

Revisiting Multi-threaded Compaction in LSM-trees: Enabling Compaction Pipelining

Hongsu Byun*
Sogang University
Republic of Korea
byhs@sogang.ac.kr

Honghyeon Yoo*
Sogang University
Republic of Korea
yhh@sogang.ac.kr

Sungyong Park†
Sogang University
Republic of Korea
parksy@sogang.ac.kr

Abstract

We reveal that modern LSM-tree multi-threaded compaction suffers from limited cross-level parallelism, which prevents concurrent compactions across multiple levels. This limitation leads to an imbalance in thread assignment and causes throughput to saturate even when more threads are added. To address this limitation, we propose a compaction strategy called DOWNFORCE. DOWNFORCE enables multiple compactions to be executed across levels by introducing non-blocking pipelined compaction, allowing level-wise compactions to proceed simultaneously. This resolves thread imbalance and achieves fully multi-threaded compaction. DOWNFORCE is implemented in RocksDB, a representative LSM-tree-based key-value store, and supports both leveled and tiered compaction. In our evaluation, leveled compaction enhanced with DOWNFORCE achieves an average of 1.44× higher thread-level parallelism and delivers up to 1.81× higher throughput under write-intensive workloads, compared to the conventional multi-threaded leveled compaction.

CCS Concepts

• Information systems → Key-value stores; Flash memory.

Keywords

Key-Value Store, Log-Structured Merge-tree, Parallel Processing

ACM Reference Format:

Hongsu Byun, Honghyeon Yoo, and Sungyong Park. 2025. Revisiting Multi-threaded Compaction in LSM-trees: Enabling Compaction Pipelining. In *54th International Conference on Parallel Processing (ICPP '25)*, September 08–11, 2025, San Diego, CA, USA. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3754598.3754675>

1 Introduction

LSM-tree. The traditional log-structured merge-tree (LSM-tree) [20] was designed to maximize the benefits of sequential writes on magnetic storage devices such as HDDs. It achieves high write performance by ingesting data in a sequential manner. In the background, a compaction [11] process runs to remove duplicate entries and

sort the data. Since compaction reorganizes the append-only data layout, it helps maintain read performance while reducing space amplification. When compaction is delayed, read performance suffers and space amplification increases, making compaction a critical factor in overall LSM-tree performance. In the traditional LSM-tree, compaction was inherently executed using a single thread due to the nature of the design.

Compaction Optimization. Unlike traditional HDDs, modern SSDs provide internal parallelism, which has led LSM-tree to evolve from single-threaded to multi-threaded compaction [12]. This shift has significantly improved compaction throughput. However, performance degradation due to compaction delays still occurs under heavy write workloads. In extreme cases, it can even lead to I/O blocking events known as write stalls [10, 27, 28]. As a result, accelerating compaction has become a widely studied topic in LSM-tree research. Prior approaches include software-based techniques such as optimizing compaction policies [3–5], improving compaction scheduling [1, 28], or redesigning the LSM-tree structure itself [17, 21]. Other efforts utilize hardware accelerators like FPGAs [30], GPUs [24], and DPUs [7], or explore new storage media such as Non-Volatile Memory [6, 14, 27].

Revisit Multi-threaded Compaction. It is widely accepted without doubt that multi-threaded compaction delivers higher throughput than single-threaded compaction. However, its effectiveness has surprisingly not been examined in detail. Despite numerous studies on accelerating compaction, there has been little skepticism about the existing multi-threaded compaction itself. In this work, we begin by re-evaluating whether the current multi-threaded compaction truly works as intended.

Limitation of Multi-threaded Compaction. We identify and experimentally verify two key limitations of multi-threaded compaction: (i) it cannot perform compactions concurrently across different levels, and (ii) thread workloads are unevenly assigned, often concentrated within a single level. As a result, increasing the number of compaction threads does not improve throughput beyond four threads, and at most, only six threads are effectively utilized. A detailed analysis is provided in Section 3. Due to these limitations, allocating more threads to the compaction thread pool yields diminishing returns and fails to enhance overall throughput. **DOWNFORCE.** To overcome the limitations of existing multi-threaded compaction, we propose a compaction strategy called DOWNFORCE. The key idea of DOWNFORCE is to enable concurrent compactions within the same level—an operation previously not allowed—and to pipeline these tasks so that multiple threads can participate in compaction simultaneously. We refer to this pipelined compaction approach as DF-Compaction. With DF-Compaction, parallel compactions can occur across multiple levels, resolving the

*These authors are first co-authors and have contributed equally.

†S. Park is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ICPP '25, San Diego, CA, USA

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2074-1/25/09

<https://doi.org/10.1145/3754598.3754675>

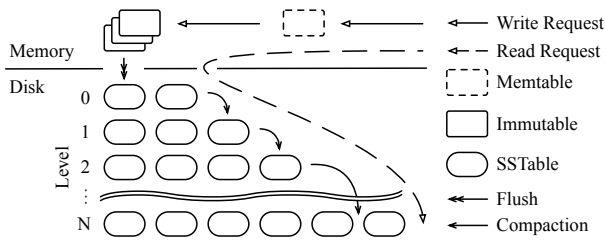


Figure 1: The architecture of the LSM-tree.

imbalance of thread usage between levels and ultimately enabling fully multi-threaded compaction.

Allowing DF-Compaction within a single level can lead to unexpected data accumulation in the next level, potentially degrading read performance. Moreover, if the completion order of compaction jobs is not properly managed, older versions may overwrite newer ones, leading to version inconsistency and violating the correctness guarantees of the LSM-tree. To address these issues, we introduce intra DF-compaction within the same level, which reorganizes the intermediate data to preserve read performance and maintain the LSM-tree structure. Additionally, we adopt a consistency-guaranteed commit mechanism that enforces the correct compaction completion order, thereby preventing versioning inconsistencies.

Contributions. Our contributions are summarized as follows:

- We identify and analyze why multi-threaded compaction in the current LSM-tree fails to improve throughput despite increasing the number of threads.
- To overcome the limitations of existing multi-threaded compaction, we propose DOWNFORCE, a strategy that enhances parallelism by enabling compaction pipelining across levels.
- DOWNFORCE is designed to be compatible with various compaction policies, and we implement it in RocksDB for both leveled and tiered compaction strategies.
- In our evaluation, DOWNFORCE improves thread parallelism by 1.44× and write throughput by 1.81× compared to the baseline leveled compaction.

2 Background

This section describes the structure and characteristics of the LSM-tree, as well as their core operations, as used in various key-value stores.

