vLSM: Low tail latency and I/O amplification in LSM-based KV stores

Giorgos Xanthakis
gxanth@ics.forth.gr
University of Crete & ICS-FORTH
Heraklion, Greece

Antonios Katsarakis[1]
antonios.katsarakis@huawei.com
Huawei
Edinburgh, United Kingdom

Giorgos Saloustros
gesalous@ics.forth.gr
University of Crete & ICS-FORTH
Heraklion, Greece

Angelos Bilas
bilas@ics.forth.gr
University of Crete & ICS-FORTH
Heraklion, Greece

Abstract

LSM-based key-value (KV) stores are an important component in modern data infrastructures. However, they suffer from high tail latency, in the order of several seconds, making them less attractive for user-facing applications.

In this paper, we introduce the notion of compaction chains and we analyse how they affect tail latency. Then, we show that modern designs reduce tail latency, by trading I/O amplification or require large amounts of memory.

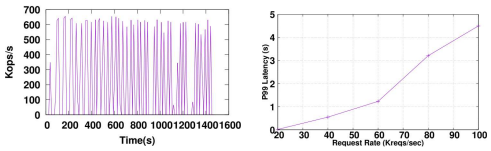
Based on our analysis, we present *vLSM*, a new KV store design that improves tail latency significantly without compromising on memory or I/O amplification. *vLSM* reduces (a) compaction chain width by using small SSTs and eliminating the tiering compaction required in L_0 by modern systems and (b) compaction chain length by using a larger than typical growth factor between L_1 and L_2 and introducing overlap-aware SSTs in L_1.

We implement *vLSM* in RocksDB and evaluate it using db_bench and YCSB. Our evaluation highlights the underlying trade-off among memory requirements, I/O amplification, and tail latency, as well as the advantage of *vLSM* over current approaches. *vLSM* improves P99 tail latency by up to 4.8× for writes and by up to 12.5× for reads, reduces cumulative write stalls by up to 60% while also slightly improves I/O amplification at the same memory budget.

1 Introduction

Log-structured merge-tree (LSM) key-value (KV) stores are a cornerstone in the evolution of modern storage systems [4, 10, 23]. They are the backbone of popular user-facing applications and services, including social media [3, 8], financial systems [28], and AI workflows [27]. Such applications demand cost-effective access to large volumes of data via highly concurrent and latency-sensitive requests.

To deliver high throughput cost-effectively, state-of-the-art LSM KV stores optimize for low I/O amplification and low memory usage. To achieve this, modern KV stores introduce a tiering step in the first level on the device (L_0) [17]. This tiering step allows them to use a small amount of memory for the in-memory component, while L_0 can still be large. This results in an LSM with fewer levels and, therefore, reduced I/O amplification. However, the tiering compaction step leads to prolonged write-stalls, which inflate tail latency [20, 30].



(a) Write stalls (b) P99 latency

Figure 1. RocksDB throughput and P99 latency in YCSB Load A, using 350M KV pairs with a KV size of 240 B.

Figure 1(a) shows RockDB's throughput over time for YCSB (Load A – more details in § 5). The runtime is significantly impacted by write stalls, which occur due to compaction chains that must occur to free space before RocksDB can ingest new requests. These write stalls account for approximately 40% of the total runtime and drastically affect tail latency and the user experience. Figure 1(b) shows the 99th-percentile (P99) latency as the load increases in the same setup. RocksDB's P99 latency is in the order of seconds, even with a load less than 60% of its maximum throughput.

Related works have tried to mitigate the high tail latency caused by compactions using two different approaches: *memory-based* or *scheduling-based* solutions. Memory-based solutions, keep significantly more data in memory, i.e. mandate a high memory budget, to reduce the number of slower disk-resident compactions in the request critical path. These solutions are costly and mainly rely on new memory technologies, such as byte-addressable non-volatile memory [20, 30] that may not be broadly available [29].

In contrast, scheduling-based solutions aim to schedule compactions in the background to reduce the amount of work that needs to occur in the critical path, and therefore, reduce tail latency [1, 26, 31]. While these approaches can significantly lower write stalls and tail latency, they assume a light load or over-provisioned resources (e.g., underutilized CPU or device resources) to perform compactions in parallel

[1]This work was done while the author was at the University of Edinburgh.

with the running workload. More importantly, scheduling-based approaches increase I/O amplification by temporarily increasing the size of levels, resulting in more background work per compaction. For example, our evaluation shows that a state-of-the-art scheduling approach ADOC [31] increases I/O amplification from 26× to 46×.

We identify two main factors in traditional LSM compactions that affect tail latency. Namely, the maximum *width* and the maximum *length* of compaction chains that can occur on the critical path of requests. Simply put, the width is determined by the amount of compacted bytes per level, while the length refers to the total number of levels that must be compacted to free memory. For state-of-the-art LSM KV stores, the tiering compaction step in L_0 governs the width, which typically is 2 GBs per level in RocksDB. Assuming we have 5 to 7 levels in a typical LSM KV store, the combined width and length of compaction chains reaches tenths of GBs. Intuitively, an LSM design should reduce those without inflating I/O amplification or memory usage.

Based on this insight, we introduce *vLSM*, a novel LSM design that ensures low tail latency and low I/O amplification at a small memory budget. To achieve this, *vLSM* carefully reduces both the width and height of compaction chains in LSMs. It reduces the width of compaction chains by ❶ employing smaller SSTs and ❷ removing tiering compactions. To keep compaction chains short without compromising on memory or I/O amplification, *vLSM* meticulously handles the first device-only levels (L_1 and L_2). It applies ❸ a larger growth factor Φ from L_1 to L_2 and ❹ introduces overlap-aware variable-size SSTs (vSSTs) in L_1. The size of vSSTs in L_1, is determined by a novel look-ahead compaction policy that minimizes their overlap with SSTs in L_2.

We evaluate *vLSM* against RocksDB and ADOC over standard workloads and benchmarks (YCSB and db_bench) as well as sensitivity studies. Our experiments demonstrate substantial improvements in tail latency without compromising on I/O amplification or memory size. Compared to RocksDB (ADOC), *vLSM* reduces P99 latency up to 5.2× (1.99×) for YCSB Load A that performs inserts. For mixed read-write workloads, such as YCSB Run A, *vLSM* improves the P99 write latency by 4.8× (2.13×) and P99 read latency by 12.5×. At the same time, *vLSM* requires a similar amount of memory as existing designs that optimize for I/O amplification and memory use. Overall, our contributions are:

- **We identify that state-of-the-art memory-frugal LSM stores sacrifice tail latency or I/O amplification**. We pinpoint the root cause of this conundrum to the width and length of compaction chains that occur on the critical path of a request.
- **We introduce *vLSM*, a novel KV store design** that achieves low tail latency without inflating I/O amplification or memory usage. *vLSM* achieves this by reducing the width and length of compaction chains through the combination of four key ideas: ❶ smaller SSTs, ❷ eschewing tiering compactions, ❸ larger growth factor from L_1 to L_2, and ❹ overlap-aware vSSTs in L_1.
- **We extensively evaluate *vLSM*** using standard benchmarks and sensitivity studies. Our results show that *vLSM* shrinks compaction chains by up to 10×, which translates into 4.8× (12.5×) lower latency for writes (reads) over RocksDB and up to 1.7× lower I/O amplification than the state-of-the-art scheduling solution of ADOC.

