SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores

Oana Balmau, Florin Dinu, and Willy Zwaenepoel, *University of Sydney;*
Karan Gupta and Ravishankar Chandhiramoorthi, *Nutanix Inc.;*
Diego Didona, *IBM Research–Zurich*

https://www.usenix.org/conference/atc19/presentation/balmau

This paper is included in the Proceedings of the
2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the
2019 USENIX Annual Technical Conference
is sponsored by USENIX.

SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores

Oana Balmau
University of Sydney

Florin Dinu
University of Sydney

Willy Zwaenepoel
University of Sydney

Karan Gupta
Nutanix Inc.

Ravishankar Chandhiramoorthi
Nutanix Inc.

Diego Didona
IBM Research – Zurich

Abstract

LSM-based KV stores are designed to offer good write performance, by capturing client writes in memory, and only later flushing them to storage. Writes are later compacted into a tree-like data structure on disk to improve read performance and to reduce storage space use. It has been widely documented that compactions severely hamper throughput. Various optimizations have successfully dealt with this problem. These techniques include, among others, rate-limiting flushes and compactions, selecting among compactions for maximum effect, and limiting compactions to the highest level by so-called fragmented LSMs.

In this paper we focus on latencies rather than throughput. We first document the fact that LSM KVs exhibit high tail latencies. The techniques that have been proposed for optimizing throughput do not address this issue, and in fact in some cases exacerbate it. The root cause of these high tail latencies is interference between client writes, flushes and compactions. We then introduce the notion of an I/O scheduler for an LSM-based KV store to reduce this interference. We explore three techniques as part of this I/O scheduler: 1) opportunistically allocating more bandwidth to internal operations during periods of low load, 2) prioritizing flushes and compactions at the lower levels of the tree, and 3) preempting compactions.

SILK is a new open-source KV store that incorporates this notion of an I/O scheduler. SILK is derived from RocksDB, but the concepts can be applied to other LSM-based KV stores. We use both a production workload at Nutanix and synthetic benchmarks to demonstrate that SILK achieves up to two orders of magnitude lower 99th percentile latencies than RocksDB and TRIAD, without any significant negative effects on other performance metrics.

1 Introduction

Latency-critical applications require data platforms that are able to deliver low latency and predictable throughput. Tail latency is especially important, because applications often exhibit high fan-out queries whose overall latency is determined by the response time of the slowest reply. Log-structured merge key-value stores (LSM KVs), such as RocksDB [18], LevelDB [14] and Cassandra [30], are widely adopted in production environments to provide storage beyond main memory for such latency-critical applications, especially for write-heavy workloads. At Nutanix, we use LSM KVs for storing the meta-data of our core enterprise platform, which serves thousands of customers with petabytes of storage capacity.

KV stores support a range of client operations, such as `Get()`, `Update()` and `Scan()`, to store and retrieve data. LSM KVs strive for good update performance by absorbing updates in an in-memory buffer [36, 37]. A tree-like structure is maintained on storage. In addition to client operations, LSM KVs implement two types of *internal operations*: *flushing*, which persists the content of in-memory buffers to disk, and *compaction*, which merges data from the lower into the higher levels of the tree.

In this paper we demonstrate that tail latencies in state-of-the-art LSM KVs can be quite poor, especially under heavy and variable client write loads. We introduce the notion of an *I/O scheduler for LSM KVs*. We implement this I/O scheduler in RocksDB, and we show up to two orders of magnitude improvements in tail latency.

Our work complements much recent work that has sought to improve the client throughput of LSM KVs (e.g., [4, 24, 28, 32, 34, 38, 39, 42, 43]). Client throughput is improved by reducing the cost of internal operations, but this does not suffice to reduce tail latency. Internal operations remain necessary, and client operations that arrive during ongoing internal operations experience increased latency because of interference with these internal tasks. The internal operations may be fewer in number and less costly, reducing the probability of latency spikes, but in practice they occur sufficiently often to influence the higher order percentiles of the latency distribution, especially if client load is bursty.

One may at first think that limiting the I/O bandwidth allocated to internal operations, as is commonly done in production systems, would avoid latency spikes due to interference between internal work and client load. On closer inspection, however, we find this not to be the case. As a simple example, consider a burst of client writes, triggering a burst of flushes. If a number of compactions is going on at the same time, the flushes have to share the limited bandwidth with the compactions, and they become slow. This leads to the in-memory component filling up and blocking further writes, hence producing latency spikes. Limiting the rate of compactions is also insufficient, because they can lead to the lowest level of the tree filling up, stalling flushes, and in turn stalling writes.

These and other observations lead us to the conclusion that reducing the cost of internal operations or limiting their bandwidth allocation does not suffice to avoid latency spikes, and that instead there is a fundamental need for coordination between the client load and the load imposed by different internal operations. To this end we introduce an *I/O scheduler* for an LSM-based KV store.

We build a new KV store, SILK, which we derive from RocksDB. The I/O scheduler in SILK (1) dynamically allocates bandwidth between client and internal operations, (2) gives preference to internal operations that may block client operations, and (3) allows preemption of less critical internal operations. Other techniques could possibly be included, but we have found these sufficient to get two orders of magnitude benefits in tail latency for write-heavy workloads, with no negative effects on throughput or average latency. Also, SILK does not produce significant negative effects in read- and scan-heavy workloads.