LSM-tree. Figure 1 illustrates the structure and key operations of the LSM-tree. The LSM-tree is well suited for handling write-intensive workloads. Based on the traditional memory hierarchy, they use fast main memory as a write buffer and persistent storage as the main data repository. These are referred to as the memory component and the disk component, respectively. The memory component is composed of memtables, an in-memory data structure that stores incoming key-value pairs from write requests. Managed by structures such as skip list, the memtable maintains keys in sorted order. Once it fills up, it is converted into an immutable memtable, which is flushed to disk in the background. Each flushed immutable memtable becomes a file on disk, containing sorted key-value pairs,

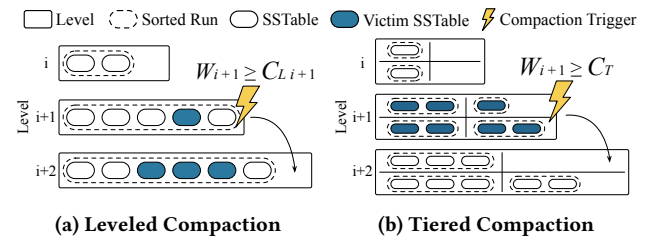


Figure 2: Representative compaction strategies.

and is referred to as a Single Sorted Table (SSTable). The disk component is organized into multiple levels, each containing several SSTables. As level sizes grow by a fixed ratio T , each subsequent level can hold more SSTables or key-value pairs. Let N be the total volume of inserted data, B be the size of first level. At this point, the size of each level L_i can be approximated as $B \cdot T^{i-1}$, and the total number of levels in the LSM-tree for a dataset of size N can be estimated as $L = \lceil \log_T(\frac{N}{B}) \rceil$. The symbols and notations used throughout the paper are summarized in Table 1.

Compaction. We now describe the characteristics and side effects of compaction. Compaction is one of the most important background operations in the LSM-tree, responsible for eliminating redundant data in the disk component. Since the LSM-tree operates in an append-only manner, all inserts, deletes, and updates are handled as write operations (i.e., out-of-place updates). As a result, multiple entries with the same key commonly exist within a single LSM-tree. These redundant entries not only consume additional storage space but also degrade query (read) performance. Compaction is both CPU-intensive and I/O-intensive, as it involves repeatedly moving large volumes of data between memory and disk. In practice, a compaction operation consists of reading multiple SSTables from the disk component, performing a merge sort, and then writing the results back into new SSTables. As a result, even a unique key-value pair written only once to the LSM-tree may be rewritten to disk multiple times due to background compaction processes. This behavior is referred to as write amplification (WA). **Sorted Run.** In the LSM-tree, a sorted run refers to a sequence of contiguous key-value pairs maintained in sorted order. The smallest unit of a sorted run is a single memtable or SSTable, and in the disk component, multiple SSTables can collectively form a sorted run. Depending on the specific configuration of the LSM-tree, each level may contain either a single sorted run or multiple sorted runs.

Weight of Sorted Runs. When and how compaction is performed significantly affects both its behavior and associated side effects. Two classic compaction policies commonly used in the LSM-tree are leveled and tiered compaction. These two policies differ in their compaction triggering conditions, which in turn lead to distinct characteristics in the disk component. For clarity and precision in our explanation, we define W_i as the weight of sorted runs in L_i to represent both the size and number of sorted runs. That is, in this paper, the term weight of sorted runs reaching the compaction threshold is used to represent either of the following two conditions. First, in leveled compaction, it means that the *size* of the sorted run has reached the threshold. Second, in tiered compaction, it means that the *number* of sorted runs has reached the threshold.

Leveled Compaction. In this case, only a single sorted run is allowed per level. Although each level may contain multiple SSTables, they must collectively maintain a globally sorted order. As a result, all SSTables within the same level have non-overlapping key ranges. In leveled compaction, the size of each level serves as the compaction threshold, which we denote as C_{L_i} .

When a level exceeds its size threshold, one or more SSTables from that level are selected and merged into the adjacent bigger level. During this process, SSTables in the output level that have overlapping key ranges with the selected SSTables from the input level are also chosen and merged together. Figure 2(a) illustrates the LSM-tree structure with a single sorted run per level and shows an example of compaction occurring in this setting. In this example, a compaction is triggered once W_{i+1} exceeds the threshold $C_{L_{i+1}}$. One SSTable from L_{i+1} and three overlapping SSTables from L_{i+2} are selected and merged together, with the result written back to L_{i+2} . If this compaction causes the output level to exceed its size threshold, additional compactions may be triggered in a cascading manner.

Leveled compaction offers two main advantages: high read performance and low space amplification, due to its well-organized data layout across levels. Since each level contains a single sorted run, data can be quickly located by reading just one run per level.

On the downside, leveled compaction causes frequent compactions, leading to high write amplification and reduced write performance. Data flushed to the disk component is rewritten during compaction, along with overlapping key range data from the lower level. On average, the write amplification increases by a factor of $\frac{T}{T-1}$ per level, as approximated in Equation 1.

$$WA_{LV} \approx 1 + \frac{T}{T-1} \log_T\left(\frac{N}{B}\right) \quad (1)$$

Tiered Compaction. With the tiered compaction policy, multiple sorted runs can exist within a single level. As a result, SSTables with overlapping key ranges may coexist in the same level. In the tiered compaction, the number of sorted runs in a level serves as the compaction threshold, which we denote as C_T .

When the number of sorted runs exceeds this threshold, all sorted runs in that level are merged into a single sorted run and written to the next level. Figure 2(b) shows the LSM-tree structure with multiple sorted runs per level and an example of compaction in this setting. Compaction is triggered when W_{i+1} exceeds C_T , selecting all sorted runs in L_{i+1} , merging them, and writing the result to L_{i+2} . This can cause a cascading effect if the number of sorted runs in the output level also exceeds the threshold.

Table 1: Symbols and Descriptions.

Symbol	Descriptions
T	Level increase ratio
L	Number of levels in LSM-tree
L_i	i -th level
W_i	Weight of sorted runs in L_i (Bytes or Count)
N	Size of total KV entries (Bytes)
B	Size of the first level L_1 (Bytes)
C_{L_i}	Compaction threshold for L_i in leveled (Bytes)
C_T	Compaction threshold for tiered (Count)

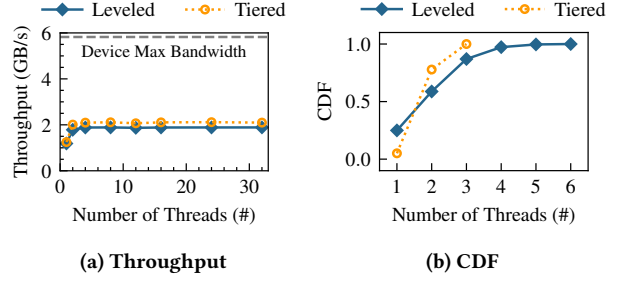


Figure 3: Throughput and parallelism under different compaction strategies. (a) Throughput with varying numbers of compaction threads. (b) Cumulative distribution function of concurrently active compaction jobs in a 16-thread pool.

Tiered compaction provides high write performance and low write amplification due to less frequent compactions. Since it allows redundancy within a level, data can be written without waiting for duplicate removal. Flushed data remains in a level until C_T sorted runs accumulate, then it is merged and moved to the next level. This results in a write amplification of approximately $\frac{1}{C_T}$ per level, as shown in Equation 2.

On the downside, tiered compaction suffers from reduced read performance and high space amplification due to data redundancy within each level. Since up to C_T sorted runs can exist per level, the logical data size may grow up to C_T times larger than the physical size in the worst case.

$$WA_T \approx 1 + \frac{T}{T-1} \frac{\log_T\left(\frac{N}{B}\right)}{C_T} \quad (2)$$

3 Problem Analysis

Limitations of Compaction Scalability. The primitive LSM-tree was originally designed to maximize the benefits of sequential writes on magnetic storage devices such as hard disks. Due to this characteristic, compaction was executed using a single thread. With the emergence of high-speed flash-based SSDs, modern LSM-tree-based key-value stores (LSM-KVS) began to support multi-threaded compaction to leverage the internal parallelism of such devices. While it has been widely assumed that multi-threaded compaction naturally outperforms single-threaded compaction in terms of throughput, the actual extent and nature of its benefits remain unclear. Thus, we first evaluated the effectiveness of multi-threaded compaction.

