usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

FPGA-Accelerated Compactions for LSM-based Key-Value Store

Teng Zhang, *Alibaba Group, Alibaba-Zhejiang University Joint Institute of Frontier Technologies, Zhejiang University;* Jianying Wang, Xuntao Cheng, and Hao Xu, *Alibaba Group;* Nanlong Yu, *Alibaba-Zhejiang University Joint Institute of Frontier Technologies, Zhejiang University;* Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, and Wei Cao, *Alibaba Group;* Zhongdong Huang and Jianling Sun, *Alibaba-Zhejiang University Joint Institute of Frontier Technologies, Zhejiang University*

https://www.usenix.org/conference/fast20/presentation/zhang-teng

This paper is included in the Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)
February 25–27, 2020 • Santa Clara, CA, USA
978-1-939133-12-0

Open access to the Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20) is sponsored by
NetApp®

FPGA-Accelerated Compactions for LSM-based Key-Value Store

Teng Zhang[*,†], Jianying Wang[*], Xuntao Cheng[*], Hao Xu[*], Nanlong Yu[†], Gui Huang[*], Tieying Zhang[*], Dengcheng He[*], Feifei Li[*], Wei Cao[*], Zhongdong Huang[†], and Jianling Sun[†]

[*]Alibaba Group
[†]Alibaba-Zhejiang University Joint Institute of Frontier Technologies, Zhejiang University
{jason.zt,beilou.wjy,xuntao.cxt,haoke.xh,qushan, tieying.zhang, dengcheng.hedc,lifeifei,mingsong.cw}@alibaba-inc.com
{yunanlong,hzd, sunjl}@zju.edu.cn

Abstract

Log-Structured Merge Tree (LSM-tree) key-value (KV) stores have been widely deployed in the industry due to its high write efficiency and low costs as a tiered storage. To maintain such advantages, LSM-tree relies on a background compaction operation to merge data records or collect garbages for housekeeping purposes. In this work, we identify that slow compactions jeopardize the system performance due to unchecked oversized levels in the LSM-tree, and resource contentions for the CPU and the I/O. We further find that the rising I/O capabilities of the latest disk storage have pushed compactions to be bounded by CPUs when merging short KVs. This causes both query/transaction processing and background compactions to compete for the bottlenecked CPU resources extensively in an LSM-tree KV store.

In this paper, we propose to offload compactions to FPGAs aiming at accelerating compactions and reducing the CPU bottleneck for storing short KVs. Evaluations have shown that the proposed FPGA-offloading approach accelerates compactions by 2 to 5 times, improves the system throughput by up to 23%, and increases the energy efficiency (number of transactions per watt) by up to 31.7%, compared with the fine-tuned CPU-only baseline. Without loss of generality, we implement our proposal in X-Engine, a latest LSM-tree storage engine.

1 Introduction

Key-value (KV) stores developed based on the Log-Structured Merge Tree (LSM-tree) have emerged as the backbone database storage system serving many applications in the cloud that are sensitive to both the cost and the performance, such as instant messaging, online retail and advertisements. Notable examples include LevelDB [8], BigTable [2], RocksDB [13], Dynamo [9], WiredTiger [35], and X-Engine [16] from various companies. LSM-tree is favoured in these cases because its advantages on the write efficiency and storage costs compensate the shortcomings of the widely adopted Solid State Drives (SSDs) in industrial storages [13]. For example in the online retail context, the underlying database storage is prefered to be able to provide a high write throughput for the placement of new orders, an acceptable read latency for users to query hot items, and a low storage cost for storing all records of the stockpiles, logistics, payments and other business-critical e-commerce details on top of a web-scale storage built upon SSDs [16].

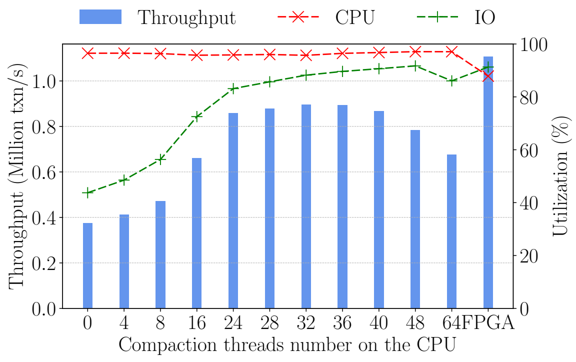


Figure 1: Impacts of added CPU threads for compactions on the overall system throughput.

KV stores built atop the LSM-tree usually exploit other database structures to offer a general-purpose storage service that excels in read, write performance and the cost at the same time. For example, indexes and caches contribute significantly to answering point lookups, which are among the majorities in the common write and point read-intensive (WPI) workloads [7]. Indexes offer a fast path to navigate in the huge storage, the size of which is often amplified by Multi-version Concurrency Control (MVCC) [16]. And, caches help reducing disk I/Os by buffering hot items in the main memory [16, 25].

Despite these efforts made, we have observed that the performance of LSM-tree KV stores often fatigues after serving the WPI workloads for long hours due to poorly maintained shapes (i.e., oversized levels) of the LSM-tree under workload pressures. An LSM-tree keeps multiple levels of records with inclusive key ranges in the disk (details introduced in

Section 2.1). This data organization often forces a query to traverse in multiple levels to merge scattered records for a complete answer or seek a record, even with indexes. Such operations carry the extra overhead of skipping over invalid records marked for deletion. To contain these drawbacks, background compaction operations (a.k.a., merge) are introduced to merge inclusive data blocks between adjacent levels and remove deleted records, intending to keep the LSM-tree in a properly tiered shape. However, in this work, we find that there exists a difficult trade-off between resources devoted to query and transaction processing in the critical path and resources devoted to background compactions which consume heavy computation and disk I/O resources, especially for WPI workloads containing both reads and writes. If we allocate more software threads to compactions, we throttle up the background maintenance of the storage at the risk of hurting the overall performance by depriving CPUs of actually processing queries and transactions.

Figure 1 plots the throughputs of an LSM-tree KV store processing a typical WPI workload (75% point lookups, 25% writes) using an increasing number of CPU threads for compactions. With the number of threads scaling up to 32, the total compaction throughput increases, resulting in significant performance benefits. However, with more threads added for compactions, CPUs become saturated and contended. Consequently, the system throughput drops after 32 threads as shown in Figure 1. We also find out with detailed profilings (introduced in Section 4) that compactions are still not fast enough to deal with the above-introduced problems with 32 or more threads.

Research efforts exploring two major approaches have been made to optimize compactions [6, 16, 18, 28, 29, 40]. One is to reduce the workload per compaction task by exploiting data distributions (e.g., almost sorted, not overlapped ranges) to avoid unnecessary merges [16, 29], or by splitting data into multiple partitions and schedule compactions for each partition when needed separately [18]. The other is to optimize the scheduling of compactions in terms of when and where compaction shall be executed [28]. Ideally, compaction should be completed when it is most needed for the sake of performance, and its execution has minimal resource contentions with other operations in the system. However, these two conditions often contradict with each other in WPI workloads, because the needs for compactions and the needs for high KV throughput often peak at the same time. Thus, the resource contentions in the storage system for both CPUs and I/O remain as a challenge, limiting the scalability of compaction speed with added CPU threads, and leaving the performance fatigue of LSM-tree storage system as an open problem.