Contributions. The main contributions of this paper are:

1. An extensive empirical study that demonstrates the high tail latencies of current LSM KVs.
2. The introduction of an I/O scheduler for LSM KVs, and various scheduling techniques useful for reducing tail latency while maintaining good throughput.
3. An implementation of an LSM KV store I/O scheduler in an industry-standard LSM KV store (RocksDB).
4. An experimental evaluation demonstrating up to two orders of magnitude improvements in tail latency in our production workload, without significant negative effects on other performance metrics or workloads.

2 LSM KV background

2.1 LSM KV architecture

An LSM KV store has three main components: the memory component, the disk component, and the commit log.

Memory component. The memory component C_m is a sorted data structure that resides in main memory. Its size is typically small, around a few tens of MBs. The purpose of C_m is to temporarily absorb user updates. When C_m fills up, it is replaced by a new, empty component. The old memory component is then in the background flushed as is to level 0 (L_0) of the LSM disk component.

Disk component. The disk component C_{disk} is structured into multiple levels (L_0, L_1, . . . L_n), where each level is larger than the previous (lower) level by a configurable factor (e.g., 10). Each level contains multiple sorted files, called SSTables. The number of SSTables on a given level is limited by a configuration parameter, as is the maximum size of an individual file for a given level. SSTables on levels L_i $(i > 0)$ have disjoint key-ranges. L_0 allows overlapping key-ranges between files.

Commit log. The commit log C_{log} stores the updates that are made to C_m (in small batches) on stable storage. C_{log} is usually a few hundreds of MBs large. It is used if the application requires the data to be recoverable in case of a failure, but it is not mandatory. The techniques we propose in SILK apply regardless of whether C_{log} is active or not.

2.2 LSM KV operations

LSM KVs implement two main kinds of operations, which are executed by disjoint thread pools.

Client operations. The main client operations in LSM KVs are writes *(Update(k, v))*, reads *(Get(k))*, and range scans *(Scan(k1, k2))*. *Update(k, v)* associates value v to key k. Updates are absorbed in C_m, to achieve high write throughput. *Get(k)* returns the most recent value of k. The read first goes to C_m. If k is not found in C_m, the read continues to L_0, L_1, . . . L_n, until k is found. By design, at most one SSTable is checked on each level L_i for $i > 0$. On the contrary, more than one SSTable in L_0 may need to be checked because L_0 SSTables may contain the entire key-range. Per-SSTable Bloom filters [7, 25] are used to address this issue. Therefore, in practice, only one SSTable ends up being checked on L_0 most of the time. *Scan(k1, k2)* returns a range of key-value tuples with the keys ranging from $k1$ to $k2$. First, C_m is queried for keys in the $k1$–$k2$ range. Then, SSTables in C_{disk} that may contain the $k1$–$k2$ range are read, going down the levels, until all the keys are found. Client operations are enqueued and served in FIFO order by a fixed-size worker thread pool.

Internal operations. LSM KVs implement *flushing* and *compaction* as background processes. Flushing writes C_m as is to L_0. Because flushing speed affects the rate at which new memory components can be installed, memory components are written to disk without additional processing. As a result, L_0 allows overlapping key-ranges between files. Compaction is the operation that cleans up the LSM tree. It merges SSTables in level L_i of C_{disk} into SSTables with overlapping key-ranges in L_{i+1}, discarding older values in the process. When the size of L_i exceeds its maximum, an SSTable F in L_i is picked and merged into the SSTables in L_{i+1} that have overlapping key-ranges with F, in a way similar to a merge sort. Most LSM KVs support parallel compactions, apart from compactions from L_0 to L_1 which are not parallelized because of overlapping key-ranges on L_0. Compaction induces large I/O overhead by reading the SSTables and writing the new ones to disk.

The system maintains an internal FIFO work queue, where flushes and compactions are enqueued. When a new internal work request is enqueued, it is placed at the end of the internal work queue. A flush is enqueued when C_m fills up. A compaction operation may be enqueued after a flush completes, or after a compaction completes. A pool of internal worker threads serve the requests in the internal work queue. In current LSM KVs an internal operation is enqueued whenever the system deems it necessary in order to maintain the structure of the LSM tree (e.g., when the maximum size or maximum number of files is reached on a level).

2.3 State-of-the-art LSM-based systems

Our experimental study includes three state-of-the-art systems: RocksDB, TRIAD, and PebblesDB.

RocksDB [18] developed at Facebook, is a popular system in production environments, including ours. Its architecture follows the description above. In addition, RocksDB provides a rate limiter to restrict the I/O bandwidth for internal operations. The bandwidth can be set to a fixed value, or RocksDB can change it over time in a multiplicative-increase, multiplicative-decrease manner [19]. This *auto-tuned* version of the rate limiter adapts the bandwidth to the amount of internal work, allocating more bandwidth when there is more pending work.