Figure 3(a) shows the compaction throughput as the number of compaction processing threads increases, measured using the `db_bench` [8] with the `FillUniqueRandom` workload in `RocksDB` [9], a representative LSM-KVS. Detailed experimental settings are provided in Section 6. For both leveled and tiered compaction, throughput saturates with 4 threads, thereby failing to fully utilize the available disk bandwidth. Notably, this behavior persists despite increasing I/O thread counts, key-value sizes, and memtable size and count to boost the overall write rate. This indicates that merely increasing the number of compaction threads does not improve throughput.

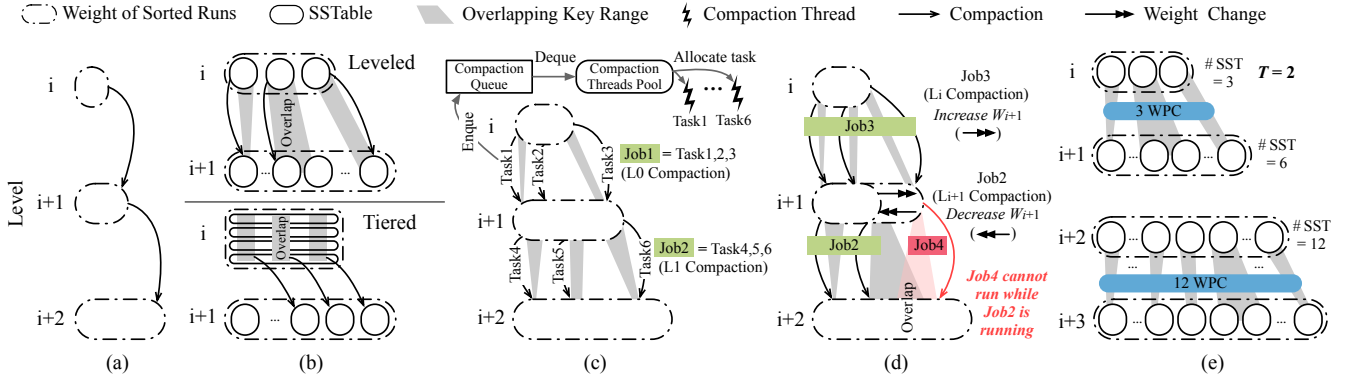


Figure 4: (a) Cross-level Parallelism Compaction (CPC), (b) Within-level Parallelism Compaction (WPC) in leveled and tiered compaction, (c) Multi-threaded compaction combining CPC and WPC, (d) Limitations of CPC, (e) Limitations of WPC.

To determine whether compaction threads beyond 4 are effectively utilized, we tracked the number of concurrently running compaction threads. Figure 3(b) shows the cumulative distribution function (CDF) of active compaction threads sampled every 0.1 seconds in an experiment with a 16-thread pool. We observe that leveled and tiered compaction utilize at most 6 and 4 threads, respectively, indicating limited effectiveness of multi-threaded compaction. To understand the root cause of this limitation, we address the following question.

- 1) **What exactly** is multi-threaded compaction?
- 2) **How** does multi-threaded compaction **work**?
- 3) **Why** is there a **limitation** of multi-threaded compaction?

The answer to these questions applies consistently to both leveled and tiered compaction. For clarity, as described in Section 2, note that the weight of sorted runs (W_i) refers to the size of a sorted run in leveled compaction and to the number of sorted runs in tiered compaction.

Explanation of Multi-threaded Compaction. We begin by closely analyzing how multi-threaded compaction affects compaction parallelism and identify two distinct forms of parallelism. Compaction parallelism can be categorized into Cross-level Parallelism and Within-level Parallelism. Figure 4 illustrates these two types of parallelism in the context of multi-threaded compaction. For clarity, we denote compaction from level i to level $i+1$ as L_i compaction.

1) Cross-level Parallelism Compaction (CPC). Cross-level Parallelism Compaction (CPC) increases parallelism by performing compactions across multiple levels concurrently. For example, as shown in Figure 4(a), L_i , L_{i+1} , and L_{i+2} compactions can run in parallel.

2) Within-level Parallelism Compaction (WPC). Within-level Parallelism Compaction (WPC), also known as partial merge[5], refers to dividing a compaction operation within a single level into multiple tasks that can be processed in parallel. Figure 4(b) illustrates WPC in both leveled (upper) and tiered (lower) compaction. When an L_i compaction is triggered, WPC in leveled compaction divides tasks based on overlapping key ranges between sorted runs in L_i and L_{i+1} , while in tiered compaction, tasks are divided based on overlapping ranges among sorted runs within L_i .

Works of Multi-threaded Compaction. Figure 4(c) illustrates how multi-threaded compaction actually operates. For example, when an L_i compaction (job1) is triggered, WPC splits job1 into tasks (task1–task3) based on overlapping key ranges. These tasks are enqueued into the compaction queue and scheduled for execution according to priority. As the tasks of job1 are processed, W_{i+1} gradually increases. Once it reaches the compaction threshold, a new L_{i+1} compaction (job2) is triggered and similarly divided into tasks (task4–task6), which are also enqueued. To process compaction tasks efficiently, all available threads in the compaction thread pool are utilized, allowing tasks from job1 and job2 to be processed concurrently through CPC. This parallelism enables the weight of sorted runs in each level to grow independently and incrementally, thereby preserving the structure of the LSM-tree.

Limitation of Multi-threaded Compaction. Then, why does the current multi-threaded compaction fail to improve throughput? We identify two key problems behind this limitation. The evaluation results related to these issues are presented in Section 6.2.

1) Blocked Multiple CPC. In CPC, only one compaction job can be processed per level at a time, and any additional CPC jobs targeting the same level are blocked. Consider the scenario illustrated in Figure 4(d), where an L_{i+1} compaction (job2) is followed by an L_i compaction (job3). Job2 decreases the W_{i+1} , while job3 increases it. As job2 and job3 proceed concurrently, job3 may again cause the weight of sorted runs in L_{i+1} to reach the compaction threshold. However, a new L_{i+1} compaction (job4) cannot be initiated until job2 is completed. This is because the compaction range of job4 may overlap with that of job2, which could break the sorted order of the SSTables if executed concurrently. To ensure strong sorted run consistency, existing LSM-tree implementations disallow concurrent CPC jobs on the same level.

2) Imbalanced WPC Between Levels. WPC is only triggered when compaction occurs at a specific level, which leads to an imbalance where compaction tasks are concentrated on a single level rather than being distributed across multiple levels. As shown in Figure 4(e), even if compactions are triggered at both L_i and L_{i+2} , their corresponding WPC tasks are not executed concurrently, and instead, WPC at one level is prioritized. This imbalance fundamentally stems from the inability to execute multiple CPC in parallel across different levels.