In this paper, we propose to offload compactions from CPUs to FPGAs for accelerated executions due to four considerations. Firstly, offloading compactions away from CPUs relieves CPUs from such I/O-intensive operations. This enables the storage system to use less CPUs or increase the throughput using the same number of CPUs, both of which translate to monetary savings for users, especially in the public cloud. Secondly, FPGAs suit the requirement to accelerate compactions which are pipelines of computational tasks, compared with GPUs which are prefered by computations of the SIMD paradigm. We have also found out in this work that compactions merging short KV records (pairs) are surprisingly bounded by computation, partially because the disk I/O bandwidth has been improved significantly recently. This calls for dedicated performance optimization. Thirdly, the high power efficiency of FPGAs offers a competitive advantage on reducing the total cost of ownership (TCO), compared with other solutions. And, users of LSM-tree KV stores are sensitive to such costs, given that they pay for the storage almost permanently. We argue in this work that offloading compactions to FPGAs increase the economical value of LSM-tree KV stores in the cloud.

On the FPGA, we design and implement compaction as a pipeline including three stages on decoding/encoding inputs/outputs, merging data, and managing intermediate data in buffers. We also implement an FPGA driver, and an asynchronous compaction task scheduler to facilitate the offloading and improve its efficiency. We integrate our FPGA offloading solution with X-Engine, a state-of-the-art LSM-tree storage system [16]. We evaluate X-Engine with and without our proposal using typical WPI workloads with varying compositions. Experimental results show that the proposed FPGA offloading approach accelerates compactions by 2 to 5 times, improves the overall throughput of the storage engine by 23%, compared with the best CPU baseline. Overall, we make the following contributions:

- We have identified the slow and heavy compaction as a major bottleneck causing resource contentions, causing oversized level L_0 and other problems that eventually lead to the performance degradation of the LSM-tree KV store.

- We have designed and implemented an efficient multi-staged compaction pipeline on FPGAs (i.e., Compaction Unit), supporting merge and delete operations. We have introduced an asynchronous scheduler (i.e., Driver) to coordinate CPUs and the FPGA for the offloading of compactions. To the best of our knowledge, this is the first work on offloading compactions to FPGAs.

- We have modeled the FPGA compaction throughput analytically with validations. Using this model, we are able to predict the performance of the proposal for different offloaded compaction tasks. And, we have identified future optimization opportunities on the FPGA for compactions.

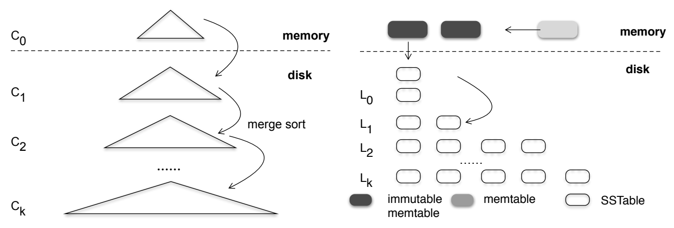
- We have compared the performance and energy consumptions of a state-of-the-art LSM-tree KV store (X-Engine [16]) using both the CPU-only baseline and the FPGA-offloading solution for compactions using both micro- and macro-benchmarks. Evaluation results show that our proposal increases the overall throughput of the KV store by 23% and increases the energy efficiency (number of transactions per watt) by 31.7%, compared with the CPU-only baseline.

This paper is organized as follows. Section 2 introduces backgrounds of LSM-tree KV stores, FPGAs and our motivations to offload compactions to FPGAs. Section 3 explains the overview of our design and introduces design details of compactions on the FPGA and the offloading process, including both implementation details and analytical models. We evaluate our proposals in Section 4. Finally, we discuss related work in Section 5 and conclude in Section 6.

2 Background and Motivation

2.1 LSM-tree KV-Store

LSM-trees have been extensively studied in academia and deployed in the industry due to its high write efficiency and low storage cost on SSDs. In the original design [23] shown in Figure 2a, an LSM-tree contains two tree-like components C_0 and C_1, residing in the main memory and the disk, respectively. For fast writes, incoming KV records are inserted into C_0, accessing only the main memory. When C_0 is full in the main memory, parts of it are merged with C_1 in the disk, leaving space in the main memory for new data. The overhead of this merge operation increases as the size of C_1 grows, because leaf nodes of C_0 may overlap with many leaf nodes of C_1. To bound such overhead, it is preferable to divide a single disk component into multiple ones: C_1, C_2, ..., C_k, where each component C_{i+1} is larger than its previous one C_i. However, a single KV record now has to be merged multiple times among these components over time, causing write amplification. And, a lookup may also have to access multiple components with inclusive key ranges, which are referred to as read amplifications in this work.



(a) Original LSM-tree. (b) Modern LSM-tree.

Figure 2: Original LSM-tree and Modern LSM-tree Architectures.

To bound write and read amplifications, many studies have envolved LSM-tree into a tiered storage with optimized main memory data structure, multi-level disk components with each level consisting of multiple files or fine-grained data blocks. Figure 2b shows a typical LSM-tree architecture as in many industrial systems such as RocksDB [13] and X-Engine [16]. Incoming data are inserted into memtables (often implemented as skiplists [24]). Once filled, memtables are switched to be immutable and flushed to the first level L_0 in the disk. A level L_k is similar to component C_k in the original design with a major difference that L_k is partitioned into many files (e.g., Sorted Sequence Tables as in RocksDB [13]) or data blocks (e.g., extents as in X-Engine [16]). There are two types of policies for the merging operation (i.e., compaction). For each batch to be merged into the next level (or component), one policy is to merge it with existing data in the target level, know as the *leveling* policy. This approach keeps data in a level in a nicely sorted order at the expense of the compaction speed. The other approach is to simply append data into the next level without merging, known as the *tiering* policy. In this way, the compaction itself is fast at the expense of the sorted order within a level. Dayan et al. have compared these two policies analytically [5]. Huang et al. proposed a data reuse technique allowing compaction to only physically merge data blocks with overlapping key ranges and reuse the rest to reduce the total I/O accesses [16]. Caches, indexes, and bloom filters have been introduced to compensate the shortcoming of the log-structured storage on lookup performance [7, 13, 16]. With skewed data accesses, these optimizations can reduce disk I/Os significantly [16].

2.2 Motivations

Despite the state-of-the-art optimizations introduced above, we find that slow compactions still lead to the performance fatigue problem of an LSM-tree KV store running the WPI workloads for the following reasons:

Problem 1: Shattered L_0. In the first level L_0, data blocks often have overlapping key ranges because they are directly flushed from the main memory without being merged. Unless compactions merge them in time, a point lookup may have to check multiple blocks for a single key, even with indexes. As time passes in such cases with slow compactions, data blocks in L_0 stockpiles, continuously increasing the lookup overhead. Such shattered L_0 have significant performance impacts because records flushed into L_0 are still very hot (i.e., very likely to be accessed) due to the data locality.

Problem 2: Shifting Bottlenecks. A compaction operations naturally consist of multiple stages: decoding, merging and encoding because KV records are often prefix encoded. To identify the bottleneck among these stages, we profile a single compaction task performed by a single CPU thread on an SSD. Figure 3 shows the execution time breakdown. As the value size increases, the percentage of computation time (*decoding*, *merging*, *encoding*) decreases. For short KVs, computation takes up to 60% time of the whole compaction procedure. When the value size exceeds 128 bytes, I/O operations occupy most of the CPU time. This breakdown shows that the bottleneck transfers from the CPU to the I/O with increasing KV sizes. This phenomenon suggests that the compaction is bounded by computation while merging short KVs and bounded by I/O in other cases.