TRIAD [4] reduces the overhead of internal operations through three techniques. First, TRIAD keeps frequently updated keys in C_m, decreasing internal operation overhead in skewed workloads. Second, TRIAD provides an improvement to the flushing operation, by leveraging data already written in C_{log}. Finally, at the disk level, TRIAD employs a cost-based approach to trigger compaction from L_0 to L_1. Compaction happens only when there is significant key-range overlap, reducing the frequency of compaction operations and amortizing their cost.

PebblesDB [39] avoids most of the compaction overhead of merging and rewriting SSTables, by allowing overlapping key-ranges on all but the highest tree level through the use of Fragmented LSM trees. PebblesDB orders SSTables by key-ranges on each level, and uses special pointers called *guards* to indicate where a given key-range is on a level. When the number of guards on a level reaches a threshold, the guards and the corresponding keys are moved to the next level, mostly without re-writing the SSTables. PebblesDB only requires compaction at the highest tree level, when the number of guards reaches a threshold.

3 Performance requirements for LSM KVs

LSM KVs should meet the following requirements:

1. **Low tail latency.** In environments where LSM KVs serve applications with high fan-out operations, in which the slowest reply within an operation determines the latency of the whole operation, low tail latency is a key requirement [15].

2. **Predictable throughput.** LSM KVs must deliver a throughput that matches the client load *at any time*. Throughput variability is a well-known problem in LSM KVs, mostly stemming from the interference between LSM internal work and client requests.

3. **Small main memory footprint.** Typically, LSM KVs are only one piece of a wider set of services that are accessed by an application. For example, a KV store that handles meta-data can co-exist on the same machine with other services that require large amounts of memory, making memory a constrained resource.

LSM issues. A common issue in LSM KVs is interference between LSM internal work and client operations when a sudden burst of client-side writes occurs in parallel with long-running, resource-intensive compaction tasks. Despite the fact that internal LSM work directly influences client latency, it is handled without being aware of the client load. For instance, large compactions (e.g., compacting tens of GBs) may occupy a large fraction of the I/O bandwidth for extended periods of time (e.g., tens of minutes). The resulting problem is two-fold. First, when flushes do not proceed in a timely manner, the memory component fills up, and incoming writes cannot be absorbed in the memory component. Second, slow L_0 to L_1 compactions lead to the accumulation of a large number of SSTables on L_0. In extreme situations, when the maximum number of SSTables on L_0 is reached, this dynamic brings the entire system to a halt. Both scenarios lead to severe spikes in client latency.

4 Experimental study of tail latency

We perform an extensive experimental study to show that established techniques used in industry and state-of-the-art research systems do not solve the issue of tail latency.

4.1 Experimental environment

We use the YCSB [11] update-intensive workload, corresponding to a 50:50 read:write ratio on 1 KB items (YCSB

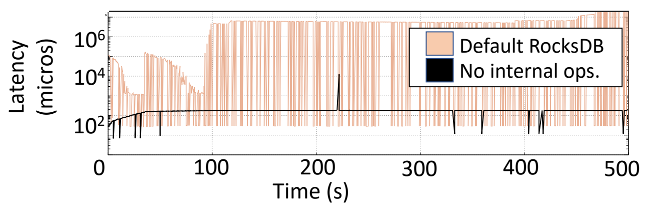


Figure 1: RocksDB compared to a RocksDB version with no internal operations. Internal operations lead to spikes in client request 99th percentile latency.

workload A), with a uniform data distribution. We choose this workload because it is representative for write-intensive workloads in LSM KVs. Furthermore, a uniform workload allows us to detect more quickly performance problems that in skewed workloads would remain hidden due to in-memory caching [4]. LSM KVs are notorious for having numerous tuning knobs. Throughout this study, we configure all the involved systems to the best of our abilities, following guidelines in [23, 39]. The hardware configuration we use in this study is described in Section 6.1.

Each experiment consists of a population phase followed by read and write operations issued at 18 Kops/sec. Unless stated otherwise, all experiments are run without I/O bandwidth rate limiting for internal operations. For RocksDB, unless stated otherwise, we use two memory components of 128MB each. For the rest of the systems we limit the memory use to 1GB. We configure the maximum number of concurrent internal operations to ten for all systems. The latency is measured every second. The 99th percentile latency is computed for every 1-second interval.

4.2 RocksDB

We first show the performance degradation of client operations caused by internal operations in RocksDB. To this end, we compare RocksDB with a modified version of RocksDB in which compaction and flushing are disabled. We disable internal operations by discarding C_m when it fills up (the data store is pre-populated with the full set of keys, so persistent storage is accessed by reads, if necessary).

Figure 1 shows the performance of the two systems over time. The 99th percentile latency of operations in RocksDB is 2 to 4 orders of magnitude higher than in the system without internal operations. These spikes are not present at the 50th and 90th percentiles of the latency distribution. Both reads and writes experience latency spikes at the same time and of the same magnitude, despite their different access paths in the LSM KV store (i.e., writes complete in-memory, while reads are typically served from persistent storage).

The main culprit for the latency spikes is the fact that writes get blocked by virtue of C_m filling up. The reads then get queued behind these writes in the threading architecture.

