Re-LSM: A ReRAM-based Processing-in-Memory Framework for LSM-based Key-Value Store

Qian Wei, Zhaoyan Shen*
Yiheng Tong, Zhiping Jia
Lei Ju, Jiezhi Chen
Shandong University
Qingdao, Shandong, China

Bingzhe Li
Oklahoma State University
Stillwater, Oklahoma, USA

ABSTRACT

Log-structured merge (LSM) tree based key-value (KV) stores organize writes into hierarchical batches for high-speed writing. However, the notorious compaction process of LSM-tree severely hurts system performance. It not only involves huge I/O operations but also consumes tremendous computation and memory resources. In this paper, first we find that when compaction happens in the high levels (i.e., L_0, L_1) of the LSM-tree, it may saturate all system computation and memory resources, and eventually stall the whole system. Based on this observation, we present Re-LSM, a ReRAM-based Processing-in-Memory (PIM) framework for LSM-based Key-Value Store. Specifically, in Re-LSM, we propose to offload certain computation and memory-intensive tasks in the high levels of the LSM-tree to the ReRAM-based PIM space. A high parallel ReRAM compaction accelerator is designed by decomposing the three-phased compaction into basic logic operating units. Evaluation results based on db_bench and YCSB show that Re-LSM achieves 2.2× improvement on the throughput of random writes compared to RocksDB, and the ReRAM-based compaction accelerator speedups the CPU-based implementation by 64.3× and saves 25.5× energy.

KEYWORDS

LSM-tree, KV Store, Compaction, Processing-in-Memory

ACM Reference Format:
Qian Wei, Zhaoyan Shen, Yiheng Tong, Zhiping Jia, Lei Ju, Jiezhi Chen, and Bingzhe Li. 2022. Re-LSM: A ReRAM-based Processing-in-Memory Framework for LSM-based Key-Value Store. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '22), October 30-November 3, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3508352.3549392

1 INTRODUCTION

Log-structured merge (LSM) tree based key-value (KV) stores, such as LevelDB [3] and RocksDB [4], have been widely deployed in data centers. The LSM-tree adopts a memory buffer to turn small and random writes into sequential writes and flushes the batched data to disk with a leveled structure. Once the space of one level is consumed up, a compaction process would be triggered to compact all overlapping KV items to their adjacent lower level. However, with the increment of data volume, the compaction process involves large write amplification (or high I/O pressures) and high computation pressures, which eventually result in frequent write stalls and severe throughput decrement.

Many efforts have been performed to mitigate the overhead of the LSM-tree compaction process. To reduce the extra I/O induced by the compaction process, some work proposed to optimize system data structure by exploring data access patterns, such as KV separation [6, 16] and hot/cold data isolation [5, 11]. Some other work turned to implement KV stores on top of the high-capacity and low-latency emerging non-volatile memory (NVM) (e.g., PCM or 3D-xpoint) [7, 14, 23]. The main idea is to maintain the high levels of LSM-tree in NVM to reduce access latency. However, all these I/O optimization designs ignore that the compaction processes in the LSM-tree are also computation-intensive, which would consume up all the computation resources of the system and result in suboptimal performance. On the other hand, to relieve the computation pressure incurred by compaction, some studies [19, 24] proposed to offload compaction tasks to FPGAs and implemented a multi-staged compaction pipeline. However, this approach still involves lots of data movement between disk and memory and between memory and the processing units.

Resistive random-access memory (ReRAM) [17], as an emerging non-volatile memory, provides high density and low-energy consumption. It also supports in-place matrix-vector multiplication (MAC) operations based on the crossbar structure. Many application-specific ReRAM-based process-in-memory (PIM) optimization frameworks have been proposed to accelerate their computation-intensive processes and eliminate data movement between memory and processing units, such as neural networks [8, 15, 25], graph computing [18] and blockchains [20, 21]. Therefore, these appealing features and applications of the ReRAM-based PIM architectures motivate us to take advantage of ReRAM to benefit the LSM-tree based system design.

In this work, we aim to design a ReRAM-based PIM optimization framework for the LSM-tree based KV store to mitigate its computation and memory pressures caused by the compaction processes. To achieve this goal, three challenging issues must be addressed: (1) Which part of the computation and I/O tasks should be offloaded to ReRAM; (2) How to translate compaction operations into basic ReRAM crossbar multiplication modules; (3) How to maximize the parallelism potentials between compaction workflow and ReRAM crossbar design.

*Zhaoyan Shen is the corresponding author (email: shenzhaoyan@sdu.edu.cn).

ICCAD '22, October 30-November 3, 2022, San Diego, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9217-4/22/10…$15.00
https://doi.org/10.1145/3508352.3549392

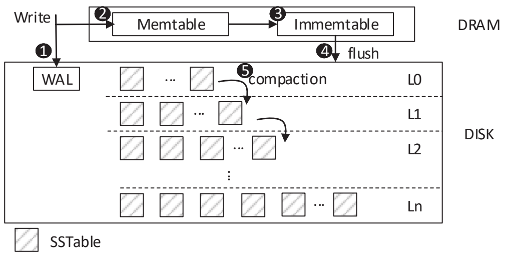


Figure 1: The structure of LSM-tree based KV store.

To address the first challenge, we perform some preliminary experiments to explore the system performance bottleneck of the KV store. We observe that most of the CPU, memory, and I/O resources are consumed when compaction happens in high levels (i.e., from L_0 to L_1) of the LSM-tree. Thus, we adopt the hierarchical DRAM-NVM-DISK hybrid storage architecture in the design of the KV store. DRAM is used as memory buffers to transform small and random KV writes into sequential ones. The ReRAM space is separated into the memory part and processing part. The memory part is used to persist certain components (i.e., L_0, L_1 and WAL (write-ahead log)) of LSM-tree. The processing part is used to perform in-place compaction operations for these components. The disk space is used to accommodate the data belonging to lower levels for the KV store. Besides, to further reduce the I/O overhead, we propose a WAL-to-SSTable transformation method to decrease the flushing I/Os based on the persistency of WAL.