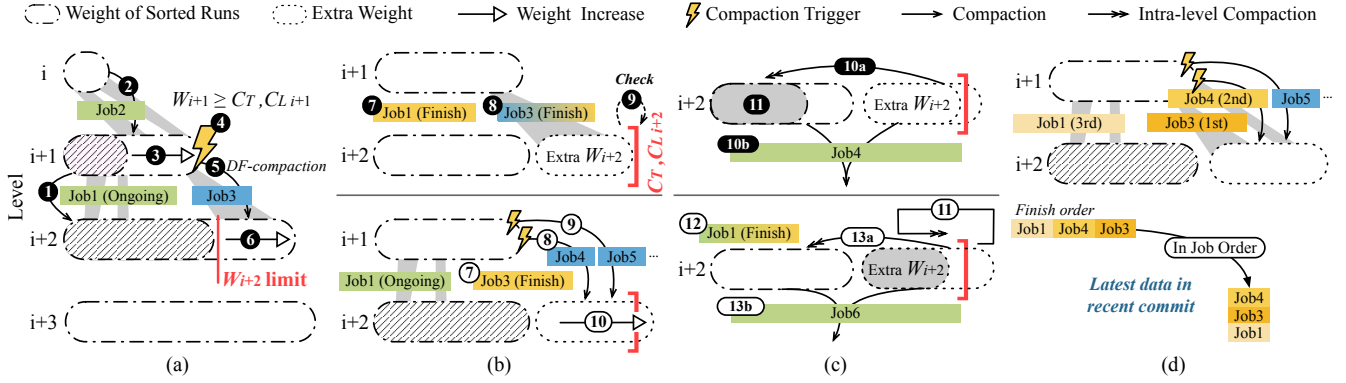


Figure 5: Step-by-step Illustration of DOWNFORCE Operation. (a) Step 1: Trigger conditions for DF-Compaction (b) Step 2: Relaxed sorted run management in DF-Compaction (c) Step 3: Intra DF-Compaction (d) Step 4: Consistency-guaranteed commit

4 Why Compaction Needs to Be Faster

In Section 3, we showed that current multi-threaded compaction is inefficient and does not work as intended. This raises an important question: Why should compaction multi-threading work well? In other words, why is it important to process compaction faster? Compaction acceleration is a well-studied topic in LSM-tree research, primarily because delayed compaction leads to several critical issues. When compaction is delayed, the following problems can occur:

Chain of Problems in Compaction Delay. Compaction delay refers to the case where sorted runs are generated faster than they can be compacted, resulting in a backlog of pending compactions. As the weight of sorted runs increases, more sorted runs must be searched during reads, which degrades read performance. To prevent the continuous growth of sorted run weight, modern LSM-tree triggers a write stall [1, 10, 27, 28]—an I/O blocking mechanism—when the weight reaches a threshold, even if compaction is still in progress. Write stalls block all I/O operations until the delayed compaction completes, causing severe throughput degradation and latency spikes. Furthermore, delayed compaction postpones the removal of redundant key-value pairs, thereby increasing space amplification.

Better the Devil You Know. As discussed in Section 7, numerous prior studies have explored ways to accelerate compaction using software techniques or hardware such as Non-Volatile Memory, FPGAs, GPUs, and DPUs. However, these approaches often require modifications to the LSM-tree structure or rely on additional hardware, making them difficult to adopt in practice. In this context, we revisit the effectiveness of compaction multi-threading, a technique already integrated into modern systems. We identify a key limitation in how current multi-threaded compaction is underutilized and, to address this, we propose DOWNFORCE, a new approach that maximizes compaction pipelining.

5 DOWNFORCE: Design and Implementation

This section presents the design principles of DOWNFORCE and describes its corresponding architecture.

5.1 Design Principle

To address the limited throughput improvement of multi-threaded compaction in the current LSM-tree, we incorporate the following three design principles into our approach.

Principle #1: Compact Down. In an ideal CPC scenario, compactions targeting different output levels should be able to run concurrently without conflict. However, as illustrated in the example in Section 3, a delayed L_{i+1} compaction can block the execution of an L_i compaction. This blockage occurs because preserving the LSM-tree’s structural invariants takes precedence over compaction. Specifically, each level strictly avoids exceeding its configured weight W_i , meaning that compaction cannot proceed until the weight of the bigger level is reduced below its threshold. In DOWNFORCE, we ease this strict constraint on the weight of sorted runs within a level while still preserving the LSM-tree’s structural integrity. By prioritizing compaction execution over structural constraints under write-intensive workloads, DOWNFORCE enables compactions to proceed without delay.

Principle #2: Multiple Level Parallelism. WPC aims to increase parallelism within a single level, so maximizing thread utilization at that level is desirable. However, since the LSM-tree is composed of multiple levels, focusing all resources on a single compaction-heavy level presents clear limitations. Moreover, in the LSM-tree composed of many small, fixed-size SSTables, smaller levels—which experience more frequent compactions—may not offer sufficient task granularity for effective parallelism. These conflicting characteristics highlight the need to balance parallelism across levels by considering the overall LSM-tree structure. DOWNFORCE addresses this by maximizing WPC across multiple levels, fully utilizing available parallelism throughout the LSM-tree.

Principle #3: LSM-tree Consistency. On the other hand, naively relaxing the W_i restriction to improve CPC can temporarily allow data redundancy or amplification that is otherwise disallowed, potentially increasing read costs or violating data consistency. To address this, DOWNFORCE introduces opportunistic intra-level compaction and a consistency-guaranteed commit mechanism. These ensure that the LSM-tree consistency is strictly maintained while minimizing the side effects of temporary amplification.

5.2 Design

In this section, we illustrate how compaction behavior changes when our design principles are applied through DOWNFORCE, using a representative example scenario.

DOWNFORCE Compaction for Non-Blocking CPC. In the existing approach, if an ongoing L_{i+1} compaction has not yet completed, any new L_i compaction is deferred to prevent the W_{i+2} limit from being violated.

Figure 5(a) illustrates a scenario in which DOWNFORCE is triggered. ① Suppose job1, an L_{i+1} compaction, is in progress and takes a long time to complete. ② Meanwhile, continuous writes cause an L_i compaction (job2) to be triggered repeatedly. ③ As job2 proceeds, the weight W_{i+1} increases. ④ Once W_{i+1} reaches the level limit, the condition for triggering another L_{i+1} compaction is met. ⑤ Instead of waiting for job1 to complete and stabilize the state of L_{i+2} , DOWNFORCE immediately triggers another compaction, job3, referred to as DF-compaction. The purpose of DF-compaction is to proactively free up capacity of the triggered level (e.g., L_{i+1} for job3 case). ⑥ When job3 runs, it may cause the weight of L_{i+2} to exceed its limit. This excess is temporarily allowed and later resolved through opportunistic intra-level compaction, as discussed in Figure 5(c).

Balanced WPC with DOWNFORCE Compaction. The additional data, denoted as $Extra W_{i+2}$, can be processed using multi-threading, thereby enhancing WPC. Figure 5(b) shows two possible outcomes after job3 is deployed. ⑦ In the upper case, the initial job1—which triggered DF-compaction due to its long execution time—finishes before job3. In this case, the system waits for the ongoing DF-compaction (job3) to finish before proceeding. ⑧ Once job3 completes, L_{i+2} contains an excess weight. ⑨ At this point, the system checks whether the sum of the updated W_{i+2} (reflecting job1's output) and the $Extra W_{i+2}$ from job3 exceeds the compaction threshold. Note that whether the threshold is exceeded depends on the compaction policy in use, and W_{i+2} may have already been reduced by another compaction occurring concurrently while job1 and job3 were in progress. Thus, exceeding the threshold is not guaranteed.