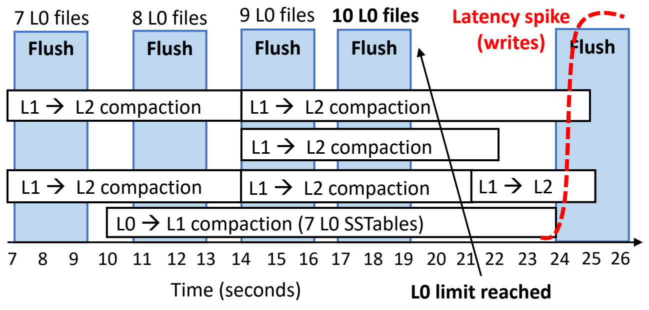


Figure 2: RocksDB. Timeline of internal operations during a writes latency spike (dashed red line) caused by L_0 reaching full capacity. L_0 reaches 10 SSTables at $t = 19$, so flushes are temporarily paused.

We identify two main reasons for write latency spikes, illustrated in Figures 2 and 3. The examples showcase real scenarios encountered while profiling RocksDB.

Figure 2 illustrates an example of a write latency spike (the red dashed line) occurring because L_0 reaches maximum capacity (10 SSTables in this example). Several compactions on levels L_0, L_1 and L_2 occur in parallel between $t = 14$ and $t = 23$. Even if many parallel compactions can run at higher levels (i.e., L_i to L_{i+1}, where $i > 0$), there can only be one L_0 to L_1 compaction running at a time. Since I/O bandwidth is spread equally over all compactions, L_0 to L_1 compaction is slowed down. Consequently, L_0 is not cleared fast enough, which, in turn, causes flushes from C_m to be temporarily halted.

A second cause for latency spikes is C_m filling up because of slow flushing, as illustrated in Figure 3. Here, L_0 does not fill up, reaching only 7 SSTables at $t = 5$. However, the flush starting at $t = 0$ takes an unusually long time (5 seconds compared to 1-2 seconds for a typical flush). The cause is that, by coincidence, a large number of compactions are running at the same time, which makes flushing slow because of limited available I/O bandwidth. There are 7 ongoing compactions at the time of the very slow flush.

4.3 Rate-limited RocksDB

Limiting the I/O bandwidth for internal operations is a popular technique to prevent them from consuming an excessive amount of I/O bandwidth, and hence to "shelter" client operations. We now report the results that we obtain when running RocksDB with limited I/O bandwidth for internal operations [22].

Figure 4 shows the 99th percentile latency of client operations over time when limiting the I/O bandwidth for internal operations to 50, 75 and 90 MB/s. For the sake of legibility, in Figure 4 we show the results of the experiment only up to the time that the tail latency greatly deteriorates (at 900s for 50MB/s, etc). The higher the bandwidth assigned to inter-

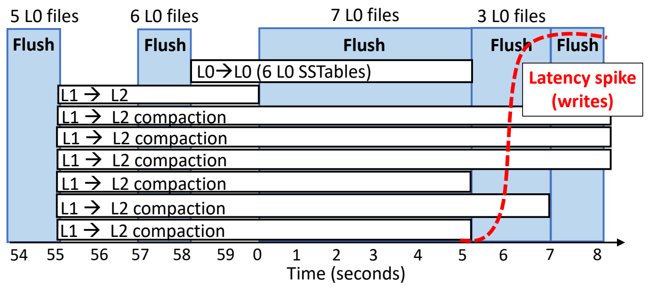


Figure 3: RocksDB. Timeline of internal operations during a writes latency spike caused by slow flushing. From $t = 0$ to $t = 7$, flushes are slowed down by many parallel L_1 to L_2 compactions monopolizing I/O bandwidth. Consequently, C_m fills up, not being able to absorb incoming updates.

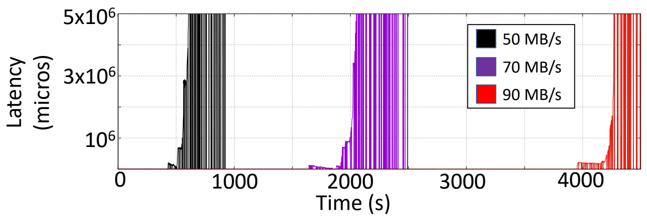


Figure 4: RocksDB. 99th percentile of client request latency when limiting the I/O bandwidth for internal operations.

nal operations, the longer the system is able to postpone the occurrence of latency spikes. However, restricting the bandwidth for internal operations results in slowing them down. This approach therefore increases the likelihood that at some later point many compactions are running at the same time, contending for the limited I/O bandwidth.

4.4 RocksDB with increased C_m

We investigate whether allocating larger memory buffers influences tail latencies in LSM KV. To this end, we run a series of experiments in RocksDB where we increase the total size of the memory component(s) to 1GB – a value close to the upper limit of what we can allow in our production environments (see Section 3). We distribute the memory first into two 500MB memory components and then into ten memory components of 100MB each. We also vary the maximum number of flushes, experimenting with one and ten parallel flushing threads. We find the setup with ten memory components and a single flushing thread to be the most efficient in postponing the latency spikes because the memory absorbs more updates and the data flow from memory to L_0 matches more closely the L_0 to L_1 compaction flow. However, we encounter tail latency spikes sooner or later in all of these cases, for similar reasons to the ones in the scenarios above.