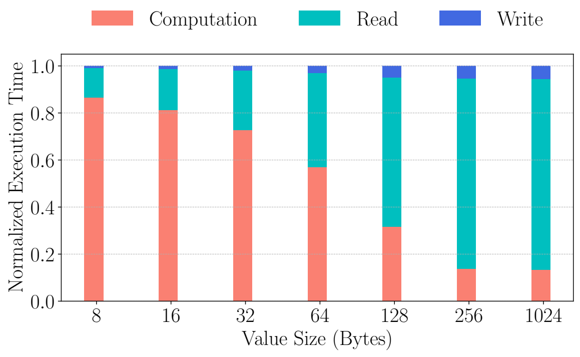


Figure 3: The breakdown of CPU compaction (key=8 bytes).

In this work, we map the above introduced three stages to a pipeline and offload it to accelerators. With faster compactions, data blocks in L_0 are more frequently merged. With offloading, CPUs are relieved from the heavy compaction operations, leaving more resources to process queries and transactions. We choose to offload compactions to FPGAs due to its suitability to accelerate pipelines of computational tasks like compactions, low power consumption for low TCO, and its flexibility as pluggable PCIe-attached accelerators.

2.3 FPGA offloading

A Filed-programmable Gate Array (FPGA) chip is an array of programmable hardware units, including look-up tables, flip-flops, and DSPs (digital signal processors). Software developers enjoy the flexibility in configuring FPGAs for dedicated purposes without other software overheads (e.g., the OS). FPGAs have been widely applied to accelerate pipelines of computational tasks for high performance and low power consumptions, compared with other accelerators such as GPUs, and many-core CPU processors [30, 33, 41, 42]. FPGAs suit such applications because programmed hardware units can be easily wired to implement pipelines with multiple stages. And, each hardware unit executes fixed operations, specified by designs, at every cycle with no extra overhead, achieving a high level of efficiency. In the market, FPGA chips are usually embedded in an SoC (system-on-chip) together with other hardware (e.g, RAM, CPUs, PCIe interfaces, SSD controllers) to meet different system requirements. It is also available in the public cloud (e.g., Amazon EC2 F1 instances, Alibaba Cloud F3 instances) to offer customizable computation services.

There are two main approaches to integrate FPGAs in a KV store. One is the “bump-in-the-wire" design that places the FPGA between the CPU and the disk [37]. This approach, in which the FPGA serves as a data filter, is favoured when the on-chip RAM of FPGAs are small in size so that it can only temporarily hold a slice of a data stream. As the on-chip RAMs grow in capacity, FPGAs are now capable to serve as a co-processor, onto which we can offload operations with large data volume to process. This approach suits asynchronous tasks, during which CPUs are not stalled by the offloaded tasks. In this paper, we adopt the latter approach to offload compactions as a whole to FPGAs and only use the CPU for task generations. Such solutions based on FPGA-offloading can naturally evolve into a computational storage with coupled FPGA-SSD devices emerging in the market (e.g., Samsung SmartSSD).

3 Design and Implementation

3.1 Overview

Figure 4 shows our design of the FPGA-offloading for compactions, integrated with X-Engine. X-Engine is one of the state-of-the-art LSM-tree KV store [16]. It exploits memtables to buffer newly inserted KV records, and caches to buffer hot pairs and data blocks. In the disks, each level of the LSM-tree contains multiple extents. Each extent in turn stores KV records with associated indexes and filters. X-Engine maintains a global index to accelerate lookups. In the baseline system without offloading compactions, all put, get, delete, multi-get operations in addition to background flush and compaction operations are processed by CPUs. We refer to this as the CPU-only baseline.

To offload compactions, we first design a **Task Queue** to buffer newly triggered compaction tasks, and a **Result Queue** to buffer compacted KV records in the main memory. Software-wise, we introduce a **Driver** to offload compactions, including managing the data transfer from the host and the FPGA (the device). On the FPGA, we design and deploy multiple **Compaction Units (CUs)**, responsible for merging KV records.

3.2 Driver

3.2.1 Managing Compaction Tasks

In LSM-tree KV stores such as X-Engine, compaction between level L_i and L_{i+1} are triggered when the data size of L_i (i.e., the number of extents as in X-Engine) reaches a predefined threshold. To facilitate offloading, we maintain three types of threads: builder threads, dispatcher threads and driver threads at the CPU side to build compaction tasks, dispatch tasks to CUs in the FPGA, and install compacted data blocks back into the storage, respectively. We illustrate this process in Figure 5 and introduce the details of these threads in the following.

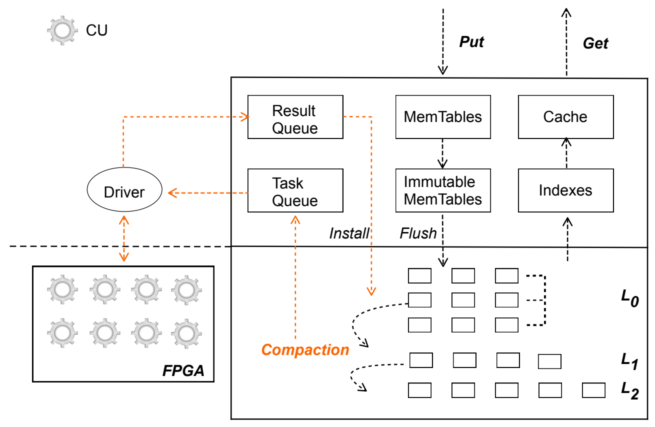


Figure 4: System Overview of the Storage Engine with the FPGA-offloading of Compactions.

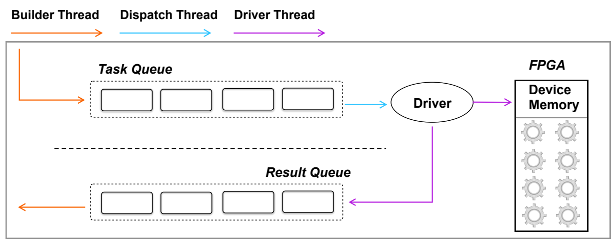


Figure 5: Asynchronous Compaction Scheduler. By implementing a thread pool to distribute FPGA-offloading compaction tasks, the cost of context switch is reduced.

Builder thread For each triggered compaction, the builder thread partitions extents to be merged into multiple groups of similar sizes. Each group then forms a compaction task, with its data loaded into the main memory. An FPGA compaction task is built containing the metadata required including pointers to the task queue, input data, results, a callback function (transferring compacted data blocks from the FPGA to the main memory), return code (indicating whether a task is finished successfully) and other metadata of the compaction task, as shown in Figure 6. This compaction task is pushed to the task queue, waiting to be dispatched to a CU on the FPGA. This builder thread also checks the result queue and installs compacted data blocks back to the storage when the task is successful. In cases when the FPGA failed to complete

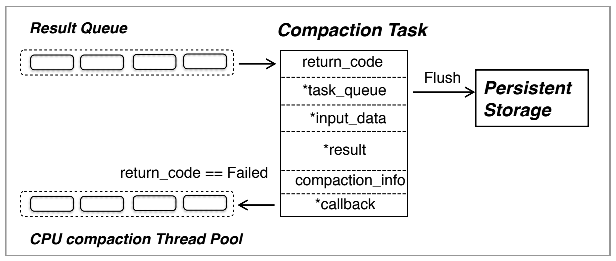


Figure 6: CPU checks the task's status, redoes the task and flushes the data as SSTable if necessary.

a compaction task (e.g., the key-value size exceeds FPGA's capacity), it starts a CPU compaction thread to retry this task. In our practices, we find that only 0.03% tasks offloaded to the FPGA fail on average.