⑩ In the lower case of Figure 5(b), job3—which was deployed later—completes before job1. If job1 remains in progress for an extended period, additional DF-compactions such as ⑪ job4 and ⑫ job5 may be triggered consecutively, cascading from smaller levels. ⑬ In this case, since job1 has not yet completed, consecutive DF-compactions increase the $Extra W_{i+2}$. If too much extra weight accumulates, the level may become oversized, potentially even surpassing the compaction threshold. Consequently, the cost of handling this excess weight increases accordingly.

Next, Figure 5(c) illustrates how the $Extra W$ generated by DF-compaction is handled through opportunistic intra-level compaction in the scenarios that immediately follow each case shown in Figure 5(b). Note that the distinction between the upper case and the lower case at this point is whether job1 has been completed. In the upper part of the Figure 5(c), two cases—A and B—are distinguished based on whether the sum of W_{i+2} and $Extra W_{i+2}$ exceeds the compaction threshold of L_{i+2} . Case A: If the sum of weight does not exceed the compaction threshold, ⑭ an intra-level compaction is performed throughout L_{i+2} , including both W_{i+2} and $Extra W_{i+2}$. The final result is shown as ⑮, representing the reduced W_{i+2} .

Case B: If the sum of weight exceeded threshold, ⑯ additional compaction is triggered to the next bigger level, instead of intra-level compaction. In the lower part of the Figure 5(c), Job1 is still incomplete; in this context, the continually growing $Extra W_{i+2}$ must be reduced. ⑰ Thus, intra-level compactations targeting only the $Extra W_{i+2}$ are repeatedly executed until job1 completes. This process is called intra-DF compaction because it operates exclusively on the $Extra W_{i+2}$ generated by DF-compaction. ⑱ Once job1 completes, the total weight is recalculated, and the system either ⑲ merges the remaining weight within the current level (same as ⑲a) or ⑲b triggers another compaction to the next bigger level.

Algorithm 1 presents the compaction procedure of DOWNFORCE as described thus far. In summary, DOWNFORCE enables compactations in L_{i+1} to proceed without waiting, as soon as they are triggered. As a result, compaction delays are no longer vertically propagated, allowing compactations in L_i to proceed without interruption. Moreover, any temporary increase in weight caused by DF-compaction can be handled independently at each level through the opportunistic intra-level compaction scheme. This leads to more balanced parallel processing across multiple levels, rather than overloading a single level.

Maintaining LSM-tree Consistency. However, when using DF-compaction, issues may arise where a later compaction job completes before an earlier one, even if their key ranges overlap. Additionally, partially merged data may be compacted before the level has stabilized. Such cases can violate the consistency of the LSM-tree, which operates in an append-only manner. To address this, we adopt a consistency-guaranteed commit mechanism: even if a compaction job within a level finishes early, it is not immediately

Algorithm 1: DOWNFORCE Compaction

Input: Index i with ongoing compaction $L_{i+1} \rightarrow L_{i+2}$
Data: W_i, C_{L_i}, C_T for all levels; helper routines
Result: Triggers DF-compactions, preserves consistency, opportunistic merges

```

1 compact ← GetOngoingCompaction( $L_{i+1}, L_{i+2}$ );
2 if compact = NULL then
3   return
4 Extra $W_{i+2}$  ← 0;
5 while ( $W_{i+1} \geq C_T$ ) or ( $W_{i+1} \geq C_{L_{i+1}}$ ) do
6   df ← StartDFCompaction( $L_{i+1}, L_{i+2}$ );
7 while IsActive(compact) or HasActiveDF() do
8   foreach df in FinishedDF() do
9     Extra $W_{i+2}$  ← Extra $W_{i+2}$  + df.outputWeight;
10    if ( $W_{i+2} + ExtraW_{i+2} > C_T$ ) or ( $W_{i+2} + ExtraW_{i+2} > C_{L_{i+2}}$ )
11      then
12        IntraDFCompaction( $L_{i+2}, ExtraW_{i+2}$ );
13        WaitIntraComp(df);
14    WaitForAnyDFCompletion();
15 Commit(compact);
16 foreach df in issue order do
17   Commit(df);
18  $W_{i+2}$  ←  $W_{i+2} + ExtraW_{i+2}$ ;
19 if ( $W_{i+2} > C_T$ ) or ( $W_{i+2} > C_{L_{i+2}}$ ) then
20   StartCompaction( $L_{i+2}, L_{i+3}$ );
21 else
22   IntraLevelCompaction( $L_{i+2}$ );
```

committed. Instead, compaction results are applied to the LSM-tree in the order the jobs were issued. Figure 5(d) illustrates this mechanism in DOWNFORCE. When jobs 1, 3, and 4 are issued in sequence but complete in the order of 3, 4, and 1, the system does not commit them as they finish. Instead, it enforces the original issue order—job1, job3, job4—for consistency.

5.3 Implementation

We implemented DOWNFORCE in RocksDB v9.10.0, a widely used LSM-KVS. Within RocksDB, the `compaction_picker.cc` module contains abstract classes responsible for triggering compactions and managing compaction pointers, which determine the input level and candidate victim SSTables. To maintain the weight of sorted runs, RocksDB prevents compaction from being scheduled on a level already undergoing compaction by returning a nullptr as the compaction pointer. We modified `compaction_picker.cc` to enable compaction pipelining, laying the foundation for DOWNFORCE.

The design of DOWNFORCE is agnostic to compaction strategy and can be applied to both leveled and tiered compaction. To comprehensively evaluate its effectiveness across different strategies, we implemented DOWNFORCE for both leveled and tiered compaction strategies. Specifically, we extended our modifications to `compaction_picker_level.cc` and `compaction_picker_universal.cc` to implement Algorithm 1.

The source code for DOWNFORCE is publicly available at <https://github.com/DISCOS-LAB/DownForce.git>

6 Evaluation

6.1 Experimental Setup

Platform. Our experiments were conducted on a server equipped with a Ryzen 9 7950X (16 cores, 32 threads, SMT enabled) @ 4.5 GHz CPU, 32 GB DDR5 DRAM, and a 1 TB Samsung 990 PRO NVMe SSD. The system ran Ubuntu 22.04.5 with the ext4 file system.

Compaction Strategies. We comprehensively evaluated DOWNFORCE across the following compaction strategies. All strategies were implemented and tested on RocksDB [9] v9.10.0:

- **Leveled:** The classic leveled compaction, where each level maintains only a single sorted run.
- **Tiered:** The classic tiered compaction, in which each level allows up to 4 sorted runs.
- **R-Leveled:** The default hybrid compaction in RocksDB, where the first level is tiered and the remaining levels are leveled.
- **R-Tiered:** RocksDB’s implementation of *universal compaction* [13], a tiered strategy with L0-to-L0 intra-compaction enabled to reduce space amplification under write-intensive workloads.
- **DF-Leveled:** DOWNFORCE is applied to R-Leveled compaction.
- **DF-Tiered:** DOWNFORCE is applied to Tiered compaction, rather than R-Tiered, to enable a fair evaluation of the strategy without the influence universal compaction optimizations.

Table 2: YCSB workload characteristics.