To handle the second and the third challenges, we first analyze the compaction process and separate it into fine-grained phases to make it easier to map the compaction to the basic operations in ReRAM. The compaction process is composed of three stages of decoding-merging-encoding: KV pairs are first obtained by decoding the disk data structure of LSM-tree (i.e. $SSTable$), then ordered through the merging process and encoded into new $SSTables$. In each phase, we build the connection between the compaction and the basic operations in ReRAM by transforming the compaction operations to basic logical operations, and then mapping these logical operations to ReRAM analog crossbar units. To explore the parallelism of the compaction process, we divide keys of the store into different fine-grained subranges, so that the compaction process can be performed for each subrange in parallel.

In summary, in this paper, we propose a ReRAM-based PIM framework for the LSM-based KV store, named Re-LSM, aiming to mitigate the huge computation and I/O overhead caused by the compaction processes. Specifically, we propose to offload certain compaction-frequent high levels of the LSM-tree to ReRAM and perform both CPU- and I/O-efficient in-place compactions. To achieve this goal, a fine-grained ReRAM-based compaction accelerator is proposed to maximize the parallelism of the hardware crossbar structure. Further, a WAL-to-SSTable transformation method is introduced to mitigate the I/O overhead of flushing operations. The simulation results show that the proposed Re-LSM achieves 2.2× improvement on the throughput under db_bench workload compared with RocksDB, and the PIM-based compaction accelerator achieves 64.3× speedup and 25.5× energy saving compared with the CPU-based compaction implementation.

The contributions of this work are summarized as follows:

- We perform plenty preliminary experiments and observe that the system computation and I/O bottleneck mainly come from the resource hungry compaction processes occurred in the high levels of the LSM-tree.
- We design an efficient ReRAM-based PIM framework by offloading the data management of high levels to ReRAM storage space and performing in-place compactions with ReRAM processing engine. A fine-grained compaction strategy based on KV subranges is proposed to maximize the device parallelism.
- We evaluate the proposed Re-LSM with both micro and macro benchmarks. Evaluation results show that Re-LSM can efficiently improve the system performance and reduce the energy consumption.

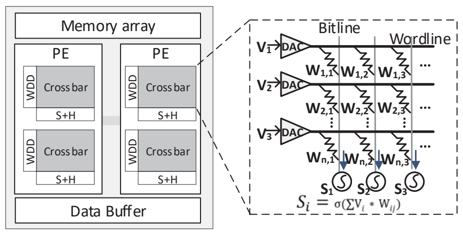


Figure 2: The architecture of a ReRAM-based PIM accelerator.

The rest of this paper is organized as follows. Section 2 introduces the background and motivation of this work. Section 3 presents the proposed Re-LSM design in detail. Section 4 gives the evaluation results and discussions and Section 5 concludes this paper.

2 BACKGROUND AND MOTIVATION

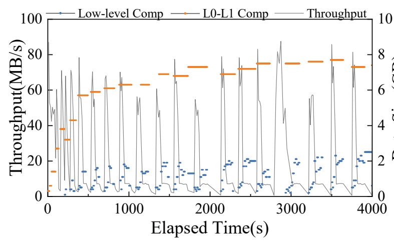
2.1 LSM-tree based KV store

Figure 1 presents a brief structure overview of the LSM-tree based KV store. The structure is mainly separated into memory components and disk components. A flushing process is used to transform memory components to disk components for persistence. A compaction process is used to organize disk components into levels of varied sizes.

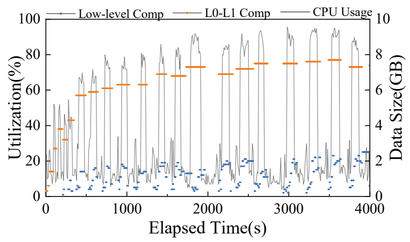
•**Memory components**. The memory components includes a *memtable* (memory table) and an *immemtable* (immutable memory table). The *memtable* is used to buffer the incoming small and random KV pairs and keep them sorted. Once the *memtable* is full, it will be transformed into an *immemtable*. After this, a new *memtable* will be generated to serve the following incoming writes and the *immemtable* will be persisted to disk with a **flushing** process.

•**Disk components**. An *SSTable* is a basic on-disk storage unit of the LSM-based KV store, which is converted by an *immemtable*. The *SSTable* maintains the KV items in several data blocks. When packing the KV items into data blocks, the same prefix of different keys (called the shared prefix) is only saved once in the top KV items, which is called prefix compression. In addition, when compacting data blocks to *SSTables*, the Snappy algorithm [1] is adopted to iteratively compress those repeated bit fields. These two compression algorithms have a same principle, that is, compress the duplicate parts by performing comparison operations.

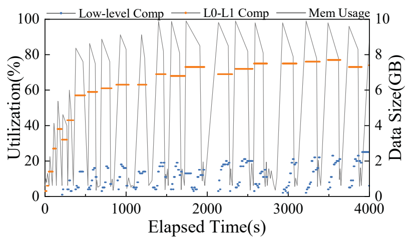
The *SSTables* are logically organized in multiple levels. The size of two adjacent levels is increased exponentially. Generally, the size of the lower level is 10× larger than its adjacent higher level. Except for L_0, all the KV pairs in a level are sorted in order of keys, meaning that the key ranges of *SSTables* on the same level are non-overlapping. The disk also maintains a WAL file for crash recovery.



(a) Throughput.



(b) CPU Utilization.



(c) Memory Utilization.

Figure 3: Random write performance in RocksDB.

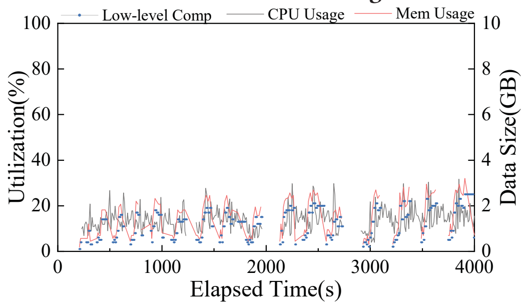


Figure 4: Performance on low-level compaction only.

Each KV pair is first written to the WAL file before being inserted into the *memtable* to promise crash consistency.