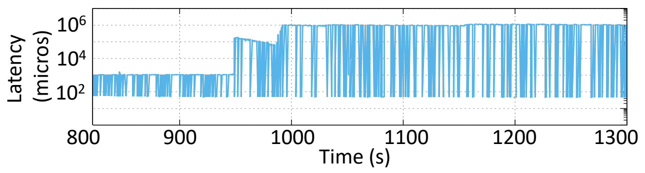


Figure 5: TRIAD. 99th percentile latency. Despite reducing internal operations overhead, TRIAD does not prevent latency spikes during resource-intensive compactions.

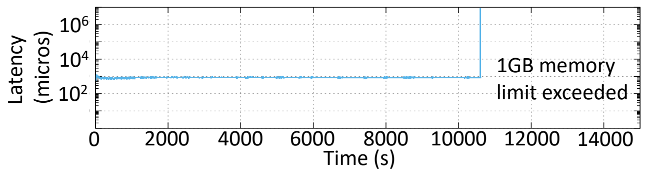


Figure 6: PebblesDB. 99th percentile latency. Postponing compaction keeps the latency low. Experiment ends because of high memory overhead.

4.5 TRIAD

Reducing the overhead of internal operations, as done by state-of-the-art systems [4, 12, 13, 17], is not enough to avoid resource interference. We use TRIAD [4] as a representative of such state-of-the-art systems in the next experiment. Figure 5 shows the 99th percentile latency of client operations over time. In this scenario, TRIAD reduces compaction overhead mainly by choosing when to run L_0 to L_1 compactions depending on key-range overlap. Postponing compactions at the lower levels (closer to C_m) results in postponing compactions at the higher levels. So, in the long term, TRIAD increases the likelihood of running many concurrent compactions. Consequently, the 99th percentile of client operations shows no spikes for the first $\approx 1{,}000$ seconds but, after that point, shows frequent and significant spikes.

4.6 PebblesDB

Figure 6 shows PebblesDB's 99th percentile latency over time. The experiment stops after 10,500 seconds. Although we provide PebblesDB with more memory than RocksDB and TRIAD, it runs out of memory at this time. The memory consumption is due to the frequent creation of guards and Bloom filters in a write-intensive workload. During its uptime, PebblesDB provides very good tail latencies due to the absence of compactions. In other words, the LSM tree is restructured through the use of guards but no data compactions occurred.

To create a situation in which PebblesDB experiences compactions, we run it with a read-intensive workload (95:5) which reduces memory pressure. With this workload, tail latencies remain very good in the early going, but after around

8 hours, when compaction sets in, the system effectively comes to a halt. PebblesDB stalls client operations when it has to perform the very resource-demanding compaction on the highest level of the tree. When such compaction takes place, all threads for internal operations are busy, so they cannot push down guards and keys from the lower levels of the tree. Hence, to maintain the tree integrity, PebblesDB stalls client operations until compaction terminates.

4.7 Lessons learned

We gain three main insights from our experimental study.

Lesson 1) The main reason for high tail latency is the fact that writes get blocked by C_m filling up. There are two principal reasons for this. The first reason is that L_0 on disk is full, which causes flushes from C_m to be halted. L_0 reaches its capacity if L_0 to L_1 compaction cannot keep up. The second reason is that, by coincidence, a large number of compactions are happening concurrently, which causes flushing to be slow because of limited available bandwidth.

Lesson 2) Simply limiting bandwidth for internal operations does not solve the problem of limited bandwidth being available for flushes and can in fact exacerbate it in the long run. This approach effectively postpones compactions, and therefore increases the likelihood that at some later point many compactions occur at the same time.

Lesson 3) Recent approaches to improve throughput, such as being selective about starting compactions or only performing compactions at the highest level, avoid latency spikes in the short run, but aggravate the problem in the long run, because they too increase the likelihood of many concurrent compactions at some later point in time.

As a corollary to Lesson 1, we conclude that not all internal operations are equal. Internal operations on the lower levels of the tree (i.e., closer to C_m) are critical, because failing to complete them in a timely fashion may result in stalling client operations.

Finally, as a corollary to Lessons 2 and 3, it is essential to run performance tests for an extended amount of time, lest these issues go undetected.

5 SILK

5.1 SILK design principles

SILK integrates the lessons we learn from our experimental study into an *I/O scheduler* for internal and external work. SILK follows three core design principles.

1) Opportunistically allocating I/O bandwidth to internal operations. SILK leverages the fact that, in production workloads, the load of client-facing operations typically varies over time (see Figure 7). SILK allocates less I/O bandwidth to compactions on higher levels during peak

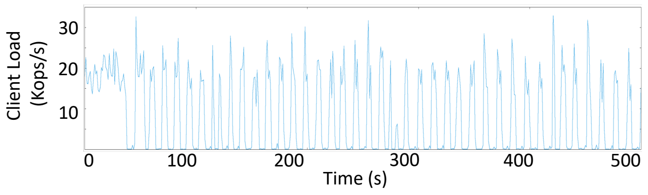


Figure 7: Client load in Nutanix production workload. Real workloads are not flat lines.

client load, and exploits transient low-load periods to boost the processing of internal operations. Dynamic I/O throttling enables SILK (1) to limit interference between internal operations and client-facing ones, and (2) to avoid accumulating over time too large a backlog of internal work, preventing overload conditions in the long term.