Type	Description	Distribution
A	50% Updates, 50% Reads	Zipfian
B	95% Reads, 5% Updates	Zipfian
C	100% Reads,	Zipfian
D	95% Reads, 5% Inserts	Latest
E	95% Seeks, 5% Inserts; Scan length:100	Uniform
F	50% Read-Modify-Write, 50% Reads	Zipfian

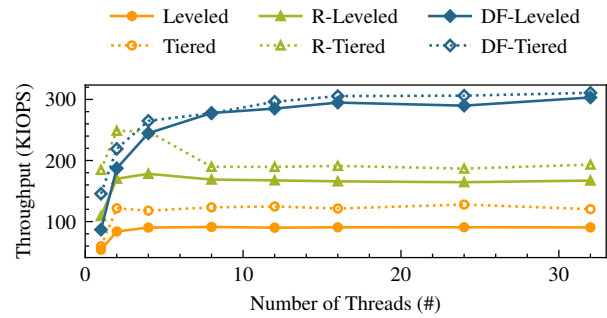


Figure 6: Throughput comparison of compaction strategies with increasing number of threads.

In all configurations, the level size ratio is set to $T = 5$, and for tiered compaction, the run threshold is $C_T = 4$.

RocksDB Parameter Configurations. To ensure a fair comparison focused solely on compaction strategies, we disabled the Write-Ahead Log, block cache, and compression. We also enabled direct I/O for both compaction and I/O requests. All experiments used two 64 MB memtables and a single flush thread.

Workloads. We first evaluated and analyzed the effectiveness of DOWNFORCE in compaction multi-threading using `db_bench` [8], and then conducted performance comparisons across compaction strategies using the YCSB benchmark [2]. We tested the following workloads with key-value sizes of 16 B and 1 KB.

- **FUR:** The FillUniqueRandom workload from `db_bench`, consisting of 100% writes totaling 50 GB.
- **RRWR:** The ReadRandomWriteRandom workload from `db_bench`, using 50 million entries with varying read/write ratios.
- **YCSB:** The YCSB benchmark executed on 10 million entries. The request distribution follows the YCSB default configuration. Detailed workload characteristics are provided in Table 2.

6.2 Evaluation Results

Performance Scalability. In this experiment, we evaluated the performance of each compaction strategy by varying the number of compaction threads from 1 to 32, using the FillUniqueRandom workload. Figure 6 shows how I/O throughput changes as the number of compaction threads increases. We observe that our design brings performance benefits to both compaction strategies. Compared to the default RocksDB strategies, DOWNFORCE achieves up to 1.81× improvement in leveled compaction and 1.63× in tiered compaction.

Other strategies reached peak performance with 8 or fewer threads, showing no further gains—or even degradation—when more threads were added. In particular, R-Leveled peaked at 4 threads and exhibited a sharp decline beyond that, due to its hybrid design allowing a large number of overlapping sorted runs in the first level. In contrast, DOWNFORCE continued to scale consistently with increasing thread counts, showing noticeable improvements up to 16 threads—unlike other strategies, which plateaued much earlier.

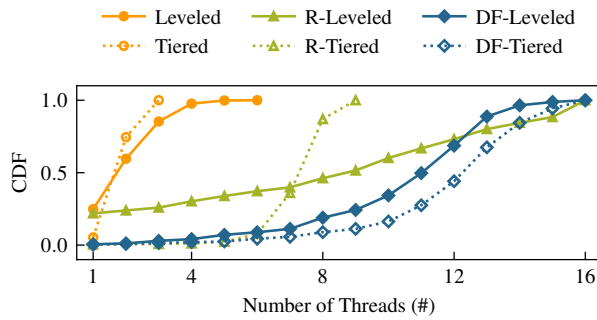


Figure 7: Cumulative distribution function of concurrently active threads for compaction strategies with a 16-thread pool.

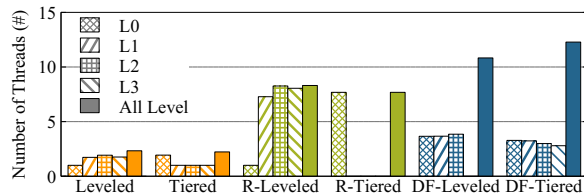
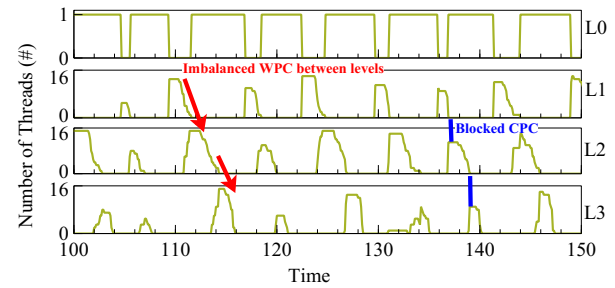


Figure 8: Average number of concurrently running compaction threads per level at 0.1 second intervals. ‘All level’ indicates the average number of total compaction threads running concurrently across all levels.

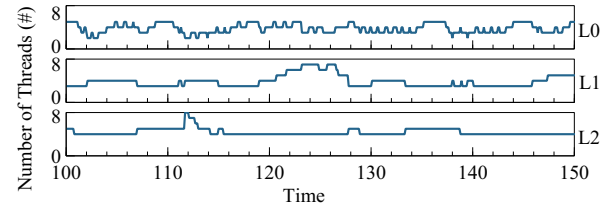
Parallelism of Multi-threading. Next, we evaluated how effectively our proposed design utilizes multi-threaded parallelism in practice. Figure 7 presents a cumulative distribution function (CDF) of the number of compaction threads actively running at 0.1 second intervals, with 16 threads allocated and using the FillUniqueRandom workload. DOWNFORCE executed more than 12 compactions concurrently for over 50% of the time. In contrast, all other strategies except R-Tiered never exceeded 9 concurrent threads. Although R-Tiered utilized up to 16 threads due to L0-to-L0 intra-compaction, its overall utilization was lower compared to DOWNFORCE.

Figure 8 shows the average per-level count of concurrent compaction threads measured under the same workload and settings as in Figure 7. DF-Leveled utilizes up to 1.3 \times more concurrent compaction threads than R-Leveled across all levels. Though R-Leveled utilizes more threads at certain levels, All Level value is lower than DF-Leveled because threads are mostly confined to a single level because CPC is blocked, as discussed in Section 3. No L3 compaction is observed in DF-Leveled because, although the workload size is identical, opportunistic intra-compaction prevents SSTables from propagating past L3. Like DF-Leveled, DF-Tiered shows a similarly high degree of parallelism. In contrast, most data in R-Tiered is handled by L0-to-L0 intra-compaction because RocksDB’s default universal compaction policy prioritizes that operation. The unoptimized classic strategies strictly enforce the sorted-run weight limits, a constraint that prevents wide parallelism in compaction and leaves them using far fewer threads overall.

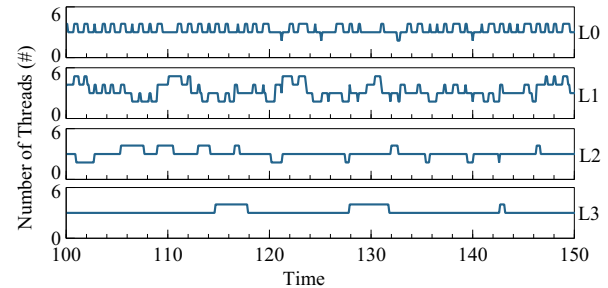
Figure 9 shows the per-level count of active compaction threads, sampled every 0.1 s, for R-Leveled, DF-Leveled, and DF-Tiered.