If the size of one level exceeds a predefined threshold, a **compaction** process is triggered to compact all overlapping *SSTables* to its adjacent level. The compaction process includes three phases (i.e., decoding, merging and encoding). Specifically, if L_N reaches its capacity, the compaction thread will first pick up all *SSTables* in L_{N+1} that have key range overlap with the *SSTables* selected in L_N into memory and decode the *SSTables* into KV pairs. Then, the KV pairs will be merge sorted together and encoded into *SSTable* formats. These *SSTables* are finally written to the lower level.

2.2 ReRAM-based PIM accelerator

ReRAM is an emerging non-volatile memory that can perform in-memory computations. The ReRAM cell is of a sandwiched structure that contains a top electrode, a bottom electrode, and a metal–oxide layer. ReRAM cells are usually organized into the crossbar array structure for area efficiency. We give the details of a crossbar in the right of Figure 2 and it shows that cells in a ReRAM crossbar are connected by wordlines and bitlines. ReRAM can be switched between a high resistance state (HRS) and a low resistance state (LRS), which can represent the logic "0" and "1" by applying an external voltage. By selecting a wordline i with discharging voltage, while sensing the current flowing at the end of bitline j, we can get the value of cell $W_{i,j}$. A sample and hold (S/H) circuit receives the current and feeds it to a shared ADC unit. By adding a rectifier and a zero-crossing comparator at the Sense Amplifier (SA) front end, the positive or negative current on bitline will be sensed and the current polarity will be obtained. When using multiple cells to represent an integer, and computing the matrix multiply-add calculation (MAC) results of each column Pixel value, the ADC results of each bitline need to be shifted and summed by shift and adder (S&A) to get the final result.

ReRAM is used as an accelerator in many applications since it can perform computation in memory [8, 15, 18, 20, 21, 25]. Figure 2 shows the architecture of the ReRAM-based PIM accelerator, which consists of Memory Array, Processing Element (PE) and Data Buffer. PE is composed of many crossbars, and those crossbars can perform computation in parallel.

2.3 Motivation

For the LSM-based KV store, once compaction happens, its performance would decrease dramatically. To investigate why and how the compaction process effects system performance, we have performed plenty preliminary experiments. We choose the widely used RocksDB as the KV store and use db_bench to issue 80GB KV items for the tests.

Figure 3 shows the system throughput, CPU, and memory usage status during the testing process. As shown, the x-axis represents the elapsed time, the right y-axis represents the involved data sizes of one compaction process, and the left y-axis represents system write throughput, CPU usage and memory usage, respectively. Each orange line represents a L_0-L_1 compaction and the blue lines represent low-level compactions, where the length along the x-axis represents the time of the compaction process. It can be observed that the occurrence of L_0-L_1 compaction involves huge I/O data size and leads to peaks of CPU and memory utilization, up to 90.9% and 98%, respectively. In addition, the write throughput drops sharply and even stalls writing during the L_0-L_1 compaction operations.

In Figure 4, we explore how the system can benefit from removing the resource hungry L_0-L_1 compaction operations. To simulate this process, when flushing operations are triggered in RocksDB, instead of writing the data to L_0, we simply abandon these KV items. For the low-level compaction processes, we record all the compaction information (including time, level, and involved KV sizes) and replay them in this test. As shown in Figure 4, the gray line represents CPU usage and the red line represents memory usage. We can observe that the resource usage has dropped significantly, about 22.4% CPU usage and 19.1% memory usage are saved on average. This observation motivates us to offload and accelerate L_0-L_1 compaction tasks with ReRAM accelerator to mitigate the I/O and computation pressures. By doing so, more resources can be used to do other work and the system throughput can be improved.

3 RE-LSM DESIGN

In this section, we present an overview of the Re-LSM design. We further provide the details for the PIM-based compaction accelerator and the fine-grained compaction process. Finally, we introduce the WAL-to-SSTable transformation method for I/O mitigation.

3.1 Overall Architecture

Figure 5 shows the architecture overview of the proposed Re-LSM design. Re-LSM is mainly composed of the DRAM memory components, the ReRAM components, and the disk components.

DRAM components. Similar to the original LSM-tree design, the DRAM components of Re-LSM is used to accommodate small

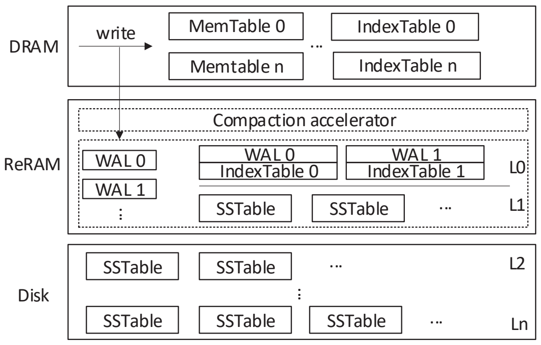


Figure 5: Architecture overview of the Re-LSM.

and random KV writes into sequential batches. To support fine-grained compaction, we separate key ranges into several subranges in the high level (i.e., L_0), and maintain one *memtable* for each subrange. Furthermore, to mitigate the I/O overhead of flushing operations, we replace an *immemtable* with an index table to record the WAL file offsets for each KV pair in *memtable*.

ReRAM components. The ReRAM components are divided into memory part and processing part. For the ReRAM memory part, certain parts of the LSM-tree (i.e., L_0, L_1, and WAL) are offloaded on it. For the processing part, we design a PIM compaction accelerator engine to handle L_0-L_1 compaction tasks.

Disk components. The disk components are the same as the original LSM-tree design except that they only service for storing the KV pairs in the low levels of LSM-tree. The low-level compaction operations are still performed in the original approach.

For write requests, the incoming KVs are first logged in WAL and stored in *memtable* based on their key range. The read request path is same with the original LSM-tree design except that ReRAM will be accessed when reading L_0 and L_1.