2) Prioritizing internal operations at the lower levels of the tree. SILK integrates Lesson 1 in its design by introducing prioritized execution of flushes and compactions from L_0 to L_1. SILK splits internal operations of LSM KVs into three categories with respect to the effect they have on client latencies: (1) SILK ensures that the flushes are fast, making room in memory to absorb incoming updates, which directly affects write latency, (2) SILK gives second priority to L_0 to L_1 compactions, ensuring that L_0 does not reach its full capacity, so that flushes can proceed, (3) SILK gives third priority to compactions on the levels below L_1 because, while they maintain the structure of the LSM tree, their timely execution does not significantly affect client operation latencies in the short term.

3) Preempting compactions. SILK implements a new compaction algorithm that allows internal operations on lower levels of the tree to preempt compactions on higher levels.

5.2 SILK implementation

5.2.1 Opportunistically allocating I/O bandwidth

SILK continuously monitors the bandwidth used by client operations and allocates the available leftover I/O bandwidth to internal operations. The client load monitoring and rate limiting are handled by a separate SILK thread. The monitoring granularity is a system parameter which depends on the frequency of fluctuations in the workload; the monitoring granularity in SILK is currently configured to 10 ms.

If the total I/O bandwidth available to the LSM KV store is T B/s, SILK measures the bandwidth C B/s used by the client requests and it continuously adjusts the internal operation bandwidth to $I = T - C - \varepsilon$ B/s , where ε is a small buffer. To adjust the I/O bandwidth, SILK makes use of a standard rate limiter (e.g., [22]). SILK maintains a minimum configurable I/O bandwidth threshold for flushing and L_0 to L_1 compactions, because these operations directly influence client latency.

To minimize overhead associated with changing the rate limit, SILK only adjusts the limit if the difference between the current value and the new measured value is significant. We empirically set this threshold to be 10 MB/s. We find that lower thresholds cause overly frequent changes in the rate limit. The role of ε is to account for small fluctuations in client load which are not significant enough to adjust internal operation bandwidth using the rate limiter.

5.2.2 Prioritizing and preempting internal operations

Recall that in LSM KVs internal work is handled by a pool of internal worker threads. Once a flush or a compaction is completed, the system checks whether more internal work is needed by assessing the size of the levels and the state of the memory component. If needed, more internal work tasks are scheduled in an internal work queue. SILK maintains two internal worker thread pools: a high-priority one for flushing, and a low-priority one for compactions.

Flushing has the highest priority among the internal operations. Flushes have their dedicated thread pool and always have access to the I/O bandwidth available for internal operations. The minimum flushing bandwidth is chosen to be sufficient to be able to flush the immutable memory component before the active one fills up. The current implementation of SILK allows two memory components (i.e., an immutable one, and an active one) and one flushing thread. If memory constraints allow it, having multiple memory components and flushing threads may help sustain longer client activity peaks.

L_0 to L_1 compaction. SILK needs L_0 to L_1 compactions to progress to ensure that there is enough room to flush on L_0. Unlike flushes, these compactions do not have a dedicated thread pool. If L_0 to L_1 compaction needs to proceed and all the threads in the compaction pool are running higher-level compactions, one of them is *preempted*. This way, L_0 to L_1 compactions do not wait behind higher-level compactions. In the current implementation the preempted compaction task is picked at random.

L_0 to L_1 compaction, like all internal operations is subject to dynamic I/O throttling. However, this type of compaction is never paused, even if SILK may choose to give no bandwidth to compactions. In order to keep L_0 to L_1 compaction running, SILK temporarily moves this job to the high priority thread pool and keeps it running via a high priority thread (i.e., same priority as the flush thread). In this case, the minimum flushing bandwidth mentioned above is shared by the flushing thread and the L_0 to L_1 compaction thread. At most one L_0 to L_1 compaction can run at a time, due to consistency issues caused by overlapping key-ranges. So, only one extra thread is added in the high priority thread pool. Recent versions of RocksDB support L_0 to L_0 compactions as an optimization to quickly reduce the number of SSTables on L_0 [21]. Since this optimization is beneficial for allowing flushes to proceed, SILK treats this case the same as L_0 to L_1 compactions.

Higher-level compactions. Compactions on levels higher than L_1 are scheduled in the low priority compaction thread pool. They make use of the I/O bandwidth available, as indicated by the dynamic rate limiter described in Section 5.2.1. SILK can pause and resume these larger compactions, either individually (because of L_0 to L_1 compaction preemption) or at the level of the thread pool (because of high user load).

It might happen that an L_1 to L_2 compaction is invalidated by the work done by an L_0 to L_1 compaction which preempted it. In this case, SILK discards the partial work done by the higher level compaction. We did not find this wasted work to significantly impact performance.