2 Motivation

2.1 Modern KV Stores

A KV store typically divides its data in *regions*, often in the order of hundreds [11]. Each region is a subset of a KV range with an independent LSM index from other regions. It has a predefined number of maximum levels n, which is derived from its maximum capacity, the size of L_0, and the growth factor f across levels. In each region, when L_0 is full, the LSM KV store selects an SST from L_0 and compacts it with the overlapping SSTs from L_1 [19].

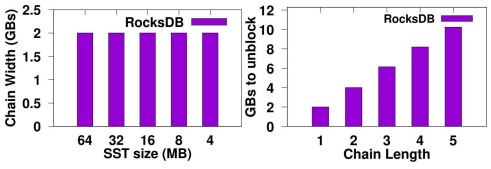
2.2 Incremental compactions

Modern KV stores use incremental compaction [10, 19]. They organize levels in non-overlapping sub-units that contain sorted KV pairs named Sorted String Tables (SSTs). Assuming a constant growth factor f, KV stores organize each level with fixed-size SSTs (e.g., 64 MB) and increase their number from level to level by f times. It is important to note that each SST is a self-contained unit, stored in a single file or extent on the device, containing all its data (KV pairs) and metadata (index and bloom filters).

SSTs enable incremental compaction. They allow KV stores to merge-sort part of a level (one or a few SSTs) into the next level. Simply, when a level is full, instead of compacting the whole level, the KV store can compact a single SST to the next level. Therefore, the LSM KV store must reorganize n levels touching only $n \times f$ SSTs to free up space for a subsequent insert operation, dramatically reducing the amount of work and decreasing tail latency compared to full compaction [24]. Under incremental compaction, the width of the compaction chains is affected by the growth factor f and the size of SSTs. The growth factor indicates how many SSTs of the target level are involved, on average, in an incremental compaction step. The size of SSTs affects the amount of work required to merge each overlapping SST.

2.3 Compaction chains

In this work, we identify that the root cause of tail latency in modern LSM-based KV stores are *compaction chains* that form during operation. Compaction chains are sequences of dependent compactions from level to level. These compactions need to be processed in order to maintain the correctness properties of the LSM and keep the latest data higher



(a) Chain width (b) Chain length

Figure 2. RocksDB chain width (left) and chain length (right) for different SST sizes.

up in the tree. Hence, when levels are full, we first need to free space in L_{n-1}, then L_{n-2}, and so on, until the in-memory component.

Compaction chains have two aspects that affect tail latency. First, the *length* of the chain, which is defined by the number of LSM levels in the chain. The number of LSM levels is defined by the size of the in-memory component and the growth factor f. Second, the amount of work for each compaction in a chain, which we call the *width* of the chain. Figure 3(a), illustrates the width and the length of compaction chains over an LSM with incremental compactions (LSMi).

A chain of dependent compactions from L_0 to L_n must take place to free space in each next level whenever they are full. The aggregate capacity of L_0 to L_{n-1} level is about 10% of the total KV store space for typical growth factors around 8-10 [2, 9]. As a result, long and fat compaction chains are observed as early as the dataset size reaches or exceeds about 10% of the capacity of each region.

Figure 2(a) shows how much data compaction moves on average on each level. Figure 2(b) shows the amount of data that needs to be compacted to free space for a memtable in L_0 with a different number of levels in RocksDB. The width of the compaction chain is 2 GBs per level and increases by L_0 size for each level. So, for a typical deployment with five levels, an LSM KV store needs to reorganize up to 10 GBs of data to free space for 64 MBs in memory, which is about 20 GBs of read/write traffic.

3 Analysis of compaction chains: Tail latency *vs.* I/O amplification *vs.* memory usage

We observe that state-of-the-art systems follow three strategies to reduce the cost of compaction chains and, thus, tail latency.

1. Reduce the length of chains by using a larger in-memory component at the cost of increased memory.
2. Reduce the compaction width of each stage by using smaller SSTs, which leads to more levels.
3. Use scheduling techniques to eliminate dependencies and execute compactions within a chain in parallel. The disadvantage of this approach is the violation of the growth factor, which increases I/O amplification.

Next, we examine each strategy for improving tail latency, concluding that in all cases, there is a negative impact on the amount of required memory or I/O amplification.

3.1 Reducing the *length* of compaction chains

It is relatively straightforward to reduce the length of compaction chains by either increasing the growth factor f or the size of L_0. However, both of these result in a significant increase in I/O amplification or memory usage. Increasing the growth factor from 10 to 20 would reduce the number of levels by one at the cost of doubling the I/O amplification [2].

Alternatively, increasing L_0 by a growth factor f (typically $8-10\times$) would reduce the number of levels by one and increase the compaction width f times. However, on servers that host several KV regions (typically hundreds or thousands [11]), this results in excessive memory usage up to f times.

Another way to reduce the length of compaction chains is to break a dataset into a larger number of smaller regions. Small regions employ fewer levels for the same growth factor f. However, this approach similarly increases the amount of memory required, as the collective in-memory component of all regions increases in size compared to using fewer regions. Therefore, such approaches cannot improve tail latency in a cost-effective manner.

A more practical approach to reduce the number of levels uses a tiering compaction step in L_0, as follows. The original LSM-tree [24] design requires that the entire L_0 is in-memory. Modern KV stores, such as RocksDB [15], deviate from the original design to increase the size of L_0, which results in fewer levels and less I/O amplification. However, to avoid increasing memory requirements, they keep two versions of L_0: a tiered L_0 (L_0 in RocksDB terminology) and a leveled L_0 (L_1 in RocksDB terminology), which are the same size. Tiered L_0 consists of overlapping SSTs, whereas L_1 consists of non-overlapping SSTs. Both L_0 and L_1 levels reside on the device without requiring memory. Then, it uses *memtables* as the in-memory component of the KV store. Each memtable, similar to SSTs, has a size of several tens of MB (64 by default).

Figure 3(b) shows the data path in RocksDB with incremental compactions and a tiering compaction step in L_0. When a memtable fills, the system converts the memtable to SST format and flushes it to L_0. When L_0 is full, RocksDB moves one SST at a time using a bottom-up approach from L_{n-1} to L_1 to free enough space for the SSTs of L_1. Then RocksDB reads the SSTs of L_0 and reorganizes them in non-overlapping units in-memory (tiering step). Finally, it flushes the resulting SSTs to the empty L_1. With this tiering step, RocksDB can use a large L_0, in the order of GB, with just

a few MB of memory budget for memtables, leading to almost one order of magnitude memory savings compared to designs that use an in-memory L_0 (e.g. the original LSM).

3.2 Reducing the *width* of compaction chains

Reducing the width of compaction chains is more intricate. Traditionally, LSM KV stores use relatively large SSTs (e.g., 64 MBs) to cater to HDD characteristics and reduce constant overheads in the KV store design, such as for managing guards in memory or for recovering from failures. However, large SSTs significantly increase the width of compaction chains, resulting in reorganizing hundreds of MB in each compaction step. Although the emergence of flash devices creates an opportunity to reduce SST size and compaction width, it turns out reducing the size of SSTs alone is not enough to significantly improve tail latency.

Problematically, with small SSTs, the number of levels in the LSM store increases, which in turn inflates I/O amplification. Figure 4(b) shows that if we change the SST size in RocksDB from 64 MB to 8 MB and adjust L_1 size from 256 MB to 64 MB to maintain the growth factor to f=8, the number of levels grows from 5 to 7 and I/O amplification increases by 69%. As a result, current systems cannot reduce SST size without significantly increasing I/O amplification.