3.2 PIM-based Compaction Accelerator

Figure 6 shows the design of the ReRAM-based PIM compaction accelerator, which consists of a Global IO Interface, a Controller, a Global Row Decoder, Memory Array, Data Buffer, and the Compaction Accelerator Engine. The detailed function units are shown in the right of the figure in which the arrow lines represent the data flow. The Global IO Interface is an input/output interface for data exchange. The Controller is an interface between software and hardware to execute simple instructions to control the hardware. The Global Row Decoder is responsible for configuring the wordline of ReRAM crossbars. The memory array stores L_0, L_1 and WAL files of LSM-tree the same as conventional memory sub-arrays [22]. Data Buffer is used to buffer intermediate results for Compaction Accelerator Engine. The engine is divided into three modules: Decoding Module, Merging Module, and Encoding Module, which correspond to the three stages of the compaction process. The engine includes a number of ReRAM crossbars driven by row decoders and performs different operations with modified peripheral circuits. The Data Buffer and the Compaction Accelerator Engine are connected through private data port (i.e. Connection), which provides a high bandwidth for frequent data transferring. The detailed designs of the three phases of Compaction Accelerator Engine are provided in the following to show the procedure of mapping compaction to basic logical operations.

Alogrithm 1 : Encoding Process

```
1: function PREFIX ENCODING(CURRENTENTRY, LASTKEY)
2:     sharedKeySize = getPrefix(currentEntry.key, lastKey)
3:     setLastKey(currentEntry.key)
4:     currentEntry = write(currentEntry.key.skip(sharedKeySize),
   currentEntry.value)
5: end function
6:
7: function SNAPPY COMPRESSION(INPUT, OUTPUT)
8:     while !input.empty() do
9:         match=findFourBytesMatch(input,&literal)
10:        emitLiteral(output,literal,sizeof(literal))
11:        while existFourBytesMatch(input,literal) do
12:            length=findMatchLength(input,literal,&match)
13:            emitCopy(output,match,length)
14:        end while
15:    end while
16: end function
```

Encoding Module. The encoding module is used to encode ordered KV pairs belonging to one *SSTable* into its defined persistent formats with Prefix Encoding and Snappy Compression. In each *SSTable*, keys are prefix-encoded into data blocks where the data block is further compressed based on the Snappy algorithm. Multiple data blocks and their indexes form a complete *SSTable*.

As shown in Algorithm 1, the prefix encoding can be divided into two steps, that is, find the common prefix and record the length of the prefix. These two steps can be performed simply by bit comparisons and addition operations which can be mapped to ReRAM easily. Figure 7 (a) shows an example of the XOR operation unit based on ReRAM. By modifying the sensor amplifier (i.e., SA1), we can make the unit output logical '1' when two cells are both of low resistance status ('1') or high resistance status ('0'), otherwise, output logical '0'. Figure 7 (b) shows an example of the realization of the basic operation of SUM on the ReRAM interleaved array. Since the inherent matrix-vector multiplication operation of ReRAM can support addition operations, a shift and add (S/A) unit is added to shift and add the results of the bitlines to obtain the result of addition operation [13].

The Snappy algorithm is used to compress the interior of a data block. As shown in Algorithm 1, the Snappy compression mainly consists of two loops, that is, select the first small part of the input as the uncompressed field (called $literal$), then compare the remaining part with the $literal$ and record the offset and length of match field (called $copy$). We illustrate the Snappy compression process with a simple example: for "xabababab", the match field is "ab" and the $literal$ is "xab". The final compressed field is represented as: $< literal : "xab" >< copy : offset = 2, length = 4 >$. The process can be also performed as a simple bit comparison and addition operation, which can be accomplished by XOR and SUM as shown in Figure 7. The related implementation of these logic units on ReRAM has been mentioned above.

Merging Module. The merging module is used to sort KV pairs with an increasing order and verify their validity. In LSM-based KV store, the heap sort algorithm with complexity O($nlgn$) is adopted. In Re-LSM, for the sorting part, we can convert it into multiple

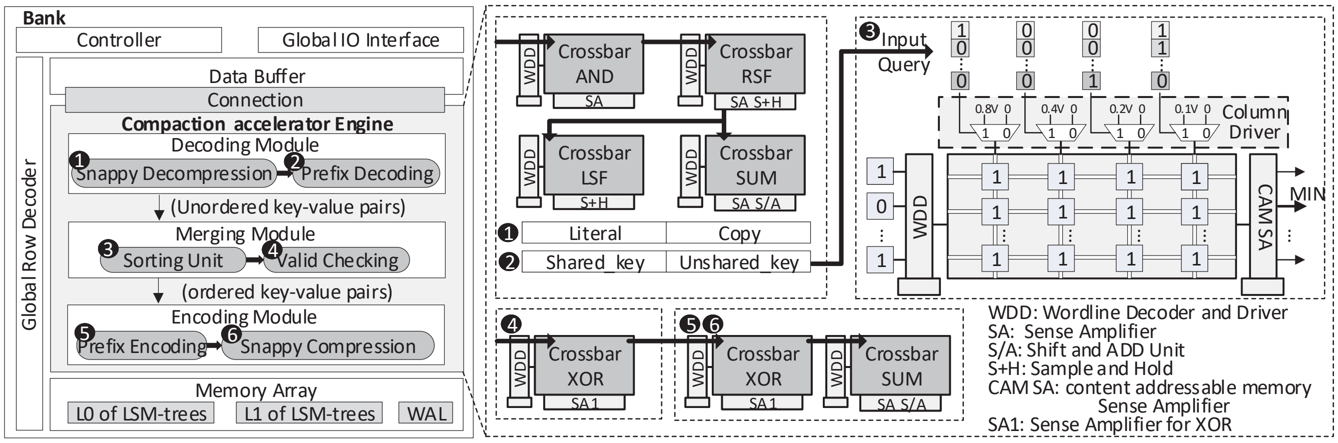


Figure 6: The compaction accelerator architecture.

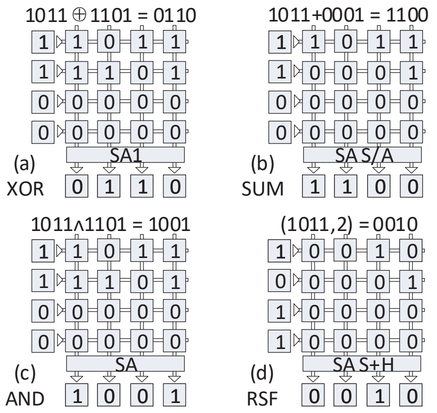


Figure 7: Examples of logical operations on ReRAM.