(a) R-Leveled



(b) DF-Leveled



(c) DF-Tiered

Figure 9: Time-series compaction thread counts per level for R-Leveled, DF-Leveled and DF-Tiered with 16 threads. Results shown for the 100–150 second interval of the FUR workload.

The x-axis denotes time. From Figure 9(a), two key observations can be made. First, as indicated by the red arrows, most available threads are concentrated on a single level during a compaction event. Once the compaction completes, the threads shift to another level. This indicates that, despite having enough threads, the system fails to realize effective WPC across multiple levels simultaneously—clearly demonstrating the Imbalanced WPC Between Levels issue discussed in Section 3. Second, as shown by the blue lines in the figure, blocking between adjacent levels hinders CPC. Except for L_0 , compactions rarely occur concurrently across different levels, highlighting the limitations of CPC under the default R-Leveled strategy.

In contrast, Figure 9(b) and 9(c) show that with the DOWNFORCE strategy, compaction threads are evenly distributed across all levels. This indicates that WPC is effectively utilized across multiple levels. Moreover, the continuous compactions occurring simultaneously at different levels demonstrate that CPC is also fully operational under DOWNFORCE.

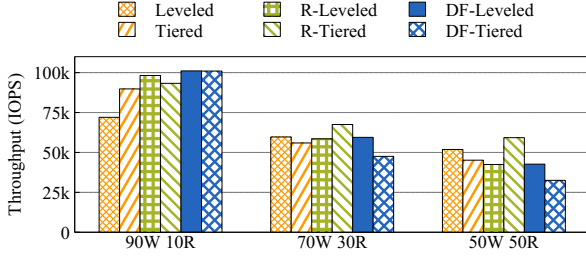


Figure 10: RRWR workload evaluation for each compaction strategy with a 16-thread pool. The x -axis represents the write-to-read ratio.

RRWR Workload. Figure 10 presents the performance trends of each compaction strategy when using 16 compaction threads under varying read/write ratios. The workload used is ReadRandomWriteRandom, and we describe each configuration by its read ratio (e.g., 90W 10R indicates 90% writes and 10% reads). When the read ratio is 10%, both DF-Leveled and DF-Tiered achieved the highest performance. This aligns with previous results, where DOWNFORCE was shown to effectively maximize compaction parallelism and improve write throughput. However, as the read ratio increases, the relative performance of DOWNFORCE-based strategies declines. At 30% reads, DF-Leveled slightly outperformed R-Leveled, whereas DF-Tiered showed lower performance than R-Tiered. At a 50% read ratio, DF-Tiered showed the lowest performance, while DF-Leveled performed slightly worse than Tiered but still better than R-Leveled. These results indicate that applying DOWNFORCE to leveled compaction improves write throughput while maintaining comparable read performance. In contrast, applying it to tiered compaction has a more negative impact on reads. This is likely due to the larger impact of worst-case $Extra W_i$ on read paths when DOWNFORCE is used with tiered compaction. In Tiered settings, more aggressive data overlap and relaxed constraints can amplify read overhead when excess weight accumulates. Leveled compaction, on the other hand, limits the $Extra W_i$ more strictly and triggers compaction based on the total size of each level. As a result, weight amplification is lower, and the side effects are effectively mitigated through our WPC enhancements.

YCSB Benchmarks. We measured the throughput of each compaction strategy using the YCSB workload with 16 compaction threads. Figure 11 shows the throughput for each workload. First, in the Load phase, the Leveled strategy exhibited noticeably lower throughput, while the other strategies showed similar performance. Across workloads A–F, we compare R-Leveled, DF-Leveled, and DF-Tiered. Except for workloads E and F, both DF-Leveled and DF-Tiered consistently outperformed R-Leveled. This suggests that the design of DOWNFORCE contributes not only to improved write throughput but also to enhanced read performance. Notably, DF-Leveled achieved higher throughput than R-Leveled across most workloads. In particular, under workloads B, C, D, and E, DF-Leveled outperformed R-Leveled by 1.5 \times , 2.9 \times , 1.7 \times , and 1.5 \times , respectively.

Table 3: Write amplification by compaction strategy.

	Leveled	Tiered	R-Leveled	R-Tiered	DF-Leveled	DF-Tiered
WA	11.1	9.3	7.1	6.9	4.8	4.5

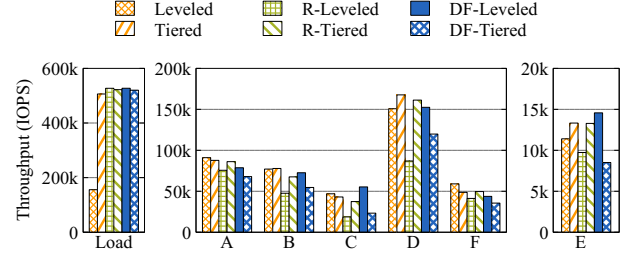


Figure 11: YCSB benchmark results for each compaction strategy with a 16-thread pool.

Next, we examine the remaining strategies. Overall, classic Leveled and Tiered compaction strategies achieved higher throughput. This is because they maintain fewer sorted runs, which leads to better read performance. Among the hybrid and DOWNFORCE-based strategies—R-Leveled, R-Tiered, DF-Leveled, and DF-Tiered—R-Tiered showed the highest throughput. This is primarily due to RocksDB’s universal compaction, which enables L0-to-L0 intra compaction and reduces compaction overhead under certain workloads.

Write Amplification. Table 3 presents the write amplification observed for each compaction strategy under the FillUniqueRandom workload with 16 compaction threads. Our strategy achieved the lowest write amplification. This result stems from the effect of $Extra W_i$ introduced by our CPC and WPC acceleration techniques. Because $Extra W_i$ allows data to participate in fewer compactions before reaching lower levels, write amplification is reduced. This behavior is particularly relevant to tiered compaction, where write amplification is influenced by the compaction threshold C_T . As shown in Equation 2, write amplification decreases as C_T increases.

7 Related Work

LSM-tree Compaction Strategies. To address compaction bottlenecks, various strategies have been proposed. Monkey [3] optimizes memory allocation for Bloom filters and buffers at each level, minimizing read overhead. Dostoevsky [4] adaptively removes redundant merges to balance write and space efficiency, while Spooky [5] minimizes redundant merging by refining compaction granularity. Pipelined Compaction [31] splits I/O and CPU stages for overlap. bLSM [22] introduces a spring-and-gear scheduler to smooth merges. LSbM-tree [26] maintains an on-disk buffer and introduces buffered merge to reduce negative impact during compaction. However, these works mainly focus on modifying merge strategies or data structures, without directly addressing the limitations of parallel compaction.

Software Optimization. Software-level optimizations for compaction include various techniques such as key-value separation [19], restructuring internal LSM-tree data layouts [17, 21], and sophisticated compaction scheduling [1, 28]. Key-value separation approaches significantly reduce data movement during compaction by storing large values separately from keys. Restructuring LSM-tree data layouts reduces compaction overhead by efficiently reorganizing storage layers, while optimized scheduling algorithms minimize interference between compaction tasks and user requests. However, these methods do not fundamentally address the inherent parallelism constraints of multi-threaded compaction.