Dispatcher thread The dispatcher consumes the task queue and dispatches tasks to all the CUs on the FPGA in a round-robin manner. Given that compaction tasks have similar sizes, such round-robin dispatching achieves balanced workload distribution among multiple CUs on the FPGA. The dispatcher notifies the driver thread to transfer the data to the device memory on the FPGA.

Driver thread The driver thread transfers input data associated with a compaction task to the device memory on the FPGA and notifies the corresponding CU to start working. When a compaction task is finished, this driver thread is interrupted to execute the callback function, which transfers the compacted data blocks back to the host memory, and pushes the completed task into the result queue.

In such implementation, we tune the size of a compaction task to provide sufficient data for a compaction unit, guarantee load balances among CUs, and limit the overhead for retrying a compaction task upon failures on the FPGA. We also tune the number of threads for the builder, dispatcher, and the installer separately, to achieve balanced throughputs among them.

3.2.2 Instruction and Data Paths

To drive the FPGA for compactions, we need to transfer instructions and data via the PCIe interconnect. To maximize such transfer efficiency, we design an *Instruction Path* and a *Compaction Data Path*. We also design an *Interruption Mechanism* to notify the *Installer Thread* when a CU completes a compaction task, and a *Memory Management Unit (MMU)* managing the device memory on the FPGA. The whole process is depicted in Figure 7.

The detailed description of the submodules are as follows:

Instruction Path The instruction path is designed for small and frequent data transfers like the CU availability check.

Compaction Data Path The data path uses DMA. The data for compaction is transferred via this path. By this way, the CPU is not involved in the data transfer procedure.

Interruption Mechanism When a compaction task finishes, an interrupt will be sent via PCIe, and then the result is written back to the host with the help of the interrupt vector.

Memory Management Unit (MMU) MMU allocates memory on the device memory to store the input data copied from the host.

3.3 Compaction Unit

Compaction Unit (CU) is the logic implementation of the compaction operation on the FPGA. Multiple CUs can be deployed on the same FPGA, the total number of which mainly

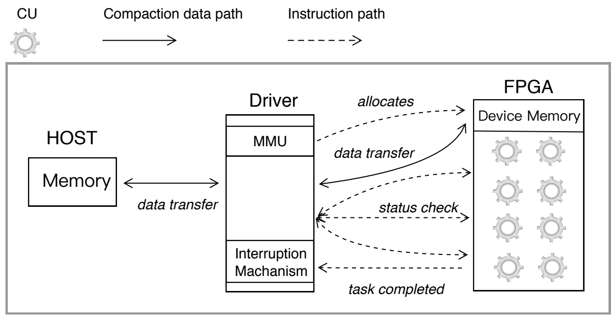


Figure 7: FPGA Driver.

depends on the available hardware resources. Figure 8 illustrates the design of a CU. In this design, a compaction task consists of multiple stages: Decoder, Merger, KV Transfer, and Encoder (details introduced below). We introduce buffers (i.e., KV Ring Buffer, and Key Buffer) between subsequent modules and a Controller to coordinate the execution of each module.

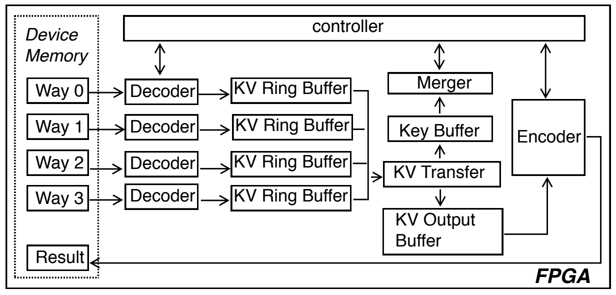


Figure 8: Compaction Unit overview.

Decoder We consider a KV store where keys are prefix-encoded to save spaces. The *Decoder* decodes input KV records. If the tiering compaction policy is adopted, there may exist multiple ways of input data blocks to be merged. For the levelling policy, there are at most two ways of inputs from the two adjacent levels. We find that most tiering compactions involve two to four ways of inputs in our practices. If we place two decoders in a CU with each one decoding one way of inputs, we need to build three two-way compactions to achieve a single four-way compaction. Instead, placing four decoders consume 40% more hardware resources without requiring more tasks for two to four ways of compactions, compared with the two-decoder design. Thus, we chose to place four decoders in each CU. The decoder outputs decoded KV records into the *KV Ring Buffers*.

KV Ring Buffer The *KV Ring Buffer* caches the decoded KV records. We have observed in our practices that KV sizes rarely exceed 6KB. Thus, we configure each KV ring buffer to contain 32 8KB slots. The additional 2KB can be used to store meta-information such as the KV length. We design three status signals for a KV ring buffer: $FLAG\_EMPTY$, $FLAG\_HALF\_FULL$ and $FLAG\_FULL$, indicating whether the buffer is empty, half full, or fully filled, respectively. Beginning with an empty buffer, the decoder keeps decoding and filling this buffer. After this buffer is half-filled, the downstream *Merger* is allowed to read the filled data and starts merging them. While the merger is working, the decoder continues to fill the rest of the buffer until it is fully filled. We match the speed of the merger and the decoder so that the merge takes the same amount of time to finish its job by filling half of the ring buffer. In this way, the decoder and the merger can be efficiently pipelined. In cases that these two modules do not match, the *Controller* pauses the decoder. We keep a read pointer for the merger to mark from where it should read in KV records, and a write pointer for the decoder similarly.

KV Transfer, Key Buffer and Merger Only keys are transferred to the *Key Buffer*, and then the *Merger* for comparisons. If a key is qualified as an output, the *KV transfer* module transfers the corresponding KV record from the upstream KV ring buffer to the *KV Output Buffer* with the same data structure as the KV ring buffer. The *Controller* gets notified with the comparison result and move the read pointer in the corresponding KV Ring Buffer forward accordingly. For example, as illustrated in Figure 9, if the KV record in *way 2* is the smallest key-value in the current comparison round, the Controller will notify KV Transfer to transfer the KV record where the read pointer points in *way 2*'s KV Ring Buffer and move the read pointer forward to fetch the next entry into the key buffer for the next round.

Encoder This module encodes the merged KV records from the KV output buffer to data blocks and places them into the device memory. Because there is only one way of merged data, we place only one encoder within each CU.

Controller The *Controller* module serves as a coordinator in a CU. It manages the read and write pointers in the KV ring buffer for the merger and decoder, respectively. It also signals each module when to start or pause.

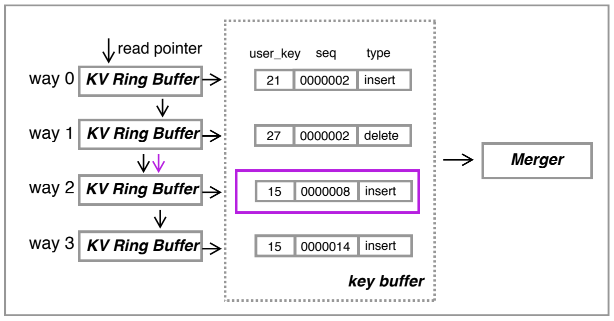


Figure 9: Detailed Design of the Merger.

3.4 Analytical Model for CU

Due to the pipeline design, it is crucial to match the throughput of different modules to avoid over-provisioning and waste of hardware resources. However, it is difficult to derive the number of resources for each component with so many tuning knobs. Therefore, in this section we propose an analytical model for CU to guide the resource provisioning. The notations and parameters that we use in the model are summarized in Table 1.

Table 1: Summary of Notations used in the Analytical Model.