"MIN" operations. We adopt the closest distance search configuration and search for the data which has the closest distance to zero to accomplish the MIN operation [12]. Figure 6 ❸ shows an example of a MIN query process. The column driver drives the bitlines of the crossbar and maps the input query to the required bitline voltage levels. We modify the bitline driving voltage to provide binary weights to the bits. The CAM SAs are responsible for detecting charging and discharging behaviors of wordlines and finding the binary value nearest to 0. When the minimum value is found, mark the row of data as invalid and repeat this process to find the minimum value in the remaining data.

Considering the relationship between the length of the key and the maximum value of the bitline drive voltage, the key fields are stored in stages. The first m most significant bits of the data are stored in the first stage, and the next m significant bits are stored in the second stage, and so on. During the MIN operation, the bitwise search operation is performed sequentially from the first stage, and the output of one stage determines which rows are to be activated at the next stage. We assign different binary weights to the m effective bits respectively. The bitline with a higher number of bits has a higher activation voltage, thereby obtaining a higher current value output and increasing the accuracy of the minimum value judgment. Considering the limitation of the threshold voltage of non-volatile components, there are only five choices for the activation voltage of the bitline (i.e. 0.1V, 0.2V, 0.4V, 0.8V, 1.6V), and the corresponding number of m bits is limited to 5. Therefore, increasing the number

Alogrithm 2 : Decoding Process

```
function SNAPPY DECOMPRESSION(INPUT, OUTPUT)
    while !input.empty() do
        tag=ReadOneByte(input)
        if tag == LITERAL then
            write(output,literal)
        else
            match = read(literal, copy.offset)
            for j=0 to copy.length / copy.offset do
                write(output,match)
            end for
        end if
    end while
end function

function PREFIX DECODING(CURRENTENTRY, LASTKEY)
    sharedKey = read(lastKey, currentEntry. sharedKeySize)
    currentEntry.key = makeKey(sharedKey, currentEntry.key)
    setLastKey(currentEntry.key)
end function
```

of bits in the key will increase the latency for the completion of the MIN operation. The Valid Checking is used to check whether the entry is marked as "DELETE", which can be accomplished by an XOR operation, as shown in Figure 7 (a).

Decoding Module. The decoding module is used to extract KV pairs from *SSTables*, which mainly includes two phases: decompressing data blocks from *SSTables* (based on the Snappy algorithm) and decoding KV items from data blocks.

As shown in Algorithm 2, the main process of Snappy decompression is to get the match field in $literal$ according to offset and restore the $copy$ field. We obtain the match field by setting the first few bits of $literal$ to 0 through AND operation, then append the match field to the output according to the length of $copy$, which requires division, left shift (LSF) and SUM operations. Figure 7 (c) shows an example of AND operation unit based on ReRAM. By modifying the sensor amplifier (SA), we can make the unit output logical '1' when two cells are both of low resistance status ('1'), otherwise, output logical '0'. The division operation can be converted into a Right Shift (RSF) operation, because the $offset$ and $length$

in the *copy* field are both multiples of 2. Figure 7 (d) shows an example of the RSF operation which performs a two bits right-shift operation. The principle of the design is that we can transform the RSF operation into a matrix-vertor multiplication operation (i.e., the input 1011 multiplied by the matrix stored in ReRAM cells). LSF operation can be transformed into matrix-vertor multiplication operation as same as RSF operation. The related implementation of SUM operation on ReRAM has been mentioned above.

As shown in Algorithm 2, the main process of the prefix decoding phase is to get the shared key from the prior KV pair and merge it with the uncompressed part to generate the complete key. The process can be performed by RSF, LSF and SUM operations and the related implementation on ReRAM have been mentioned above.

The Workflow of Compaction Accelerator Engine. We show the workflow of data processing in Compaction Accelerator Engine with black arrows on the right side of Figure 6. The Decoding Module is firstly used to generate unsorted KV pairs from *SSTables* (see Figure 6 ❶,❷), then the KV pairs are sorted and verified by the Merging Module (see Figure 6 ❸,❹). Finally, the sorted KV pairs are generated into new *SSTables* through the Encoding Module (see Figure 6 ❺,❻).

3.3 Fine-grained Compaction

In the original LSM-tree design, since the key ranges of *SSTable* in L_0 may have overlap, when compaction happens in L_0, it may involve a large number of *SSTables* in L_0 and L_1. This process involves a huge amount of data volume and computation resources and may even stall the whole system. In Re-LSM, we separate the key ranges into several subranges and maintain a *memtable* for each subrange. When flushing the data to L_0, *SSTables* in L_0 are also separated into several subranges. The *SSTables* in each subrange may have overlap, but *SSTables* in different subranges are not overlapped.

When compaction happens in L_0, we can perform the fine-grained compaction to only reclaim the *SSTables* in one subrange to reduce I/O and computation overhead, and the compaction processes of different subranges can also be distributed into different ReRAM processing crossbars and run in parallel.

Figure 8 shows an example of the key subranges partition for the fine-grained compaction. We divide keys into *n* subranges (form subrange 1 to subrange *n*) that do not have overlap. When one *memtable* is filled up, the index table will be flushed to ReRAM and band with the WAL file to be converted into a new *SSTable* in L_0. As shown, one subrange in L_0 may contain multiple *SSTables* with overlapping keys, such as (59, 187) and (0, 165) in subrange 1. We preset a maximum *SSTable* number for each subrange, once the size limit is reached, L_0-L_1 compaction will be triggered. For example, assuming that subrange 1 has reached the rated data size, the fine-grained compaction will execute among (59, 187), (0, 165) of L_0 and (0, 79), (90, 199) of L_1. As mentioned in Section 3.2, we have designed a compaction engine to accomplish the compaction process in ReRAM. Considering the parallelism of ReRAM, we can increase the number of compaction engines deployed to support the parallel execution of fine-grained compaction. That is, *SSTables* in subrange 2 (and subrange 3, subrange n, etc.) can execute compaction operations in parallel with subrange 1.