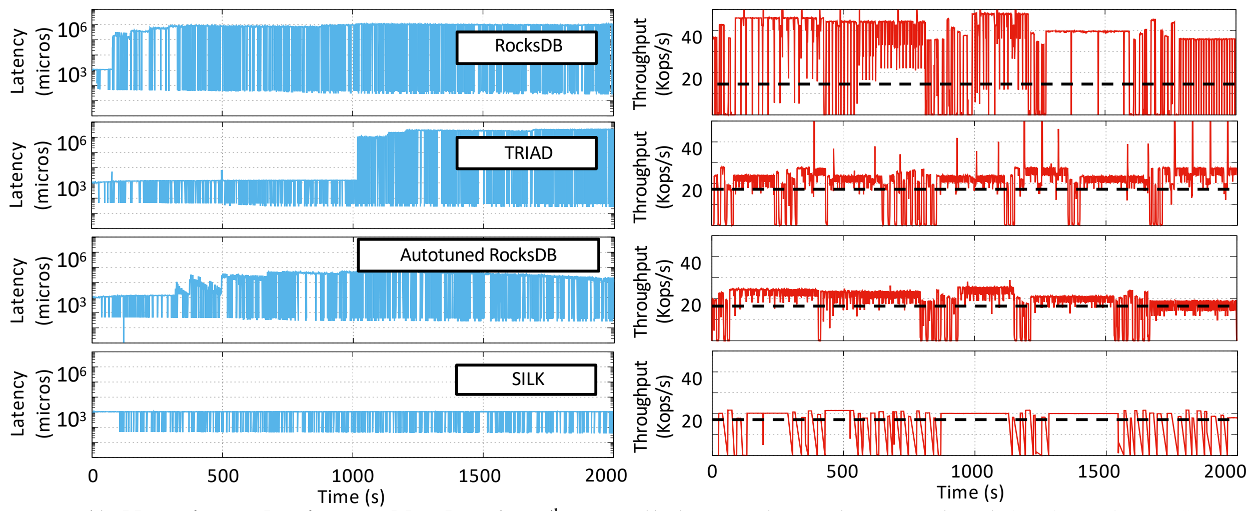
SILK supports parallel compaction, like most current LSM KVs. By default, and assuming equally aggressive compaction threads, each thread gets a similar share of the resources. LSM KVs do not allow parallel compactions from L_0 to L_1, so, if many parallel compactions are allowed, most of the compaction threads are working on the higher levels. This is detrimental to the client operation latencies, since L_0 to L_1 compactions are key to system performance and would benefit from getting the bulk of the resources. Reducing the size of the thread pool, together with SILK's compaction preempting scheme allows each internal worker thread to access a larger share of the resources, which results in faster completion of critical compactions.

The typical recommendation [23] is to set the number of compaction threads equal to the total number of cores. However, we find that the number of compaction threads should instead depend on the total drive I/O bandwidth and the amount of I/O bandwidth required by individual compaction operations. For instance, for a drive with 200MB/s bandwidth, four internal work threads is a suitable choice; even if all the threads happen to run in parallel, they are still allocated a large enough amount of bandwidth to finish the compaction operations fast, thus avoiding scenarios like the ones described in Section 4.

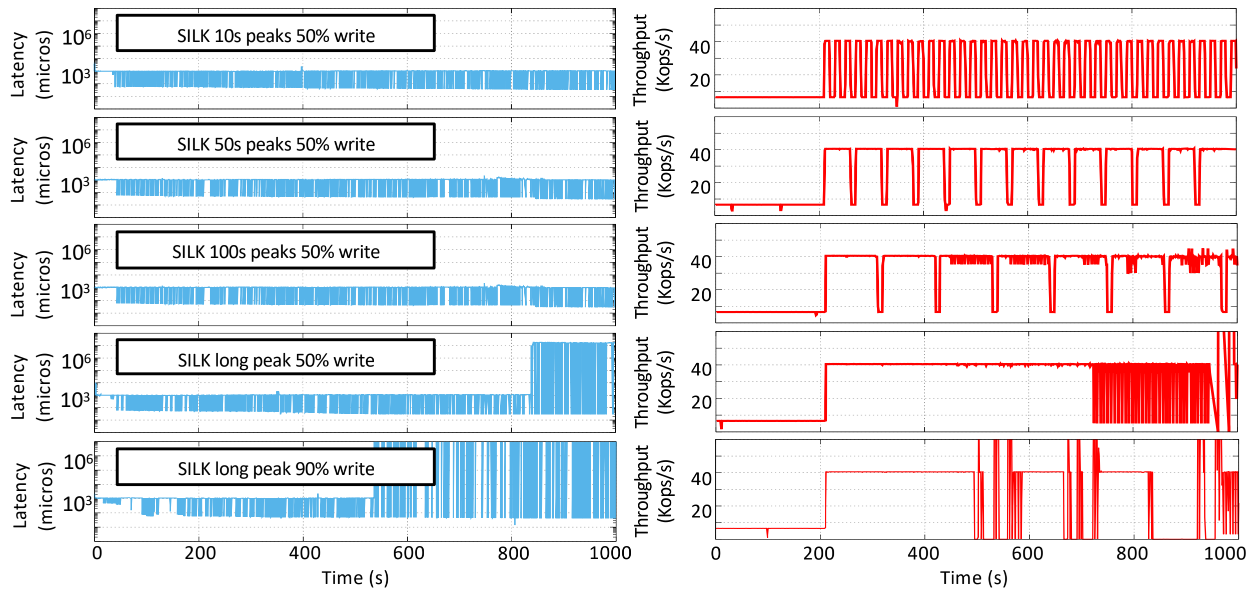
Currently, SILK controls the total I/O bandwidth allocated for compactions in the low priority thread pool. An interesting strategy to explore would be allocating different amounts of bandwidth to compactions at a finer granularity, depending on how urgent the compaction task is considered. We find the current approach to bring good improvements without this additional level of complexity.