Hardware Acceleration / Off-loading. FPGA-based approaches [25, 30] accelerate compaction by executing it on reconfigurable logic, reducing CPU overhead and I/O contention. GPU off-loaders such as gLSM [24] and GPU-accelerated compaction [32] pipeline decoding, radix sort, and encoding on the GPU while the CPU performs only minor merges. Near-data processing engines [23] move compaction logic into the SSD controller so merges occur inside the device. DComp [7] off-load some compactions into DPU and gain hardware supported acceleration on data (de)compression. Persistent-memory engines [6, 27] stage early merges in byte addressable media. Cloud variants run compaction on FaaS nodes [18] or within an external micro-service [29], while edge deployments often offload compaction to remote servers [15, 16].

Unlike these approaches, our work enhances multi-threaded compaction efficiency solely through software optimization, without structural or hardware modifications.

8 Conclusion

We have shown that modern LSM-tree systems suffer from limited scalability in multi-threaded compaction due to restricted cross-level parallelism and thread imbalance. To address this, we proposed DOWNFORCE, a compaction strategy that enables pipelined compaction across levels and activates true multi-threaded execution. DOWNFORCE is implemented in RocksDB and improves both thread utilization and write throughput under write-intensive workloads. Our results demonstrate that DOWNFORCE is a practical and effective solution for enhancing compaction parallelism in the LSM-tree.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (RS-2024-00453929) (RS-2024-00416666).

References

- [1] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhramoorthi, and Diego Didona. 2019. SILK : Preventing latency spikes in Log-Structured merge Key-Value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 753–766.
- [2] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.
- [3] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 79–94.
- [4] Niv Dayan and Stratos Idreos. 2018. 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*. 505–520.
- [5] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: granulating LSM-tree compactions correctly. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3071–3084.
- [6] Chen Ding, Ting Yao, Hong Jiang, Qiu Cui, Liu Tang, Yiwen Zhang, Jiguang Wan, and Zhihu Tan. 2022. TriangleKV: Reducing write stalls and write amplification in LSM-tree based KV stores with triangle container in NVM. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 4339–4352.
- [7] Chen Ding, Jian Zhou, Jiguang Wan, Yiqin Xiong, Sicen Li, Shuning Chen, Hanyang Liu, Liu Tang, Ling Zhan, Kai Lu, et al. 2023. Dcomp: Efficient off-load of lsm-tree compaction with data processing units. In *Proceedings of the 52nd International Conference on Parallel Processing*. 233–243.
- [8] Facebook. 2013. db_bench. <https://github.com/facebook/rocksdb/wiki/Benchmarking-tools>. Accessed: 2025-05-01.
- [9] Facebook. 2013. RocksDB: A Persistent Key-Value Store for Fast Storage Environments. <https://github.com/facebook/rocksdb>. Accessed: 2025-05-01.
- [10] Facebook. 2021. RocksDB: Write Stalls. <https://github.com/facebook/rocksdb/wiki/Write-Stalls>. Accessed: 2025-05-01.
- [11] Facebook. 2023. Compaction · RocksDB. <https://github.com/facebook/rocksdb/wiki/Compaction>. Accessed: 2025-05-01.
- [12] Facebook. 2023. Multi-threaded Compaction · RocksDB. <https://github.com/facebook/rocksdb/wiki/RocksDB-Overview#multi-threaded-compaction>. Accessed: 2025-05-01.
- [13] Facebook. 2023. Universal Compaction · RocksDB. <https://github.com/facebook/rocksdb/wiki/universal-compaction>. Accessed: 2025-05-01.
- [14] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for nonvolatile memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 993–1005.
- [15] Jeeseob Kim, Hongsu Byun, Seungjae Lee, Myoungjoon Kim, Youngjae Kim, Zaipeng Xie, and Sungyong Park. 2025. ECO-KVS: Energy-Aware Compaction Offloading Mechanism for LSM-Tree Based Key-Value Stores in Edge Federation. In *2025 IEEE 25th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 570–579.
- [16] Jeeseob Kim, Honghyeon Yoo, Seungjae Lee, Hongsu Byun, and Sungyong Park. 2024. Coordinating compaction between lsm-tree based key-value stores for edge federation. In *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*. IEEE, 419–429.
- [17] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 447–461.
- [18] Jianchuan Li, Peiquan Jin, Yuanjin Lin, Ming Zhao, Yi Wang, and Kuankuan Guo. 2021. Elastic and Stable Compaction for LSM-tree: A FaaS-Based Approach on TerarkDB. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management (Virtual Event, Queensland, Australia) (CIKM '21)*. Association for Computing Machinery, New York, NY, USA, 3906–3915. doi:10.1145/3459637.3481913
- [19] Lanyue Lu, Thanumalayan Sankaranarayanan Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions On Storage (TOS)* 13, 1 (2017), 1–28.
- [20] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.
- [21] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 497–514.
- [22] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 217–228.
- [23] Hui Sun, Bendong Lou, Chao Zhao, Deyan Kong, Chaowei Zhang, Jianzhong Huang, Yinliang Yue, and Xiao Qin. 2023. An Asynchronous Compaction Acceleration Scheme for Near-Data Processing-enabled LSM-Tree-based KV Stores. *ACM Transactions on Embedded Computing Systems* (2023).
- [24] Hui Sun, Jinfeng Xu, Xiangxiang Jiang, Guanzhong Chen, Yinliang Yue, and Xiao Qin. 2024. gLSM: Using GPGPU to Accelerate Compactions in LSM-tree-based Key-value Stores. *ACM Transactions on Storage* 20, 1 (2024), 1–41.
- [25] Xuan Sun, Jinghuan Yu, Zimeng Zhou, and Chun Jason Xue. 2020. FPGA-based Compaction Engine for Accelerating LSM-tree Key-Value Stores. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1261–1272. doi:10.1109/ICDE48307.2020.00113
- [26] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Yanfeng Zhang, Siyuan Ma, and Xiaodong Zhang. 2018. A Low-cost Disk Solution Enabling LSM-tree to Achieve High Performance for Mixed Read/Write Workloads. 14, 2, Article 15 (April 2018), 26 pages. doi:10.1145/3162615
- [27] 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 *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 17–31.
- [28] Jinghuan Yu, Sam H Noh, Young-ri Choi, and Chun Jason Xue. 2023. ADOC : Automatically Harmonizing Dataflow Between Components in Log-Structured Key-Value Stores for Improved Performance. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 65–80.
- [29] Qiaolin Yu, Chang Guo, Jay Zhuang, Viraj Thakkar, Jianguo Wang, and Zhichao Cao. 2024. CaaS-LSM: Compaction-as-a-Service for LSM-based Key-Value Stores in Storage Disaggregated Infrastructure. *Proc. ACM Manag. Data* 2, 3, Article 124 (May 2024), 28 pages. doi:10.1145/3654927
- [30] 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 *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 225–237.
- [31] Zigang Zhang, Yinliang Yue, Bingsheng He, Jin Xiong, Mingyu Chen, Lixin Zhang, and Ninghui Sun. 2014. Pipelined Compaction for the LSM-Tree. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 777–786. doi:10.1109/IPDPS.2014.85
- [32] Hao Zhou, Yuanhui Chen, Lixiao Cui, Gang Wang, and Xiaoguang Liu. 2024. A GPU-accelerated Compaction Strategy for LSM-based Key-Value Store System. (2024), 1–11.