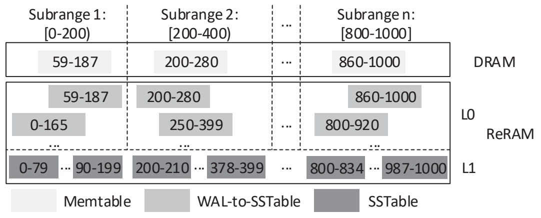


Figure 8: The fine-grained compaction.

3.4 WAL-to-SSTable transformation

In LSM-tree based KV store, the incoming KV items will be first written into a WAL file before being inserted into memory buffer. Hence, before flushing the memory buffer to ReRAM component, the KV items are actually already persistent in the form of WAL. In Re-LSM, to further mitigate the I/O overhead of flushing operations, we propose to maintain an index table to record the KV pairs in memory buffer and their locations in the WAL. When flushing happens, we only flush the index table to ReRAM component to transform the WAL file into a L_0 *SSTable*.

As shown in Figure 5, we maintain an index table for each *memtable*, which records the WAL file offset for each KV pair. The index table and *memtable* are of one-to-one correspondence, such as *memtable* 0 and index table 0. The items in index tables follow the format of (key, location of WAL), in which the location of WAL means the offset of the KV item in a WAL file. Similarly, in Re-LSM,the writes are also first persisted in WAL, then performed in *memtable*. Once a *memtable* is full and the flushing operation is triggered, only the index table is persisted to ReRAM component. The index table is then grouped with its corresponding WAL file, thus creating the new L_0 faker *SSTable* (e.g., WAL 0 and index table 0 in Figure 8). With this transformation process, when flushing operations are triggered, we can replace the expensive writing of memory tables with the more efficient index tables. Finally, when L_0-L_1 compaction is triggered, the involved index tables and the corresponding WAL files are compacted into *SSTables* in L_1.

4 EVALUATION

4.1 Experimental Setup

We implement the proposed Re-LSM scheme based on the LSM-tree based KV store RocksDB [4]. We mainly compare Re-LSM with RocksDB (including RocksDB-SSD and RocksDB-L0L1-NVM). RocksDB-SSD represents the conventional RocksDB with the DRAM-SSD storage hierarchy. Re-LSM and RocksDB-L0L1-NVM are for systems with DRAM-NVM-SSD storage hierarchy. RocksDB-L0L1-NVM simply stores L_0 and L_1 on ReRAM. RocksDB running on the CPU platform is adopted as the baseline. For fair comparison, we set the max number of background compaction threads of RocksDB and RocksDB-NVM as 32. Whereas, for Re-LSM, since the L_0 compactions are done with the PIM accelerator, we only set the max number of compaction threads as 16. Unless stated, other parameters used are set as the default configurations of RocksDB.

For Re-LSM, we separate the key ranges into five isolated subranges and one *memtable* is maintained for each subrange in DRAM. The ReRAM component is simulated by extending the platform NVSim [10]. The ReRAM crossbar is configured as 128*128. The voltage of ReRAM read and write are 0.7V and 2V and the HRS/LRS resistance is 25M/50K. The read/write latency and read/write energy are 19.31ns/50.88ns, 1.08pJ/3.91nJ, respectively. Other related

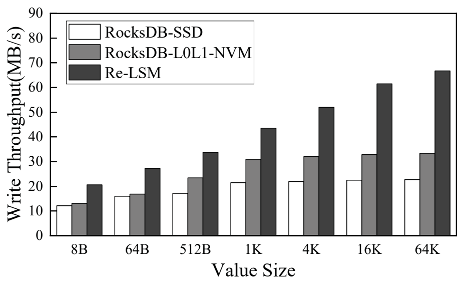


Figure 9: Write throughput with different value size.

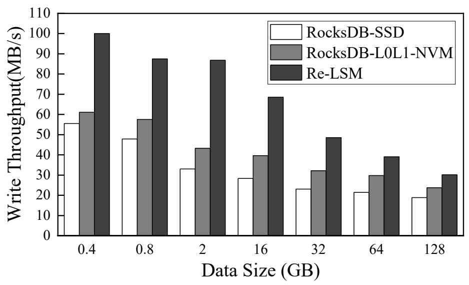


Figure 10: Write throughput with different data size.

circuit parameters are referenced from ISAAC [17]. The capacity of ReRAM applied in Re-LSM is 8GB and we utilize 30% of memory capacity for computation.

For the workloads, we first use db_bench [2] as micro-benchmark to issue random writes to evaluate the performance of Re-LSM under varied KV item sizes and data sets. Further, we adopt YCSB [9] as macro-benchmark to evaluate how Re-LSM performs under read/write mixed workload with different distributions.

4.2 Write Performance Comparison

We first use db_bench to issue 80GB KV random writes with varied value sizes (ranging from 8B to 64KB). Figure 9 presents the write throughput comparison of RocksDB (including RocksDB-SSD and RocksDB-L0L1-NVM) and Re-LSM. Benefiting from the high read and write bandwidth of NVM, RocksDB-L0L1-NVM always outperforms RocksDB-SSD. The I/O amount involved in each L_0-L_1 compaction increases as the value size increases, so the performance improvement of RocksDB-L0L1-NVM also increases. However, due to the limited resources of the CPU used for background compaction, the performance improvement is not that much (on average 1.33×). As shown, Re-LSM outperforms RocksDB across the board. When the item size is 64KB, Re-LSM achieves the highest throughput improvement (i.e., 2.93×) over RocksDB-SSD. The performance improvement mainly rises from two aspects. First, since we offload the *SSTables* in L_0 and L_1 to ReRAM, thus, when compaction happens in L_0, the time consumed by data movement between disk and DRAM could be saved. Second, Re-LSM performs in-place L_0-L_1 compaction in ReRAM, hence, data movement between memory to CPU is eliminated and these resources could be released to serve more incoming requests.