Figure 3(a) illustrates the design of a typical LSM KV store with incremental compaction without tiering. If we try to reduce the compaction chain width by reducing the SST size and removing the tiering compaction step at L_0, then I/O amplification increases dramatically. Each SST in L_0 overlaps with the entire L_1, resulting in excessive I/O amplification. Figure 4(a) shows that using the default configuration of RocksDB, after disabling the tiering compaction, the I/O amplification increases up to 128×, when L_1 is 256 MB in total. We reduce SST size from 64 MB to 8 MB, which effectively increases the growth factor between L_0 and L_1 from f=4 to f=32. Therefore, current designs can only reduce the size of L_1 to be f times the SST size while increasing the number of levels, as we show in Figure 4(b).

Finally, to reduce I/O in incremental compaction partially, operators often use over-provisioning of space as follows: They keep each region at about 40% of its maximum size [10]. Essentially, this keeps the last level relatively empty with fewer SSTs. When a compaction chain forms, the last stage incurs less work because the last-level compaction involves, on average, fewer than f SSTs and reduces I/O amplification by up to 20% [7].

3.3 Eliminating dependencies in compaction chains

Recall that compaction chains are sequences of dependent compactions from level to level. We can use two alternative approaches to perform these compactions to reduce the delay until we can free memory for the blocked operation.

Modern LSM stores, including RocksDB, may allow levels to temporarily exceed their maximum size (compaction debt). In that case, they can perform compactions in the background, ignoring the size limit for each level while serving regular requests. This approach results in a higher growth factor than f across levels and, therefore, increases I/O amplification. To avoid excessive I/O amplification, this approach places an upper per-level limit on the size of the compaction debt. Zhou et al. [34] expect to have compaction debt for short periods. In our evaluation, we show that allowing for compaction debt alone does not suffice to significantly reduce the tail latency.

On top of the compaction debt design, scheduling-based solutions also proactively perform compactions in the background. Figure 3(c), illustrates ADOC, a state-of-the-art scheduling-based solution, which extends RocksDB design with compaction debt. In short, scheduling those compactions out-of-the-critical path, strives for insert requests to always have free space at each level, therefore preventing compaction chains from forming. While scheduling compactions in the background of a running workload can drastically improve the tail latency, it comes at almost double the cost of I/O amplification (as shown in our evaluation § 6).

3.4 Summary

Overall, addressing the root causes of tail latency is a significant challenge in modern KV designs. Existing designs trade tail latency with I/O amplification or memory usage, and vice versa. In many cases today to mitigate tail latency, operators resort to circumstantial actions by either over-provisioning resources or rate-limiting storage servers [3] to avoid long queues of requests. Next, we discuss how *vLSM* reduces tail latency without such compromises.

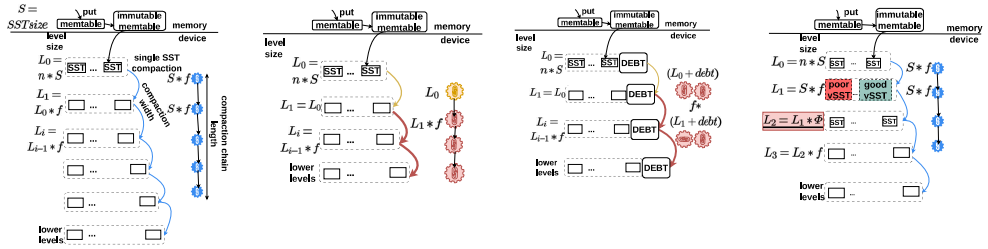
4 *vLSM*

Unlike existing LSM works, *vLSM* manages to optimize for all three metrics: memory usage, tail latency, and I/O amplification. To achieve that, *vLSM* carefully reduces both the maximum width and maximum height of compaction chains in LSMs by combining four key ideas. We discuss these ideas and their synergy in this section.

Figure 3(d) illustrates the write path of *vLSM*. Initially, *vLSM* writes KV pairs in the memtable in memory. When a memtable is full, *vLSM* serializes it to an SST and flushes it to L_0. This step proceeds even if there are SSTs present from previous flush operations until the size of L_0 reaches a predefined max number of SSTs, similar to RocksDB. At this point, *vLSM* behaves differently from RocksDB, as we explain next.

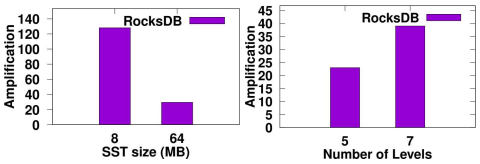
4.1 Reduce width: No tiering & Smaller SSTs

First, *vLSM* does not have to wait for L_0 to become entirely full as it does not have to perform a tiering compaction to L_0. Instead, it picks a single SST from L_0, in FIFO order each time, and compacts it to L_1 to free space in L_0 for a memtable. As a



(a) LSMi (b) RocksDB-IO (c) ADOC (over RocksDB) (d) *vLSM*

Figure 3. The main design points for mitigating high tail latency.



(a) (b)

Figure 4. Impact on I/O amplification in RocksDB of (a) compacting a single SST between L_0 and L_1 when not maintaining growth factor between L_0 and L_1 and (b) the number of LSM levels when using 8 MB SSTs emulating LSMi.

result, in *vLSM* L_0 serves merely as a queue of SSTs to handle traffic bursts efficiently. Although its size needs to be limited for read performance purposes (as in RocksDB), it does not impose any limitations in reclaiming space with compactions due to eliminating the tiering step (unlike RocksDB).

Second, contrary to RocksDB, *vLSM* always compacts a single SST from L_0 to L_1. To control I/O amplification, *vLSM* uses by design an L_1 with size f times the size of a single SST compared to RocksDB where L_1 size is equal to L_0. As a result, the width of the compaction chain now depends on the SST size and the growth factor, which affects the average overlap of SSTs between adjacent levels.

Naively removing the tiering step and decreasing the SST size to reduce the compaction width according to the SST size increases the number of LSM levels, by a constant f across all levels. As a result, *vLSM* would need to compact more levels for the same dataset size, affecting both tail latency and I/O amplification. For example, using 64 MB SSTs with a 4 TB region capacity, *vLSM* would require two additional levels at $f = 8$.

Therefore, *vLSM*'s third design aspect is it uses a larger growth factor Φ between L_1 and L_2 compared to f across the rest of the levels, to maintain the number of levels the same as when using a tiering compaction. Increasing the growth factor, increases merge amplification [2] between L_1 and L_2 [2] because it affects the amount of overlap across the two levels.

To overcome this challenge, *vLSM* introduces its overlap-aware SSTs (vSSTs) technique in L_1 to limit merge amplification between L_1 and L_2. *vLSM* creates in L_1 vSSTs of variable overlap, as follows. When creating a vSST in L_1, *vLSM* examines the overlap of the new vSST with L_2. Then, it allows some vSSTs to be smaller than regular SSTs with limited overlap to L_2. By design *vLSM*, creates two types of vSSTs: *poor* vSSTs with overlap more than f and *good* vSSTs with overlap less than or equal to f. Then, during compaction from L_1 *vLSM* compacts only good vSSTs until it frees space for an SST in L_1 for the next L_0 SST. Next, we discuss how vSSTs work in more detail.