Parameter	Values/Units	Description
N	Workload-dependent	Total number KVs processed
f_{FPGA}	200 MHz	Clock frequency of the FPGA
W_{bus}	8 Bytes	Width of bus data
W_{key}	$1 \sim 2$K Bytes	Width of key
W_{value}	$0 \sim 4$K Bytes	Width of value
$A_{decoder}$	2	Decoder amplification factor[1]
$A_{kv\_transfer}$	1	KV_transfer amplification factor
A_{merger}	5	CPE amplification factor
$A_{encoder}$	2	Encoder amplification factor
$B_{transfer}$	Bytes/second	Host-device transfer bandwidth
$b_{decoder}$	10	Base cycles for Decoder
$b_{kv\_transfer}$	18	Base cycles for KV_transfer
b_{merger}	50	Base cycles for Compaction PE
$b_{encoder}$	46	Base cycles for Encoder
μ	Workload-dependent	Merging selectivity

The throughput of a CU is dominated by the throughput of the slowest stage as shown in Equation 1, with only one exception on the KV Transfer. When the KV Transfer does work to transfer merged KVs, it is executed in serial with its predecessor stage, Merger. In this case, their costs shall be combined. The cost of each stage consists of its computation and memory overhead (i.e., moving data into/from Block RAMs inside the FPGA chip) in addition to a constant number of base cycles consumed when starting this stage. We introduce the details in the following.

$$T_{CU} = min\{T_{decoder}, T_{kv\_transfer}, T_{cpe}, T_{encoder}, T_{mem}\} \quad (1)$$

$$T_{decoder} = \frac{f_{FPGA}}{b_{decoder} + (A_{decoder} \cdot W_{key} + W_{value})/W_{bus}} \quad (2)$$

$$T_{merger} = \frac{f_{FPGA}}{b_{merger} + A_{merger} \cdot W_{key}/W_{bus}} \quad (3)$$

$$T_{kv\_transfer} = \frac{f_{FPGA}}{b_{kv\_transfer} + (A_{kv\_transfer} \cdot W_{key} + W_{value})/W_{bus}} \quad (4)$$

$$T_{encoder} = \frac{f_{FPGA}}{b_{encoder} + (A_{encoder} \cdot W_{key} + W_{value})/W_{bus}} \quad (5)$$

$$T_{mem} = \frac{N \cdot (W_{key} + W_{value}) \cdot (1+\mu)}{B_{transfer}} \quad (6)$$

Equation 2, 4, 3, and 5 model the cost of processing a single KV record for the *Decoder*, *Merger*, *KV Transfer* and *Encoder* stages, respectively. Equation 6 models the throughput of data transfer between the host and the device, bounded by PCIe bandwidth. Through inspecting the hardware implementation and performance profiling, we initiate these models with data listed in Table 1. As an example, for a KV record with 8-byte key and 32-byte value, the costs for each stage in the term of cycles consumed per KV are 16, 55, 23, and 52 for the Decoder, Merger, KV Transfer, and Encoder stages, respectively. The overall throughput is $\frac{200MHz}{55} = 3.6\ M\ records/s$, assuming the performance is bounded by the Merger. We valid this model with experiments in Section 4.2.

4 EVALUATION

In this section, we first evaluate how a Compaction Unit (CU) performs on the FPGA using microbenchmarks, in addition to validating our analytical model for a CU. We then evaluate how the proposed FPGA-offloading of compactions contribute to X-Engine using varying workloads. X-Engine is one of the state-of-the-art LSM-tree KV stores [16].

4.1 Experimental Setup

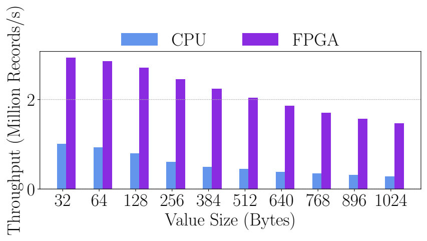
We evaluate our proposal on a server featuring two Intel Xeon Platinum 8163 2.5 GHz 24-core CPUs with two-way hyper-threading, a 768 GB Samsung DDR4-2666 main memory and a RAID 0 consisting of 10 Samsung SSDs, running Linux 4.9.79. We attach a Xilinx Virtex UltraScale+ VU9P FPGA board (running at 200MHz) with a 16 GB device memory to this server through a x16 PCIe Gen 3 interface, to which we offload compactions. Specifications of this FPGA board is summarized in its datasheet [39].

4.2 Evaluating the FPGA-based Compaction

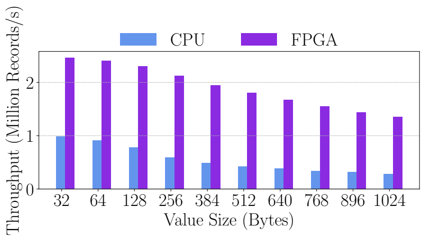
CU Performance Analysis. To evaluate the performance of a CU, we prepare 4,000,000 KV records with varying key and value sizes and manually trigger a compaction to merge them. Figure 10 compares the throughput, in the term of input KV records merged per second, achieved by a single CPU thread and the FPGA-offloading of compactions.

We observe that the throughputs of the FPGA-offloading compactions are 203% to 507% higher than those on the CPU, achieving performance speedups across all KV sizes. For shorter KV sizes (thus smaller compaction tasks), the FPGA-offloading solution suffers from a relatively higher percentage of the offloading overhead. Such costs are diluted when the KV size increases, resulting in higher speedups over the CPU.

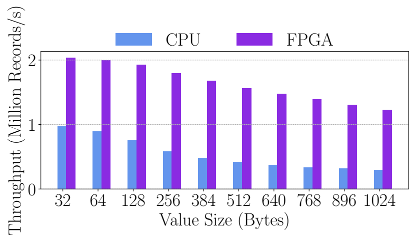
[1]The amplification factor indicates the times of the number of clock cycles for each module if we use kv_transfer as the baseline.



(a) 8-byte key.



(b) 32-byte key.



(c) 64-byte key.

Figure 10: Throughput of FPGA-based and CPU-based compactions with varying KV settings.

We validate our analytical model for a CU on the FPGA using a high-pressure scenario in which all input KV records are decoded, compared, encoded, and written back to the storage (i.e., 100% selectivity). In this case, in our current implementation, the Merger and the KV Transfer stages are not pipelined. They execute in serial. Figure 11 reports the measured and estimation throughput. For KV records up to 8-byte key and 256-byte values, estimation errors are within only 5%. With larger sizes, the error grows up to 13% for the 1024-byte value. Such estimation inaccuracies are very likely caused by potential pipeline stalls and bus contentions. And, we conclude that the overall throughput is always bounded by that of the Merger and KV Transfer stages combined, neither the accesses to the device memory nor the data transfer over the PCIe.

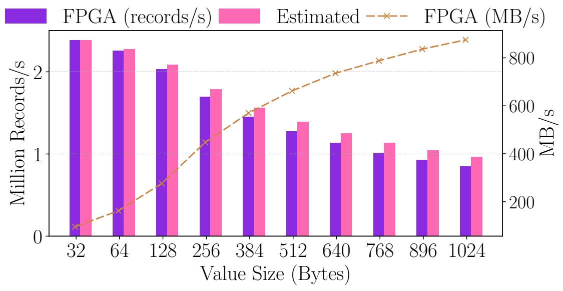


Figure 11: FPGA Compaction Model (key=8 bytes).