Figure 10 shows the write throughput comparison with different KV data set sizes (ranging from 0.4GB to 128GB). In this evaluation, we fix the key size of each KV item as 8B, and the value size as 1KB. Similarly, Re-LSM performs much better than RocksDB for all data sets. With the data set size increase, the system throughputs of both RocksDB and Re-LSM keep decrease and the improvement ratio of Re-LSM over RocksDB shows variant trends. The reason for this is that, in the beginning, when the data size is small (less than 2GB), the throughput drop is mainly caused by L_0-L_1 compaction. Since Re-LSM totally offloads L_0-L_1 compaction to ReRAM, the improvement ratio presents a rising trend. When the data size is larger than 2GB, the proportion of low-level compaction increases and the performance gains achieved by offloading L_0-L_1 compaction to ReRAM would be diluted to certain extent, resulting in a dropping trend in performance improvement.

4.3 Compaction Performance Comparison

We further evaluate the efficiency of the proposed ReRAM-based compaction accelerator. In this evaluation, we also issue 80GB KV random writes with varied KV item sizes. We collect the average throughput and energy consumption statistics of L_0-L_1 compaction operations and show the results in Figure 11 and Figure 12.

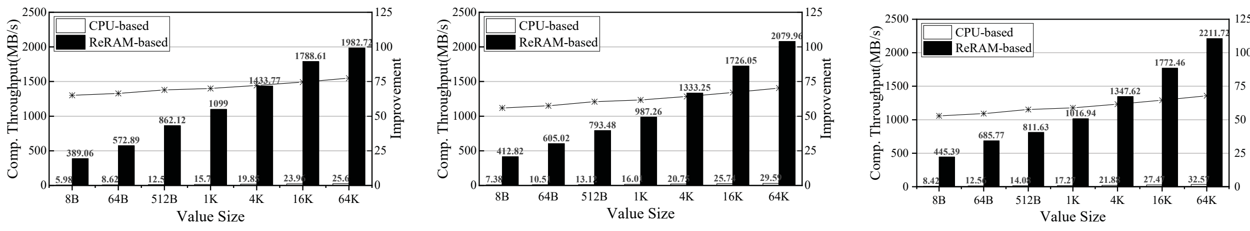
1) Compaction throughput: The compaction throughput is defined as *The total amount of data involved in compaction / computation time of compaction*. As shown in Figure 11, the proposed ReRAM-based compaction accelerator performs much faster than CPU-based implementation under different value sizes, with an average of 64.3× improvement. With the increment of value size, the compaction throughputs of both these two implementations also increase. This is because when value size increases, the number of KV pairs involved in the compaction process would decrease, and the amount of computation would be reduced accordingly, which contributes to the throughput improvement. Meanwhile, with the increment of value size, the improvement ratio of the ReRAM-based accelerator over CPU implementation also increases slightly. We also present the performance of improvement as a line in Figure 11. By comparing (a), (b) and (c), it can be seen that the performance improvement brought by the compaction accelerator gradually decreases as the key size increases. This is because the prefix compression and merging processes in the compaction tasks are closely related to the key size. For smaller key size, the number of bits to be compared in compaction tasks is less, thus the task execution speed becomes faster, which in turn leads to better overall throughput improvement.

2) Energy consumption: Figure 12 presents the normalized energy consumption comparison of the these two implementations. On average, the energy saving of the ReRAM-based compaction accelerator is 25.5× than the CPU implementation. The energy saving benefits form the highly parallel computation process and the elimination of data movement overhead of memory access power which CPU-based implementations contain. Similarly, with the increment of key size, the energy saving improvement reduces. This is due to the increase in computing resource consumption caused by the increase of keys.

4.4 Breakdown Analysis

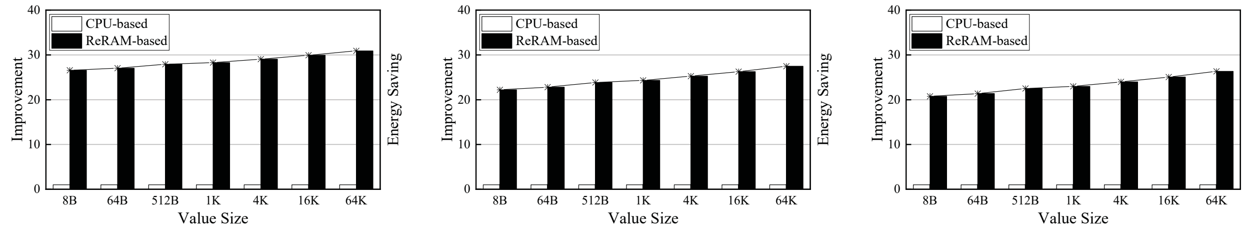
In this section, to investigate the influence of each proposed scheme on write performance, we superpose each scheme one by one until it becomes Re-LSM as described below:

- **RocksDB-SSD:** It represents the original RocksDB on the DRAM-SSD hierarchy.
- **RocksDB-L0L1-NVM:** It represents the original RocksDB on a DRAM-NVM-SSD hierarchy in which stores L_0 and L_1 in NVM.
- **RocksDB-WAL-to-SSTable:** On the basis of RocksDB-L0L1-NVM, the WAL-to-SSTable transformation strategy is added.
- **Re-LSM-k:** On the basis of RocksDB-WAL-to-SSTable, the L_0-L_1 compaction task is offloaded to the ReRAM component



(a) 8-Byte key. (b) 32-Byte key. (c) 64-Byte key.

Figure 11: Compaction throughput with different KV sizes.



(a) 8-Byte key. (b) 32-Byte key. (c) 64-Byte key.

Figure 12: Energy comparison with different KV sizes.

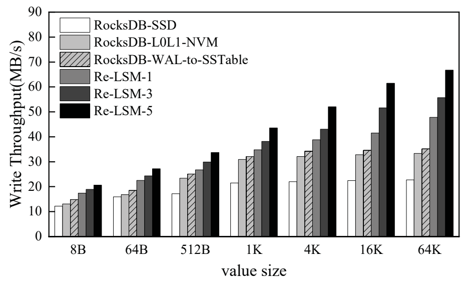


Figure 13: Breakdown analysis.