4.2 Reduce length: Larger growth factor Φ in L_1 & overlap-aware SSTs

To maintain the same number of levels as tiered approaches (RocksDB), *vLSM* uses a larger growth factor in L_1. Naively done, this would increase the growth factor up to 32×. To prevent the increase in the number of levels due to the smaller SSTs, as observed in the design of LSMi in Figure 3(a). The increased growth factor decreases the number of SSTs in L_1 and results in each L_1 SST to overlap with a larger number of L_2 SSTs. Generally, the average overlap of SSTs between adjacent levels is f in LSM KV stores. However, in *vLSM* L_1 SSTs overlap with up to 32× more L_2 SSTs, resulting in a significant increase in I/O amplification. To overcome this

challenge, *vLSM* introduces overlap-aware SSTs in L_1 to control the overlap between L_1 and L_2 SSTs. Next, we discuss how vSSTs work in more detail.

vLSM uses overlap-aware vSSTs only in L_1, while it keeps fixed-size SSTs in all other levels. *vLSM* takes advantage of the following property: since each L_0 SST usually contains keys that cover the whole key range of L_1, it reorganizes the vSSTs in L_1 on every L_0 compaction.

During each compaction from L_0 to L_1, before appending a key to the current in-flight L_1 vSST, it checks the overlap of the vSST with the SSTs of L_2 as if the key is in the vSST. If the overlap exceeds f, it closes the current vSST, flushes the vSST to L_1, and starts the next vSST. However, *naively* closing a vSST when the overlap exceeds f could lead to many small vSSTs in L_1 (poor vSSTs), with high I/O amplification. Next, we discuss how *vLSM* creates a bounded number of vSSTs while controlling I/O amplification for L_1 vSST.

4.2.1 Good and poor vSSTs.

To prevent the excessive creation of poor vSSTs that lead to high I/O amplification due to the fixed-size SSTs of L_2, *vLSM* uses a heuristic that limits the number of poor vSSTs in L_1. *vLSM* sets the minimum size of the vSSTs to $S_m = 1/f \times S_M$, where S_M is the size of the fixed-size SSTs.

Essentially, *vLSM* needs to solve the following optimization problem in L_1. Given the keys in L_1 and the fixed-size SSTs in L_2, we need to divide L_1 in vSSTs (vSSTs) with the following constraints.

Each vSST should have a size between a minimum (S_m) and a maximum (S_M) number of bytes. S_M is the maximum size for a fixed-sized SST in the system, e.g. 64 MBytes in the default RocksDB configuration and between 4-64 MB in *vLSM*. S_m depends on the overall design of the KV store. Generally, it should not be too small, e.g. in the order of KB, because this will result in a significant increase in per-SST overheads, such as the size of memory guards (in the memory part of the index) and the cost of manifest flush operations in RocksDB. In our work, we set $S_m = 1/f \times S_M$, which leads to at most f more SSTs in L_1, compared to RocksDB's fixed-size SST approach.

Assuming each vSST has overlap O (ratio of bytes between vSST and next-level SSTs) with the SSTs in L_2, *vLSM* needs to divide L_1 in vSSTs in a manner, such that there is always *enough* vSSTs with overlap O less than f. If *vLSM* achieves this, during the next compaction from L_1 to L_2, the system will pick a subset of these vSSTs for compaction, freeing adequate space in L_1 for the next L_0 SST. As an approximation, instead of creating an adequate number of good vSSTs with appropriate overlap, *vLSM* tries to maximize the number of such vSSTs. Therefore, we can state the objective as:

Objective: Given the keys in L_1 and the fixed-size SSTs in L_2, divide L_1 in overlap-aware vSSTs with size between S_m and S_M, such that you maximize the cumulative size of vSSTs that exhibit overlap O less than f with L_2 SSTs.

vLSM uses a heuristic approach to create vSSTs in L_1. First, it tracks the overlap O of the next vSST with L_2. Then, it keeps adding keys to the vSST until:

- If overlap O becomes quickly larger than f, then it closes the vSST when it reaches size S_m. We call these *poor* vSSTs as they could result in high compaction costs. *Poor* vSSTs that have overlap more than f and their size is always m.
- If overlap O is less than f, it keeps appending to the vSST until either the overlap becomes f or its size reaches S_M. We call these *good* vSSTs as they result in low compaction cost. *Good* vSSTs have overlap up to f, and their size by necessity is between [m,M].

During the next incremental compaction from L_1 to L_2, *vLSM* picks as many *good* vSSTs as necessary to free adequate space in L_1. Although, in principle, it is possible that there are not enough *good* vSSTs in L_1, due to the growth factor F between L_1 and L_2, which is larger than f, our evaluation shows that for values of Φ up to 32, the system is able to always find *good* vSSTs. By design, *vLSM* allows creating *poor* vSSTs that overlap (usually larger than f) with a large part of L_2. These *poor* vSSTs will not be used during the next compaction, but they will be reorganized later when a new L_0 SST is compacted to L_1. At that point *vLSM* will try to create again *good* vSSTs, even though it is starting from *poor* vSSTs in L_1. *vLSM* takes advantage of *poor* vSSTs by making them have really large overlap with L_2 SSTs and this forces the remaining *good* vSSTs to have lower overlap less than f. Also, *vLSM* removes the constraint of fixed-size SSTs in L_1 to free space for the next L_0 SST by allowing the compaction of multiple *good* vSSTs to L_2 without increasing I/O amplification.

Given the value for $S_m = 1/f \times S_M$, *vLSM* creates at most f times more vSSTs in L_1 than fixed-size SSTs (each of size S_M) it would have. We could reduce this number further with a larger value for S_m. However, increasing S_m has a detrimental effect: If we keep growing a *poor* vSST, this reduces the opportunity for identifying low-cost vSSTs by *absorbing* a key range that on its own could form a *good* vSST. Generally, other values for S_m, we speculate smaller than the value we currently use, could be appropriate for different KV store designs, e.g, with lower constant overheads compared to RocksDB. Next, we discuss how *vLSM* selects *good* vSSTs.

4.2.2 Selecting *good* vSSTs to compact.

vLSM uses the same compaction scheduler as RocksDB to select the best *good* vSSTs in the following manner. The default compaction scheduler of RocksDB first randomly selects 50 SSTs from the source level (L_1 in our case). Due to the size of L_1, it typically examines all its SSTs. Then, it calculates for each vSST the ratio $\frac{overlap\_in\_bytes\_of L_2}{SST\_size\_of L_1}$. As a result, it chooses the largest vSST(s) of L_1 in size with less overlap with L_2. Finally, it compacts a set of SSTs from L_1 whose cumulative size equals the fixed SST size (S_M).

5 Experimental Methodology

Experimental platform. Our testbed consists of a single server that runs the key-value store and YCSB [25] or db_bench. Our server is equipped with two Intel(R) Xeon(R) CPU E5-2630 running at 2.4 GHz, with 16 physical cores for a total of 32 hyper-threads and with 256 GB of DDR4 DRAM. It runs CentOS 7.3 with Linux kernel 3.10.0. The server has 1 Samsung SSD 970 EVO Plus 2TB device model.

Workloads. In our evaluation, we use YCSB [25] and db_bench [14]. We configure YCSB to use 15 client threads with four regions, and each shard uses four threads for background I/O operations (compactions), on top of *XFS* with disabled compression and jemalloc [13], as recommended. We configure RocksDB to use direct I/O, as recommended by RocksDB [16], and we verify that this results in better performance in our testbed than using the kernel page cache. We set the size of L_1 to 256 MB, the growth factor to 8, and the maximum number of LSM levels to 5. We use YCSB to benchmark RocksDB and *vLSM* with 350 million key-value pairs for Load A and 70 M operations for Run A-D workloads. We set the size of each key-value pair to 200 B. Finally, we configure all KV stores overflow limit to 64 MB for all configurations to accelerate all LSM stores to reach a stable state. We set the default memtable and SST size to 64 MB for RocksDB, unless stated otherwise.