Resource Consumption. In Table 2, we report the resource consumption (i.e., LUT, Flip-Flop and memory consumed) of each stage in a CU on the FPGA. A single CU consumes less than 5% of the total resources, with the encoder consuming the largest number of resources because it has the most complicated logics including prefix encoding and communication with other stages. In practice, we can place up to 8 CUs in the FPGA board, utilizing 25.6% of the LUTs, 10.4% of the Flip-Flops, and 38.4% of the device RAM. In addition to the about 15% resources consumed by the shell inside the FPGA chip (including buses, DDR controllers, etc.), our current implementation utilizes about half of the hardware resources on the FPGA. We recognize that such utilization is constrained by the necessary place and route overhead on the FPGA and our current implementation which can be improved with further optimizations.

Table 2: Hardware resource utilization by a single CU.

	LUT	**Flip-Flop**	**RAM (MB)**
Decoder	5783×4	4570×4	8×4
KV Transfer	1076	1006	0
Merger	4119	2555	0
Encoder	6489	4101	0
Others	3819	4841	14
1 CU	38635	30783	46
FPGA total	1182000	2364000	960
Utilization	**3.2%**	**1.3%**	**4.8%**

4.3 Evaluating a KV Store with FPGA-offloading of Compactions

4.3.1 Workloads

To evaluate the impacts of FPGA-offloading of compactions on an LSM-tree KV store using X-Engine [16], we adopt DBBench, a popular opensource benchmark to measure LSM-tree storages such as LevelDB and RocksDB [8, 25, 26]. We consider skewed workloads following a zipf distribution, and use the zipf factor of 1.0 to generate the default workload. We vary multiple parameters in our experiments including the number of CPU threads used for compactions, the read ratio (i.e., the percentage of reads (v.s. writes) in the workload) and others.

We prepare KV records for 32 LSM-tree tables, with each table storing 200,000,000 records in a single X-Engine KV store instance. The total data size is about 278 GB. We tune the cache size in the main memory to 70 GB. By default, we use 8-byte keys and 32-byte values, a common size as in many KV stores deployed in the industry, especially those storing secondary indexes. In all experiments, we warm up the storage for 3,600 seconds and measure performance metrics for another 3,600 seconds, which is long enough for the exposure of performance issues in our setting.

4.3.2 The Impacts of Compactions

Figure 12 shows the overall performance (transactions per second) achieved in correspondence with the size of level L_0, comparing the impacts of increasing numbers of threads for compactions on the CPU-only baseline and the FPGA-offloading solution, given our default WPI workload. By up to 24 CPU threads, added threads yield significant performance improvements by merging overlapping key ranges in the level L_0 at an increasing speed, which reduces the expected number of I/O accesses for a point lookup. In this phase, compactions are slower than the total write throughput of the KV store, so that it is not able to ingest enough newly written data in the LSM-tree. Thus, accelerating compactions using more threads pays back.

From 24 to 32 threads, the throughput barely changes. After 32 threads, the throughput drops with more threads added. The reason is two-fold. Firstly, more threads added introduce more resource contentions, as shown in Figure 1 where the overall CPU utilization almost always 100%. And, we show in Figure 3 that compactions, in this case, are bounded by computations on the CPU. Secondly, added threads eventually saturate the disk I/O as shown in Figure 1, incurring I/O contentions for both the query/transaction processing and compactions.

Figure 12 shows that the FPGA-offloading compaction achieves around 23% higher throughput than the best CPU-only baseline in this workload. Both accelerating compactions and reducing resource contentions at the CPU side contribute to such improvements (shown in Figure 1). Due to the fine-grained compaction task build and distribute, we have not observed any obvious difference in memory consumption between FPGA-offloading compaction and CPU-based compaction (shown in Figure 1). The average memory bandwidth with FPGA-offloading compaction (50.06 GB/s) is 29% lower than the best CPU baseline (70.50 GB/s), showing the CPU-only approach consumes more memory bandwidth.

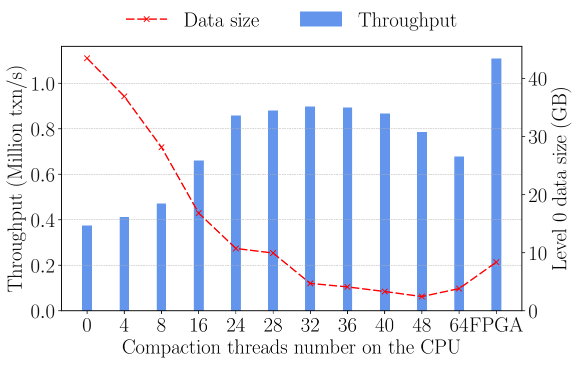


Figure 12: Compaction threads number vs. L_0 data size.

Table 3 summaries the performance metrics and power consumption of X-Engine with FPGA-offloading compaction and CPU-based compaction using 32 threads. With FPGA acceleration, the throughput is improved by 23.3% while the average response time is decreased by 14.7% and 28.1% for read and write operation respectively. The tail request latency is reduced by over 40% as the computation contention is relieved with compaction offloaded to the FPGA. Furthermore, we report the power consumptions. The average machine-wise power consumption is lowered by 7.2% with FPGA-offloading of compactions reducing the CPUs' utilization and power consumptions. Because the FPGA has higher energy efficiency for compactions than CPUs, the overall energy efficiency (number of transactions processed per watt) is improved by 31.7%. With further continuous efforts on optimizing such FPGA-offloading implementations for KV stores, more energy savings and TCO reductions are promising.

4.3.3 The Impacts of Read Ratio

In this section, we investigate the performance benefits of the FPGA-offloading compaction when the read ratio varies. In Figure 13, we gradually reduce the read ratio to 75% which resembles a write-heavy WPI workload, our proposed offloading approach outperforms the baseline with around 10% reduction in CPU consumptions in all cases and consumes slightly more I/Os due to its higher throughput than the CPU.

4.3.4 Macro Benchmarks

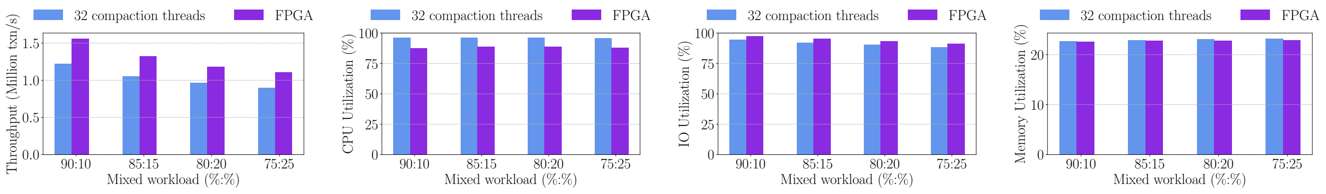
Now we compare the CPU-only baseline and our proposal using the DBBench [26] and YCSB benchmarks [4]. Figure 14 reports the DBBench results, including the following tests: **FR** (*fill random*, randomly inserting records), **SRWR** (*seek random while writing*, seeking random individual records with only one thread inserting records), **RWR** (*read while writing*, reading multiple random threads with only one thread inserting records), **UR** (*update random*, updating random records) and **RRWR** (*read random write random*, all threads reading and writing random records for 90% and 10% of the time, respectively).

In these tests, our proposal manages to reduce the CPU utilization and improve the throughput with similar I/O and memory consumptions. FR has the highest performance improvement (54% improvement in total) per CPU utilization reduction (15% utilization reduction in total) because such write-only workloads generate a lof of compactions that benefit from offloading. For the I/O-bounded RWR, our offloading approach improves the throughput by 18% by relieving CPUs from compaction I/Os (20% utilization reduction) without resolving the I/O bottleneck. In other tests, our approach achieves slightly higher I/O utilizations than the CPU-only baselines, resulting in 17% to 29% performance improvements. These results demonstrate multiple benefits of replacing CPUs for the execution of compactions by FPGAs.