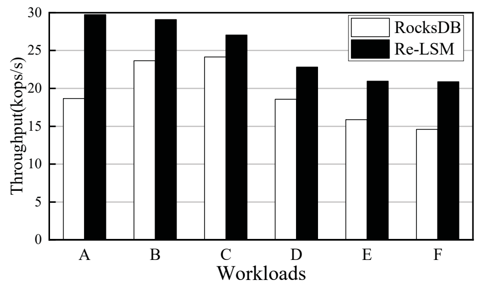


Figure 14: YCSB throughput.***The description of workloads are as follow: A, 50% updates, 50% reads; B, 5% updates, 95% reads; C, 100% reads; D, 5% updates, 95% reads; E, 5% updates, 95% scans; F, 50% read-modify-write, 50% reads.***

for execution. The value k represents that the key range is divided into k subranges. Re-LSM-1 represents no fine-grained compaction strategy.

Figure 13 shows the throughput of these implementations with varied KV sizes. As shown in the figure, Re-LSM-5 achieves the highest throughput improvement over the original RocksDB among all these implementations. By integrating the WAL-to-SSTable transformation strategy and the fine-grained compaction strategy, the write throughput increases for all KV item sizes set.

As shown, the WAL-to-SSTable transformation strategy can improve the performance of the original RocksDB by 1.43× on average, which is mainly benefited from offloading the flushing tasks to the ReRAM component. We further offload the L_0-L_1 compaction tasks to the ReRAM component, and the performance is further improved by 1.66× on average. The experimental result shows that the write stall time caused by the compaction process is significantly longer than that of the flushing process. Therefore, offloading compaction tasks brings a higher benefit. In addition, as the number of subranges increases, the overall performance continues rising as the parallelism of the compaction tasks increases.

4.5 Evaluation on YCSB

To evaluate how Re-LSM performs under workloads with mixed request of different distributions, we further test Re-LSM and RocksDB by using the six workloads from the YCSB cloud suite. We first saturate the KV data store by issuing 80GB dataset with 1KB KV item size, and then we run the workloads A-F, as described in Figure 14, to collect the system throughput.

As shown in Figure 14, Re-LSM outperforms RocksDB for all workloads. We can observe that Re-LSM performs better for workloads that are composed of more writes. For the write-intensive workload A, Re-LSM performs the best, achieving a 63% performance improvement. For the read-only workload C, the throughput of Re-LSM also increases 12%, since ReRAM provides a higher read speed than disk when accessing L_0 and L_1 of LSM-tree.

5 CONCLUSION

In this paper, we propose a ReRAM-based PIM framework for LSM-based KV store, called Re-LSM, to mitigate the computation and memory pressure during the compaction processes. A ReRAM-based PIM compaction accelerator with high efficiency is designed to speed up the high-level compaction process. Furthermore, we accomplish fine-grained high-level compaction by dividing keys into different subranges. Finally, we introduce a WAL transformation strategy to mitigate the I/O overhead of the system. Experimental results show that our proposed Re-LSM effectively improves the overall throughput.

6 ACKNOWLEDGMENTS

The work described in this paper is partially supported by the National Natural Science Foundation of China under Grant 92064008, the CCF-Huawei Huyanglin Project CCF-HuaweiST2021002, the following NSF awards 2208317, and 2204657.

REFERENCES

[1] 2010. Snappy. https://en.wikipedia.org/wiki/Snappy.
[2] 2013. db_bench. https://github.com/facebook/rocksdb.
[3] 2013. LevelDB. https://github.com/google/leveldb.
[4] 2013. RocksDB. https://rocksdb.org.
[5] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *ATC'17*.
[6] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *USENIX ATC'18*.
[7] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *FAST'21*.
[8] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In *ISCA'16*.
[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SOCC'10*.
[10] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P. Jouppi. 2012. NVSim: A Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory. *TCAD'12* (2012).
[11] Kecheng Huang, Zhiping Jia, Zhaoyan Shen, Zili Shao, and Feng Chen. 2021. Less is More: De-amplifying I/Os for Key-value Stores with a Log-assisted LSM-tree. In *ICDE'21*.
[12] Mohsen Imani, Saransh Gupta, Atl Arredondo, and Tajana Rosing. 2017. Efficient query processing in crossbar memory. In *ISLPED'17*.
[13] Mohsen Imani, Yeseong Kim, and Tajana Rosing. 2017. MPIM: Multi-purpose in-memory processing using configurable resistive memory. In *ASP-DAC'17*.
[14] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *ATC'18*.
[15] Yun Long, Taesik Na, and Saibal Mukhopadhyay. 2018. ReRAM-Based Processing-in-Memory Architecture for Recurrent Neural Network Acceleration. *VLSI'18* (2018).
[16] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *TOS'17* (2017).
[17] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *ISCA'16*.
[18] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Helen Li, and Yiran Chen. 2018. GraphR: Accelerating Graph Processing Using ReRAM. In *HPCA'18*.
[19] Xuan Sun, Jinghuan Yu, Zimeng Zhou, and Chun Jason Xue. 2020. Fpga-based compaction engine for accelerating lsm-tree key-value stores. In *ICDE'20*.
[20] Fang Wang, Zhaoyan Shen, Lei Han, and Zili Shao. 2019. ReRAM-based processing-in-memory architecture for blockchain platforms. In *ASPDAC'19*.
[21] Qian Wang, Tianyu Wang, Zhaoyan Shen, Zhiping Jia, Mengying Zhao, and Zili Shao. 2019. Re-Tangle: A ReRAM-based Processing-in-Memory Architecture for Transaction-based Blockchain. In *ICCAD'19*.
[22] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. 2015. Overcoming the challenges of crossbar resistive memory architectures. In *HPCA'15*.
[23] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *ATC'20*.
[24] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. 2020. Fpga-accelerated compactions for lsm-based key-value store. In *FAST'20*.
[25] Yuhao Zhang, Zhiping Jia, Yungang Pan, Hongchao Du, Zhaoyan Shen, Mengying Zhao, and Zili Shao. 2020. Pattpim: A practical reram-based dnn accelerator by reusing weight pattern repetitions. In *DAC'20*.