We measure tail latency of write operations with YCSB Load A under the default uniform key distribution. We choose this workload similar to previous studies [1, 30] because it represents write-intensive workloads. We configure the different baselines throughout this study, following the general guidelines in [15] and introducing the required parameter modifications, as discussed for each case. Furthermore, we use YCSB Run A, B, and D, which contain different mixes of read and write operations to evaluate the performance of the different systems under mixed workloads. Moreover, to understand how *vLSM* affects mixed workloads, we extend the YCSB completion queue to differentiate between read and write operations. For Runs A, B, and D, we present a breakdown of the tail latency of read and write operations separately, e.g. for Run A we indicate write and read latency as Run A-W and Run A-R, respectively.

We use db_bench to evaluate production workloads as in Meta's datacenters [3]. We use the same configuration noted in [3] for the KV store population but increase the number of KVs from 50 million to 908 million while maintaining the remaining parameters the same. We increase the number of KVs to fill all the LSM levels except the last one and ensure we measure the system in a steady state. We measure with db_bench only I/O amplification as the modifications needed to measure tail latency (as we do with YCSB) require significant changes that might change the benchmark's behavior. However, we do not expect tail latency to differ substantially from the results we report with YCSB.

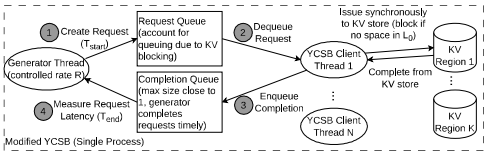


Figure 5. Modified YCSB to measure tail latency in an open-loop manner at a controlled (fixed) request rate.

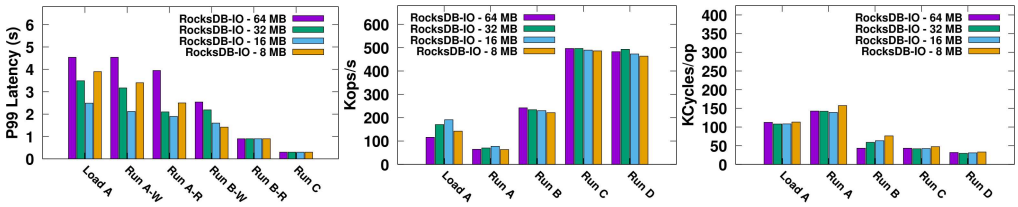
Tail latency. Similar to previous work [1, 30, 31], we use two metrics to study tail latency: a) *write stalls* and b) *P99 latency* as seen by the application. Write stall metric is the time server throughput drops to zero due to memory unavailability at L_0. They offer an indirect view of operation completion times.

There are two methods to measure tail latency in a system: a) Each client issues a number of requests and stops generating new requests until it gets a reply from the server. b) Requests are sent at a constant rate regardless of the timing of the responses received from the server.

Case (a) leads to coordinated omission scenario [18], where the request rate is inadvertently tied to the server's ability to process requests, skewing tail latency measurements [21, 33] due to reduced queueing. In addition, in cloud environments with many independent clients, even if individual clients bound their requests to the KV store, the cumulative number of requests from all clients cannot be controlled. Therefore, in our methodology, we use (b), which reflects the tail latency seen by applications. To examine different configuration points, we bound in each experiment the *rate* of the requests.

We modify YCSB [5, 25] to measure tail latency, as shown in Figure 5. We first decouple the generation of KV pairs from client threads by introducing a new thread that generates requests and places them in an unbounded queue at a fixed request rate (①). Each request includes the operation type, key, and timestamp. Several YCSB threads dequeue requests from the unbounded queue and issue them to the KV store *synchronously* (②) via synchronous operations. When each request completes, the corresponding YCSB thread detects completion and moves the request to a completion queue (③). All YCSB threads share the request completion queue. The generator thread dequeues completed requests and measures end-to-end, per-request tail latency using the request issue timestamp (④). We ensure accuracy by using the same core clock counter for end-to-end measurements, avoiding discrepancies caused by unsynchronized core clocks.

In our approach, the single generator thread limits the maximum request generation and completion rate. We find that a single thread can generate and complete requests at a maximum rate of 1.5M requests/s, significantly higher than



(a) Latency (b) Throughput (c) CPU Efficiency

Figure 6. RocksDB P99 tail latency (left), throughput (middle), and CPU efficiency(right) for all YCSB workloads while varying the SST size between 8-64 MB.

the maximum throughput of RocksDB and *vLSM*, which is approximately 200K request/s.

Additionally, we use the following technique to set the appropriate request rate without inducing excessive queuing effects. We differentiate between maximum throughput and sustainable throughput. We conduct profiling runs setting the generator to operate at a large rate and we identify the sustainable throughput for each system. Previous works [1, 30] measure tail latency at low request rates. In this paper we measure tail latency under broad load conditions, up to 95% of its sustainable throughput, which provides a clearer understanding of the impact of each design on tail latency.

CPU efficiency. Finally, we measure CPU efficiency of the KV store in cycles/op. *cpu_util%* (in the range [0,1]) is the average of CPU utilization among all processors, excluding idle and I/O wait time, as given by *mpstat*. As $cycles/s$ we use the per-core clock frequency. Finally, $average\_ops/s$ is the throughput reported by YCSB, and $\#cores$ is the number of system cores, including hyper-threads.

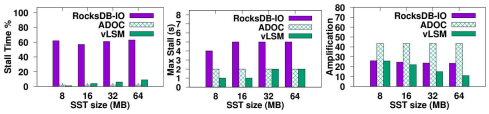
$$cycles/op = \frac{cpu\_util \times cycles/s \times \#cores}{average\_ops/s}$$

Baselines. In our evaluation, we use as baselines the design variations discussed during our design: RocksDB, RocksDB-IO, ADOC and *vLSM*. RocksDB is the default configuration of RocksDB. RocksDB-IO is a variant of RocksDB with overflow (debt) disabled. We use RocksDB-IO to evaluate the effects of tiering compaction and how it affects tail latency without the mitigating impact of debt which also increases I/O amplification. We use ADOC as the state-of-the-art scheduling approach that optimizes tail latency while also reducing I/O amplification compared to RocksDB (but has worse I/O amplification than RocksDB-IO).

6 Experimental Evaluation

To understand the impact of each design on tail latency and how effective *vLSM* is, we examine the following questions:

1. How does *vLSM* perform compared to RocksDB and ADOC regarding tail latency, throughput, I/O amplification, and CPU efficiency?



(a) Write stalls (b) Max stall (c) I/O amplification

Figure 7. RocksDB, ADOC and *vLSM* write stalls (left), max stall time (middle), and I/O amplification (right) for YCSB's Load A while varying the SST size between 8-64 MB.

2. How does *vLSM* reduce compaction chain length & width?
3. How does *vLSM* behave for different growth factors Φ and SST sizes?

Next, we examine each of these aspects.

6.1 Compaction chains and tail latency in the state-of-the-art

First, we examine the impact of SST size on tail latency in RocksDB-IO and ADOC [31] to establish the context for modern, state-of-the-art KV stores (Figure 6). We run YCSB workloads Load A and Run A-D and report tail latency, throughput, and CPU efficiency using four regions and four client threads. Figure 6(a) shows tail latency for RocksDB decreases compared to 64 MB SST size to 1.4× and 1.8×, respectively, for 32 and 16 MB SST size. Then, for 8 MB, it increases again, as in the 32 MB SST size case.