Figure 15 reports the YCSB results using the following

Table 3: The value of adding an FPGA to X-Engine.

	Million Txn/s	*Avg Get RT (μs)*	*P99 Get RT (μs)*	*Avg Put RT (μs)*	*P99 Put RT (μs)*	*Power (Watt)*	*Efficiency (Txn/Watt)*
CPU (32 compaction threads)	0.90	139.89	928.15	107.51	864.95	636.54	1416.54
CPU + FPGA	1.11	119.38	537.08	77.30	499.52	590.78	1865.44
Improvement	**+23.3%**	**-14.7%**	**-42.1%**	**-28.1%**	**-42.2%**	**-7.2%**	**+31.7%**



(a) Throughput. (b) CPU Utilization. (c) I/O Utilization. (d) Memory Consumptions.

Figure 13: Comparing the CPU-only baseline and the FPGA-offloading proposal with varying read ratios (percentages of gets).

workloads: **workload a** (50% read and 50% update), **workload b** (95% read and 5% update), **workload c** (read-only), **workload d** (95% read and 5% insert), **workload e** (95% scan and 5% insert) and **workload f** (50% read-modify-write latest records and 50% random reads). Reads dominate all these workloads, resulting in non-frequent compactions in the undelrying KV store. Hence, the throughout improvement is marginal with our proposal. We observe 23.7% and 16.1% gains on throughput for **workload a** and **workload f**, respectively. Both these workloads contain non-trival writes.

In both the DBBench and the YCSB benchmarks, the proposed FPGA-offloading of compactions contribute more to the KV store when there are significant writes in the workload, compared with read-intensive ones, because compactions in LSM-trees are only triggered upon writes. Acknowledging that both the WPI workload and read-intensive workloads are common in the applications of LSM-tree KV stores, there are efforts in both the literature and the industry that exploit FPGAs to accelerate lookups and queries [31]. Given the flexibility of FPGAs, it is worth exploring of dynamically switching the logics of FPGAs or programming different dedicated accelerator units in a single FPGA to suit various workloads in the production.

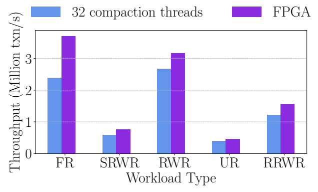
5 Related Work

5.1 Software Optimizations of Compactions

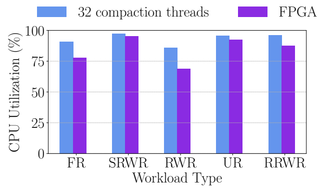
To improve the efficiency of compactions, VT-tree [29] uses the stitching technique to avoid unnecessary disk I/Os for sorted and non-overlapping key ranges. However, this method is subject to data distributions and may lead to fragmentations, worsening the performance of range scans and compactions. In bLSM [28] and PE [18], the data distribution is taken into consideration. Both algorithms partition the key range into multiple sub-key ranges and confine compaction in hot data key ranges, which accelerates the data flow. PCP [40] observed that the compaction procedure can be pipelined. It employs multiple CPUs and storages to fully utilize both CPU and I/O resources to speed up the compaction procedure. Dayan et al. offer richer space-time trade-offs by merging as little as possible to achieve given bounds on lookup cost and space and proposed a hybrid compaction policy (i.e., use levelling for the largest level and tiering for the rest) to reduce write amplifications [6, 25]. Huang et al. propose to split data in the LSM-tree into small data blocks, and reuse data blocks without overlapping key ranges during compactions extensively to reduce the write amplification [16]. All these software optimizations are orthogonal to our work on accelerating the compaction using FPGAs. We believe combined software and hardware optimization will benefit the system performance and power consumption a step further. It seems to be a fruitful area that orchestrates the hardware (e.g., CPU, FPGA, GPU) and exploit the potentials to schedule the tasks to the hardware that best suit their characteristics.

5.2 Hardware Accelerations in Databases

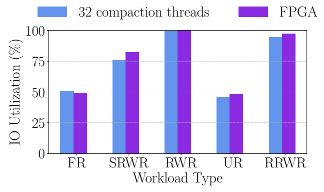
In the past decades, optimizing databases using carefully designed hardware techniques has been a very heated area in both the academia and the industry. Because both analytical and transactional databases manipulate data records through a well-defined set of operators or primitives (e.g., put, get, scan, join), we are able to either implement such operators using fixed hardware logics on a database-specific processor or machine [1, 12, 17, 36, 38], or do so using high-level languages (e.g., C++, OpenCL) on a general-purpose processor with assistances from latest hardware features (e.g., SIMD, high-bandwidth memory) [3, 15]. Comparing these two approaches, database-specific hardware very likely delivers groundbreaking performance and energy-efficiency with a relatively longer time-to-market and a higher engineering and financial cost per processor. And, hardware-aware opti-



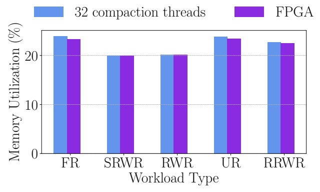
(a) Throughput.



(b) CPU Utilization.

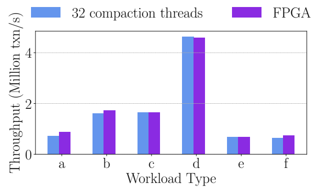


(c) I/O Utilization.

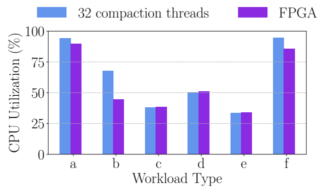


(d) Memory Consumption.

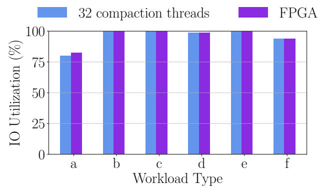
Figure 14: Comparing the CPU-only baseline and the FPGA-offloading proposal using the **DBBench** benchmark.



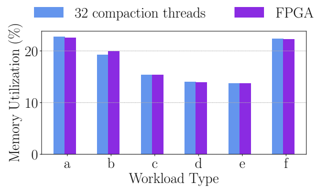
(a) Throughput.



(b) CPU Utilization.



(c) I/O Utilization.



(d) Memory Consumption.

Figure 15: Comparing the CPU-only baseline and the FPGA-offloading proposal using the **YCSB** benchmark.

mizations on general-purpose processors are much easier to program and deploy in the market with lower levels of the energy proportionality and performance speedup [32]. This work tries a third approach using FPGAs.

We explore using FPGA, a flexible and programmable hardware accelerator, for database optimizations aiming at both higher performance and higher economic values. In the literature, FPGAs have already been widely studied to accelerate individual operators or algorithms such as data partition [20], hashing [19], algorithmic trading [27], achieving significant performance improvements. For SQL accelerations, Glacier [21] is a compiler that translates SQL queries into VHDL code, targeting at streaming and standing queries. For unpredictable workloads like warehouse scenario, techniques like partial reconfiguration [10] and runtime parameterization [22] has been proposed. IBM's Netezza [14] managed to push simple selection and projection-based filtering to FPGAs. Advanced SQL offloading like `Group By` and `Where` clause has also been addressed [10, 11, 37]. The state-of-art SQL acceleration is capable of dealing with pattern matching and user-defined function [30]. Commercial examples for SQL offloading like Oracle's Exata [34] and IBM's Netezza [14] reveals the interests of the industry in this field.

