, "LSM-tree managed storage for large-scale key-value store", *Proceedings of the 2017 Symposium on Cloud Computing*, pp.142–156, 2017

[21] Li, Jianchuan and Jin, Peiquan and Lin, Yuanjin and Zhao, Ming and Wang, Yi and Guo, Kuankuan, "Elastic and stable compaction for lsm-tree: A faas-based approach on terarkdb", *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, pp.3906–3915, 2021

[22] Balmau, Oana and Didona, Diego and Guerraoui, Rachid and Zwaenepoel, Willy and Yuan, Huapeng and Arora, Aashray and Gupta, Karan and Konka, Pavan, "TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores", *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pp.363–375, 2017

[23] Didona, Diego and Zwaenepoel, Willy, "Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores", *NSDI*, pp.79–94, 2019.

[24] Kaldewey, Tim and Lohman, Guy and Mueller, Rene and Volk, Peter, "GPU join processing revisited", *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, pp.55–62, 2012

[25] Roozmeh, Mehdi and Lavagno, Luciano, "Implementation of a performance optimized database join operation on FPGA-GPU platforms using OpenCL", *2017 Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, pp.1–6, 2017

[26] Kara, Kaan and Giceva, Jana and Alonso, Gustavo, "Fpga-based data partitioning", *Proceedings of the 2017 ACM International Conference on Management of Data*, pp.433–445, 2017

[27] Kara, Kaan and Alonso, Gustavo, "Fast and robust hashing for database operators", *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp.1–4, 2016

[28] Zhang, Zigang and Yue, Yinliang and He, Bingsheng and Xiong, Jin and Chen, Mingyu and Zhang, Lixin and Sun, Ninghui, "Pipelined compaction for the LSM-tree", *2014 28th International Parallel and Distributed Processing Symposium*, pp.777–786, 2014

[29] Yao, Ting and Wan, Jiguang and Huang, Ping and He, Xubin and Gui, Qingxin and Wu, Fei and Xie, Changsheng, "A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores", *Proc. 33rd Int. Conf. Massive Storage Syst. Technol.(MSST)*, pp.1–13, 2017

[30] Chai, Yunpeng and Chai, Yanfeng and Wang, Xin and Wei, Haocheng and Bao, Ning and Liang, Yushi, "LDC: a lower-level driven compaction method to optimize SSD-oriented key-value stores", *2019 35th International Conference on Data Engineering (ICDE)*, pp.722–733, 2019

[31] Olzhas Kaiyrakhmet and Songyi Lee and Beomseok Nam and Sam H. Noh and Young-ri Choi, "SLM-DB: single-level key-value store with persistent memory", *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pp.191–205, 2019

[32] Zhang, Baoquan and Du, David HC, "Nvlsm: A persistent memory key-value store using log-structured merge tree with accumulative compaction", *ACM Transactions on Storage (TOS)*, vol.17, pp.1–26, 2021

[33] Zhang, Jiacheng and Lu, Youyou and Shu, Jiwu and Qin, Xiongjun, "FlashKV: Accelerating KV performance with open-channel SSDs", *ACM Transactions on Embedded Computing Systems (TECS)*, vol.16, pp.1–19, 2017

[34] Wang, Peng and Sun, Guangyu and Jiang, Song and Ouyang, Jian and Lin, Shiding and Zhang, Chen and Cong, Jason, "An efficient design and implementation of LSM-tree based key-value store on open-channel SSD", *Proceedings of the Ninth European Conference on Computer Systems*, pp.1–14, 2014