Furthermore, SST size does not affect I/O amplification and stall time, as shown in Figure 7(a). The compaction chains formed in RocksDB consist of two stages. The first stage, includes the tiered L_0, and L_1 compactions, whose width is unaffected by the SST size. However, SST size affects the width of the chain of the lower levels. As a result, we observe the drop in latency by up 1.8×; however, tail latency for RocksDB remains in the order of seconds. We observe write stalls in RocksDB for all SST sizes, as shown in Figure 7(a) and 7(b). The write stalls remain constant for all SST sizes, and the maximum stall time is in the order of 4-5

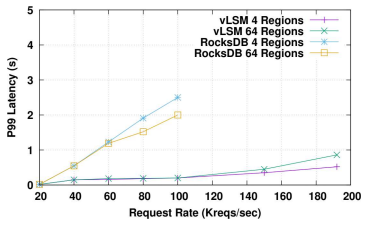
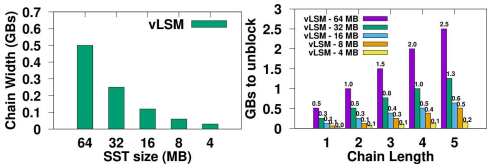


Figure 8. P99 latency for *vLSM* and RocksDB at different request rates.



(a) Chain Width (b) Chain Length

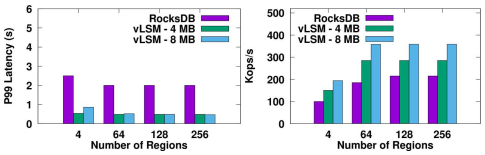
Figure 9. *vLSM* chain width (left) and chain length (right) for different SST sizes.

seconds. This is an effect of tiering compaction in RocksDB, which compacts 2.5 GB of data from L_0 on average. Although ADOC increases the chain width due to the combination of tiering compaction and the level overflow it dramatically reduces stalls similarly to *vLSM*. However, ADOC increases I/O amplification by 1.8× compared to RocksDB and *vLSM*.

6.2 Compaction chains and tail latency in *vLSM*

Next, we examine how *vLSM* behaves with respect to compaction chain length and width and how different workloads affect tail latency and write stalls.

Request rate. Figure 8 shows the P99 latency for RocksDB and *vLSM* with 8 MB SSTs and different request rates. We stop measuring RocksDB after 100K requests/s as this is our setup's maximum sustainable throughput for RocksDB. We find that when RocksDB serves more than 60K requests/s, tail latency exceeds 1 second while *vLSM*'s tail latency remains below 1 second for all request rates. We note that the request rate does not significantly affect the tail latency for RocksDB since the amount of work done in L_0 is large for all request rates due to tiering compaction. In contrast, *vLSM*'s tail latency is not affected so much by the request rate since the amount of work done in L_0 is reduced due to the reduced compaction chain length and width. For four regions, *vLSM* achieves tail latency up to 4.3× and for 64 regions up to 4× less than RocksDB.



(a) Latency (b) Throughput

Figure 10. Impact of *vLSM* on tail latency and throughput under YCSB Load A and varying number of regions.

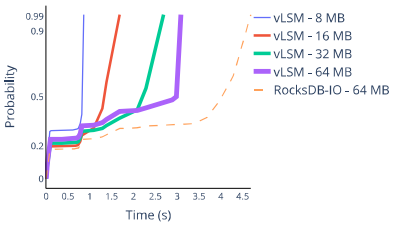
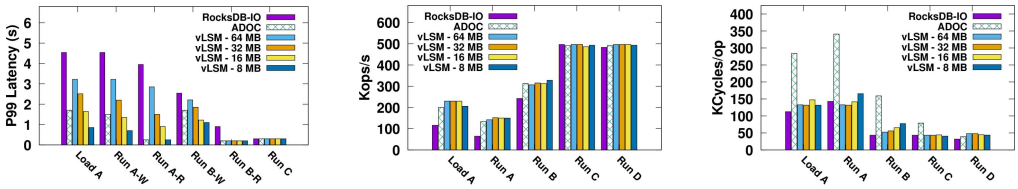


Figure 11. Open loop YCSB Load A tail latency CDF for RocksDB and *vLSM*.

Compaction chain width. Figure 9(a) shows the compaction chain width with different SST sizes for *vLSM*. For *vLSM* chain width decreases as the SST size up to 320× compared to RocksDB from 10 GB to 32 MB for 4 MB SSTs. The improvement in the chain width is due to the removal of tiering compaction and the reduction of the SST size.

To measure the impact of the compaction chain width in write stalls and tail latency, we use YCSB Load A and vary the SST size between 8-64 MB, while maintaning the number of regions at 4 in Figure 7(a). We see that *vLSM* reduces write stalls up to 60% compared to RocksDB-IO when comparing *vLSM*'s 8 MB SSTs with RocksDB-IO as *vLSM* reduces the amount of data compacted from L_0 to L_n 51× compared to RocksDB-IO. Additionally, in Figure 7(b), we observe that *vLSM* reduces the maximum stall time by 5× compared to RocksDB-IO. We note that the maximum stall time for RocksDB-IO is in the order of seconds for all SST sizes, while for *vLSM*, it is in the order of a second. Additionally, we note that *vLSM* does not only improve P99 latency but also P50 and P90. In Figure 11, we measure different percentiles of RocksDB-IO with 64 MB SSTs and *vLSM* with 8 to 64 MB SSTs for YCSB Load A. We observe that *vLSM* for 8 MB SSTs improves P50 up to 4× and P90 up to 4.88× compared to RocksDB-IO.

Compaction chain length. To examine the impact of *vLSM* on compaction chain length, we measure the amount of data (GBs) compacted before a memtable can be flushed to L_0. We vary the number of levels from 1 to 5, which corresponds



(a) Latency (b) Throughput (c) CPU Efficiency

Figure 12. P99 latency (left), throughput (middle), and CPU efficiency (right) of *vLSM* for all YCSB workloads while varying the SST size between 8-64 MB.

to different chain lengths. In Figure 9(b), we find that the amount of data in *vLSM* decreases up to 20× compared to RocksDB from 10 GB to 0.5 GB for 8 MB SSTs.

To measure the impact of the compaction chain length on tail latency, we vary the number of regions ranging from 4 to 256 in Figure 10(a). The number of regions affects the number of levels in the LSM tree since, for more regions, the number of levels and the compaction chain length decrease. For our dataset size, with four regions, the LSM tree has five levels, and the compaction chain length is four, while for 16 regions and above, each shard has three levels, and the compaction chain length is 3. We find that RocksDB, even with sharding, has tail latency in the order of seconds for all shard counts and after 16 regions, where tail latency improves by 1.4× compared to four regions, it remains constantly high. This stems from the fact that the amount of work done from RocksDB comes from tiering compaction in L_0, which on average is 2.5 GBs for five levels and 1.2 GB for three LSM levels. Compared to *vLSM*, which compacts on average 0.32 GB for five levels and 0.19 GB for three LSM levels for 8 MB SSTs and 0.16 GB for five levels and 0.12 GB for 4 LSM levels for 4 MB SSTs. *vLSM* achieves 4.23× better tail latency than RocksDB with 8 MB SSTs and 5× better tail latency with 4 MB SSTs for four regions. For 16 regions and above, *vLSM* with 4 MB SSTs improves tail latency by 6% compared to four regions since for 16 regions 4 MB SSTs require 1 more level in the LSM tree to maintain Φ below 64.

Figure 10(b) shows that *vLSM*, improves throughput for all SST sizes up to 9×. Furthermore, for 4 MB SSTs *vLSM* achieves 1.76× better latency in the expense of 20% more I/O amplification since we need to add one more level to maintain the 32 ratio between L_1 and L_2. We also note that *vLSM* uses the same amount of memory for the in-memory component as the base RocksDB configuration 2 memtables (as suggested by RocksDB) equal to the SST size.