Instead of stretching the arms on these *foreground* operators or algorithms, we offload a heavy and frequently executed *background* operation, compaction, to FPGAs. We argue that FPGAs can contribute to the overall database performance (a database storage in our case for now) by relieving CPUs from computation-bounded tasks, and performing much more efficiently for the offloaded tasks in a relatively isolated hardware environment.

6 Conclusion

In this paper, we argue that the LSM-tree KV store suffers from slow compactions due to resource contentions, oversized LSM-tree levels and other reasons. With the CPU and I/O consumptions of compactions carefully profiled, we find that compactions can be surprisingly bounded by computation for merging shot KVs. We further propose to offload compactions to a dedicated FPGA for acceleration and integrate our proposal with X-Engine, a state-of-the-art LSM-tree KV store. To facilitate such offloading, we have designed and implemented the pipelined *Compaction Units* on the FPGA and a *Driver* to achieve efficient offloading. With evaluations comparing our proposal with the fine-tuned CPU-only baseline, we show that the proposed FPGA-offloading of compactions can increase the system throughput and energy efficiency by up to 23% and 31.7%, respectively.

Acknowledgments

This work is a result of a joint project by multiple teams at Alibaba Cloud. The POLARDB X-Engine team develops the underlying LSM-tree storage engine used in this study [16]; the Hardware and Software Co-development team implements the proposals in this paper on FPGAs with significant efforts. We would like to acknowledge Xulin Yu, Ming Zeng, Xiaohui Yan, Yang Kong, and Dongdong Wei in particular for their contributions on implementing and evaluating the FPGA solution. We deeply appreciate anonymous reviewers and our shepherd for their constructive comments on this work.

References

[1] Oliver Arnold, Sebastian Haas, Gerhard Fettweis, Benjamin Schlegel, Thomas Kissinger, and Wolfgang Lehner. An application-specific instruction set for accelerating set-oriented database primitives. In *Proceedings of the 2014 International Conference on Management of Data (SIGMOD)*, pages 767–778. ACM, 2014.

[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[3] Xuntao Cheng, Bingsheng He, Mian Lu, Chiew Tong Lau, Huynh Phung Huynh, and Rick Siow Mong Goh. Efficient query processing on many-core architectures: A case study with intel xeon phi processor. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2081–2084. ACM, 2016.

[4] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[5] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 International Conference on Management of Data (SIGMOD)*, pages 79–94. ACM, 2017.

[6] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, pages 505–520. ACM, 2018.

[7] Niv Dayan and Stratos Idreos. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*, 2019.

[8] Jeff Dean and Sanjay Ghemawat. Leveldb. https://github.com/google/leveldb, 2020.

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, pages 205–220. ACM, 2007.

[10] Christopher Dennl, Daniel Ziener, and Jurgen Teich. On-the-fly composition of fpga-based sql query accelerators using a partially reconfigurable module library. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 45–52. IEEE, 2012.

[11] Christopher Dennl, Daniel Ziener, and Jürgen Teich. Acceleration of sql restrictions and aggregations through fpga-based dynamic partial reconfiguration. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 25–28. IEEE, 2013.

[12] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, H-I Hsiao, and Rick Rasmussen. The gamma database machine project. *IEEE Transactions on Knowledge and data engineering*, 2(1):44–62, 1990.

[13] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, volume 3, page 3, 2017.

[14] Phil Francisco et al. The netezza data appliance architecture: A platform for high performance data warehousing and analytics. *IBM Redbooks*, 2011.

[15] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K Govindaraju, Qiong Luo, and Pedro V Sander. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS)*, 34(4):21, 2009.

[16] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. X-engine: An optimized storage engine for large-scale e-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*, pages 651–665. ACM, 2019.

[17] Balakrishna R Iyer. Hardware assisted sorting in ibm's db2 dbms. In *International Conference on Management of Data, COMAD 2005b*, 2005.

[18] Christopher Jermaine, Edward Omiecinski, and Wai Gen Yee. The partitioned exponential file for database storage management. *The VLDB Journal—The International Journal on Very Large Data Bases*, 16(4):417–437, 2007.

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

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

[21] Rene Mueller, Jens Teubner, and Gustavo Alonso. Streams on wires: a query compiler for fpgas. *Proceedings of the VLDB Endowment*, 2(1):229–240, 2009.

[22] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. Flexible query processor on fpgas. *Proceedings of the VLDB Endowment*, 6(12):1310–1313, 2013.

[23] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[24] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[25] RocksDB. A persistent key-value store for fast storage environments. https://rocksdb.org/, 2020.

[26] Facebook RocksDB. Performance benchmarks. https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks, 2019.

[27] Mohammad Sadoghi, Martin Labrecque, Harsh Singh, Warren Shum, and Hans-Arno Jacobsen. Efficient event processing through reconfigurable hardware for algorithmic trading. *Proceedings of the VLDB Endowment*, 3(1-2):1525–1528, 2010.

[28] Russell Sears and Raghu Ramakrishnan. blsm: a general purpose log structured merge tree. In *Proceedings of the 2012 International Conference on Management of Data (SIGMOD)*, pages 217–228. ACM, 2012.

[29] Pradeep Shetty, Richard P Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building workload-independent storage with vt-trees. In *FAST*, pages 17–30, 2013.

[30] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. Accelerating pattern matching queries in hybrid cpu-fpga architectures. In *Proceedings of the 2017 International Conference on Management of Data (SIGMOD)*, pages 403–415. ACM, 2017.

[31] David Sidler, Zsolt István, Muhsen Owaida, Kaan Kara, and Gustavo Alonso. doppiodb: A hardware accelerated database. In *Proceedings of the 2017 International Conference on Management of Data (SIGMOD)*, pages 1659–1662. ACM, 2017.

[32] Dimitris Tsirogiannis, Stavros Harizopoulos, and Mehul A Shah. Analyzing the energy efficiency of a database server. In *Proceedings of the 2010 International Conference on Management of Data (SIGMOD)*, pages 231–242. ACM, 2010.

[33] Zeke Wang, Kaan Kara, Hantian Zhang, Gustavo Alonso, Onur Mutlu, and Ce Zhang. Accelerating generalized linear models with mlweaving: a one-size-fits-all system for any-precision learning. *Proceedings of the VLDB Endowment*, 12(7):807–821, 2019.

[34] Ronald Weiss. A technical overview of the oracle exadata database machine and exadata storage server. *Oracle White Paper. Oracle Corporation, Redwood Shores*, 2012.

[35] MongoDB WiredTiger. Wiredtiger. https://github.com/wiredtiger/wiredtiger.

[36] De Witt. Direct-a multiprocessor organization for supporting relational database management systems. *IEEE Transactions on Computers*, 100(6):395–406, 1979.

[37] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex: An intelligent storage engine with support for advanced sql offloading. *Proceedings of the VLDB Endowment*, 7(11):963–974, 2014.

[38] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: The architecture and design of a database processing unit. *SIGPLAN Not.*, 49(4):255–268, February 2014.

[39] Xilinx. Ultrascale architecture and product data sheet: Overview. https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf, 2019.

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

[41] Shijie Zhou, Rajgopal Kannan, Yu Min, and Viktor K Prasanna. Fastcf: Fpga-based accelerator for stochastic-gradient-descent-based collaborative filtering. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 259–268. ACM, 2018.

[42] Shijie Zhou, Rajgopal Kannan, Hanqing Zeng, and Viktor K Prasanna. An fpga framework for edge-centric graph processing. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pages 69–77. ACM, 2018.