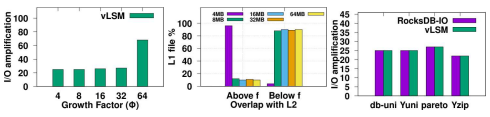
6.3 Sensitivity analysis for *vLSM*

Sensitivity to the workload. Next, we compare the impact of the SST size on all YCSB workloads. We focus especially on mixed workloads, Run A and Run B that have a mix of read and write operations to understand how writes affect read tail latency. We measure throughput, P99 Latency, and CPU efficiency.

In Figure 12, we run all YCSB workloads for Run A that has 50% read and 50% update operations *vLSM* achieves up to 2× better throughput since update operations either complete faster on higher levels or due to the reduced SST size that requires less work. We note here that the tail latency for reads in Run A also improves up to 12.5× for 8 MB SSTs compared to RocksDB-IO since read operations have to wait for significantly less time due to the large reduction in write stalls. Furthermore, we observe the same trend for Run B, which has 95% read and 5% update operations.

For Run C that has 100% read operations *vLSM* achieves the same throughput as RocksDB-IO, as SST size does not affect how an LSM tree handles read operations. Only the block cache and data block sizes can significantly affect reads. Finally, we measure CPU efficiency and observe that for all workloads containing write operations, *vLSM* decreases CPU efficiency, as it has to check for every KV pair, the overlap with the next-level SSTs, and perform compactions more often. *vLSM* decreases CPU efficiency up to 4% compared to RocksDB-IO. We note that ADOC for write heavy workloads (Load A) and mixed workloads (Run A) decreases CPU efficiency up to 2.2× due to the overflow mechanism that increases the average compaction size by 1.8×. Furthermore, due to the overflow mechanism, ADOC decreases CPU efficiency for Run C up to 2× since it has to perform compactions to reduce the debt from the previous write heavy workloads.

Sensitivity to the KV distribution. In this experiment, we investigate *vLSM* vSSTs behavior for different workloads observed in large-scale applications. We run two benchmarks with different key distributions: a) YCSB with uniform and Zipfian and b) db_bench with uniform and Pareto. Pareto distribution reflects Meta's workloads [3]. Also, both uniform key distributions of YCSB and db_bench generate a high-entropy workload typically seen in large-scale applications. We measure *vLSM* with 8 MB and RocksDB-IO with 64 MB SSTs, respectively. We set the growth factor between L_1 and L_2 to 32 for *vLSM*. In Figure 13(c), we find that *vLSM* achieves



(a) Growth factor Φ (b) File overlap (c) KV distribution

Figure 13. Sensitivity analysis for different vaules of Φ, showing I/O amplification (left) and percentage of vSST overlap (middle). Sensitivity analysis with different key distributions (right).

SST size (MB)	P99 lat (ms)	xput (Kops/s)	CPU (Kcycles/op)
2	421	39	320
4	502	161	196
8	860	194	140

Table 1. Sensitivity analysis of *vLSM* based on SST size.

the same I/O amplification with RocksDB-IO for all distributions. We conclude that variable size SSTs of L_1 keep I/O amplification equal to RocksDB-IO for the aforementioned KV distributions.

Sensitivity to Φ. In *vLSM*, the role of the growth factor Φ from L_1 to L_2 is to reduce the number of LSM levels, therefore *vLSM* uses larger values than what is typical in LSM (e.g. $f = 8 - 10$). We examine different values for Φ by varying the SST size between 64 - 4 MB. By varying the SST size, we affect the compaction size between L_0 and L_1. Varying the SST size directly affects the L_1 size since compactions from L_0 to L_1 must maintain the growth factor f.

In Figure 13(a), we find that *vLSM* cannot sustain I/O amplification below f when the L_2/L_1 ratio reaches 64 (4 MB SSTs). To verify why *vLSM* cannot sustain I/O amplification below f when the L_2/L_1 ratio reaches 64 for 4 MB SSTs, we analyze the file sizes for 8 MB and 4 MB SSTs L_1. We measure the file sizes created on L_1 and separate the files into two categories. The first category contains files that are exactly on the boundary of $1/f$ that exceeds f overlap and the remaining files above $1/f$ that do not exceed f overlap. In Figure 13(b), we find that for 8 MB SSTs, 90% of the files do not exceed $1/f$ and 10% exceeds overlap f across all compactions. As a result, every L_1 compaction finds a file that does not exceed f overlap. Thus, *vLSM* can sustain I/O amplification below f for 8 MB SSTs. In contrast, for 4 MB SSTs, 94% of the files exceed f overlap, and 94 % of the files are exactly on the $1/f$ boundary. On average, the work per byte for 4 MB SSTs is 8192× more than 8 MB SSTs when the L_2/L_1 ratio becomes 64.

Sensitivity to SST size. Smaller SSTs have the potential to reduce tail latency by reducing the amount of work in compaction chains. Although modern storage devices exhibit high IOPS and are not as sensitive to SST size, small SSTs increase the number of SSTs, which can affect other overheads. Especially in modern KV stores that are designed for large SSTs, certain aspects of the design do not necessarily cater to small SSTs. For instance, RocksDB manages recovery by flushing a system manifest to the device after every memtable flush and SST compaction. Small SSTs will lead to more manifest flush operations. Similar issues exist with guard management in-memory and the layout of internal SST metadata.

We explore the effect of smaller SSTs in *vLSM*, focusing on throughput, latency, and CPU efficiency in Table 1. We find that *vLSM*'s performance drops below 8 MB SSTs when using SST sizes between 2-4 MB. Our findings reveal a 15% decrease in throughput and CPU efficiency at 4 MB due to frequent compactions and overlap checks for each KV pair despite a 58% improvement in P99 latency. Moving to 2 MB SSTs further reduces throughput by fivefold and CPU efficiency by 2.38×, with only a 15% latency improvement over 4 MB SSTs. Furthermore, the increased CPU overhead as the SST size decreases stems from RocksDB's current design. Specifically, RocksDB is optimized to handle larger SSTs and synchronously issues all the I/O path. Future LSM-tree designs could reduce CPU overhead by batching I/O operations and compactions and reach for far smaller SST sizes with better performance compared to *vLSM*. *vLSM*'s performance is optimal with 8 MB SSTs, as it achieves the best balance between throughput, tail latency, and CPU efficiency.

7 Related Work

We group related work that directly targets improving tail latency in KV stores in the following three categories: (a) reduce the width of the compaction chains, and (b) improve concurrency through I/O scheduling approaches.

Compaction chain reduction. MatrixKV [30] is an LSM KV store that removes the tiering step between L_0 and L_1. It increases the size of L_0 to reduce the number of LSM levels and dynamically creates SSTs that compact with L_1. As a result, it reduces both I/O amplification and tail latency. NoveLSM [20] adds an extra level between memtables and storage to store immutable memtables in NVM. This allows NoveLSM to have a large temporary buffer (NVM) before compacting memtables to the storage. ChameleonDB [32] uses lazy leveling with NVM for storage. This reduces I/O amplification significantly since it uses tiering for the N-1 levels except the last. However, when the N-1 levels become full, the system will stall for a large amount of time since it will need to compact each full level to the next thus observing long tail latency.

MioDB [12] uses NVM to store skiplists instead of SSTs for the N-1 levels, replacing compaction operations with

pointer updates. Compacting only the last level to the storage. ListDB [22] builds skiplists using the WAL, avoiding costly memtable se,deserialization operations. Also, similarly to MioDB, uses pointer operations to update skiplists, avoiding compactions using the Zipper compaction. Problematically, all these memory-based approaches keep more data in (DRAM or NVM) memory, rendering them much less cost-effective compared to memory-frugal solutions like RocksDB and vLSM.

Compaction Strategies. Other work in LSM KV stores has targeted relevant aspects of compaction strategies. Spooky [7] proposes a hybrid compaction strategy to reduce I/O amplification without increasing space amplification on the device. It uses full compaction for the higher levels of the LSM-tree and incremental compaction for the lower levels. Although this is not the goal of Spooky, this approach will increase tail latency because of the full compaction steps at the higher levels of the store. Dostoevsky [6] introduces lazy leveling that combines tiering and leveling in the same LSM-tree to reduce I/O amplification significantly while sacrificing up to 10% of the overall space. Like Spooky, Dostoevsky will result in high tail latency when the N-1 levels become full since full compactions will occur for each level until the last level. Also, due to tiering compaction, Dostoevsky cannot use incremental compaction for the tiered levels, increasing exponentially the compaction width.

Compaction scheduling. bLSM [26] proposes a scheduling approach to reduce the tail latency of LSM KV stores. It uses a spring and gear algorithm to free a portion of each level during compaction. bLSM uses full (instead of incremental) compaction, however, stops each compaction step at scheduled points, to ensure progress at all levels. Silk [1] identifies two root causes of write stalls in LSM KV stores: (a) the L_0 to L_1 compaction and (b) the scheduling of I/O operations. It proposes a scheduling approach that prioritizes the compaction of the L_0 level to reduce tail latency and defers the compaction of the higher levels to execute when the system is not under heavy load.

ADOC [31] identifies that SILK increases memory consumption up to 22% and instead tries to reduce tail latency using the overflow capabilities of incremental compaction by allowing levels to exceed their size limits. As a result, ADOC reduces tail latency by sacrificing I/O amplification. To prevent excessive I/O amplification overheads, ADOC adjusts the number of compaction threads and batch sizes to reduce the period when the system is overflowing. However, this requires the server to have available CPU and memory resources. Calcspar [34] implements a scheduler for systems deployed in cloud environments where IOPS per second are limited.

Calcspar differentiates between high-priority requests (user-facing requests) and low-priority requests (compaction and prefetching requests). It prioritizes user-facing requests and opportunistically compacts L_1 to L_2 levels, while below L_2, it defers compaction for a short period. However, like ADOC, Calcspar might observe high I/O amplification overheads if the system defers compaction for an extended period. *vLSM* is compatible with all of the above scheduling approaches and could further reduce tail latency by adopting one of them based on the deployed environment.

8 Conclusion

In this paper, first we analyze the factors that affect tail latency in modern KV stores and the trade-off with I/O amplification and the amount of memory. We use the notion of compaction chains to understand how different parameters manage this trade-off and we show that modern designs and existing techniques are not able to optimize at the same time all three aspects: tail latency, I/O amplification, and memory. Then we present *vLSM* an LSM KV store that reduces tail latency by reducing both the width and length of compaction chains, without increasing I/O amplification and the amount of memory required. *vLSM* reduces (a) chain width by using small SSTs and eliminating the tiering compaction and (b) chain length by using a larger growth factor across the first device device levels (L_1, L_2) and introducing overlap-aware SSTs (vSSTs). Compared to RocksDB, *vLSM* reduces tail latency by 4.8× (12.5×) for writes (reads).

References

[1] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. Silk+ preventing latency spikes in log-structured merge key-value stores running heterogeneous workloads. *ACM Trans. Comput. Syst.*, 36(4), may 2020.

[2] Nikos Batsaras, Giorgos Saloustros, Anastasios Papagiannis, Panagiota Fatourou, and Angelos Bilas. Vat: Asymptotic cost analysis for multi-level key-value stores, 2020.

[3] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, FAST'20, page 209–224, USA, 2020. USENIX Association.

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

[5] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[6] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 505–520, New York, NY, USA, 2018. Association for Computing Machinery.

[7] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. Spooky: Granulating lsm-tree compactions correctly. *Proc. VLDB Endow.*, 15(11):3071–3084, jul 2022.

[8] Discord. How Discord stores trillions of messages? https://discord.com/blog/how-discord-stores-trillions-of-messages, 2023. Accessed: July 23, 2024.

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

[10] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Trans. Storage*, 17(4), oct 2021.

[11] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Trans. Storage*, 17(4), oct 2021.

[12] Zhuohui Duan, Jiabo Yao, Haikun Liu, Xiaofei Liao, Hai Jin, and Yu Zhang. Revisiting log-structured merging for kv stores in hybrid memory systems. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 674–687, New York, NY, USA, 2023. Association for Computing Machinery.

[13] Jason Evans. jemalloc. http://jemalloc.net/, 2018.

[14] Facebook. Benchmarking Tools:dbbench. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools, 2018. Accessed: July 23, 2024.

[15] Facebook. Rocksdb. http://rocksdb.org/, 2018.

[16] Facebook. RocksDB Direct I/O. https://github.com/facebook/rocksdb/wiki/Direct-IO, 2018. Accessed: July 23, 2024.

[17] Facebook. RocksDB Leveled Compaction. https://github.com/facebook/rocksdb/wiki/Leveled-Compaction, 2018. Accessed: July 23, 2024.

[18] Gil Tene. How not to measure tail latency. https://www.infoq.com/presentations/latency-response-time/, 2016. Accessed: July 23, 2024.

[19] Christopher Jermaine, Edward Omiecinski, and Wai Gen Yee. The partitioned exponential file for database storage management. *The VLDB Journal*, 16(4):417–437, oct 2007.

[20] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 993–1005, USA, 2018. USENIX Association.

[21] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, 2016.

[22] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young ri Choi, Alan Sussman, and Beomseok Nam. ListDB: Union of Write-Ahead logs and persistent SkipLists for incremental checkpointing on persistent memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 161–177, Carlsbad, CA, July 2022. USENIX Association.

[23] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

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

[25] Jinglei Ren. Ycsb-c. https://github.com/basicthinker/YCSB-C, 2016.

[26] Russell Sears and Raghu Ramakrishnan. blsm: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 217–228, New York, NY, USA, 2012. ACM.

[27] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: high-throughput generative inference of large language models with a single gpu. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

[28] tigerbeetle. The world's fastest financial accounting database. https://tigerbeetle.com, 2020. Accessed: July 23, 2024.

[29] Wikipedia. NVM discontinuation. https://en.wikipedia.org/wiki/3D_XPoint, 2023. Accessed: July 23, 2024.

[30] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. Matrixkv: Reducing write stalls and write amplification in lsm-tree based kv stores with a matrix container in nvm. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association.

[31] Jinghuan Yu, Sam H. Noh, Young-ri Choi, and Chun Jason Xue. Adoc: Automatically harmonizing dataflow between components in log-structured key-value stores for improved performance. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies*, FAST'23, USA, 2023. USENIX Association.

[32] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. Chameleondb: a key-value store for optane persistent memory. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 194–209, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: attributing the source of tail latency through precise load testing and statistical inference. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 456–468. IEEE Press, 2016.

[34] Yuanhui Zhou, Jian Zhou, Shuning Chen, Peng Xu, Peng Wu, Yanguang Wang, Xian Liu, Ling Zhan, and Jiguang Wan. Calcspar: A Contract-Aware LSM store for cloud storage with low latency spikes. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 451–465, Boston, MA, July 2023. USENIX Association.