6 Evaluation

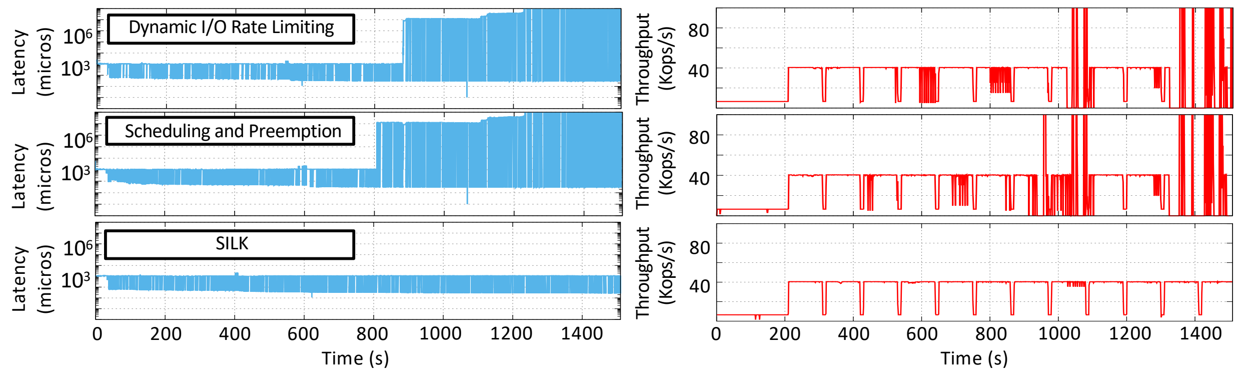
We implement SILK as an extension of RocksDB – used at Nutanix–, and of TRIAD. The source code of SILK is available at `https://github.com/theoanab/SILK-USENIXATC2019`. In what follows, we refer to the



A). Nutanix production workload. Left: 99th percentile latency, log scale on y-axis. Right: throughput. SILK maintains low and steady tail latency and its throughput closely follows the client load. Throughput presents high fluctuations in RocksDB and TRIAD. Average throughput is shown by the dashed black line.



B). Synthetic workloads. Degradation is faster during long peaks, as workloads get more write intensive.



C). Breakdown of SILK techniques. 50% write - 50% read workload, 100s peaks.
SILK's techniques complement each other in order to maintain low tail latency in the long run.

Figure 8: SILK performance in production and synthetic workloads.

RocksDB and the TRIAD extensions as RocksDB-SILK and TRIAD-SILK, respectively. An I/O scheduler could also be applied to PebblesDB, with suitable modifications for the fact that compactions only happen at the highest level. We do not extend PebblesDB with an I/O scheduler because of lack of familiarity with the code base and because PebblesDB's memory demands are not suitable in our environment.

We evaluate SILK with production and synthetic workloads, focusing on write-intensive workloads. We compare against TRIAD and RocksDB and show that:

• SILK achieves up to 2 orders of magnitude lower tail latency than state-of-the-art systems (Section 6.2).

• SILK's performance does not deteriorate over time in long running production workloads (Section 6.2).

• SILK provides stable throughput, close to the client load (Sections 6.2 and 6.4).

• SILK does not create any significant negative side effects on other important metrics such as average latency and read performance (Section 6.3).

• SILK can sustain long client activity peaks interrupted by short client activity lows (Section 6.4).

• The techniques used in SILK contribute to the results above in complementary ways (Section 6.5).

6.1 Experimental setup

Hardware. We perform the evaluation on a 20-core Intel Xeon, with two 10-core 2.8 GHz processors, 256 GB of RAM, and 960GB SSD Samsung 843T. All systems were restricted to run with 1GB of RAM using Linux control groups.

Benchmark. We compare the performance of RocksDB-SILK and TRIAD-SILK to RocksDB, TRIAD, and a version of RocksDB that uses the auto-tuned rate limiter [19]. All experiments are run through db_bench, one of RocksDB's standard benchmarking tools [20].

Measurements. Load-generator threads issue requests in an open loop according to the workload characteristics. They deposit the requests in the queues of the KV store worker threads. Latency is measured on the side of the load-generator threads, capturing both *queuing* time and *processing* time. We measure the 99th percentile tail latency and the throughput over one-second intervals (i.e., not cumulative over the entire experiment run). We report throughput and latency every second in the time-series plots.

Dataset. The dataset size for both the production and the synthetic workloads is approximately 500GB. The KV-tuple sizes vary between the production and the synthetic workloads. In all the experiments the data store is pre-populated with the entire dataset.

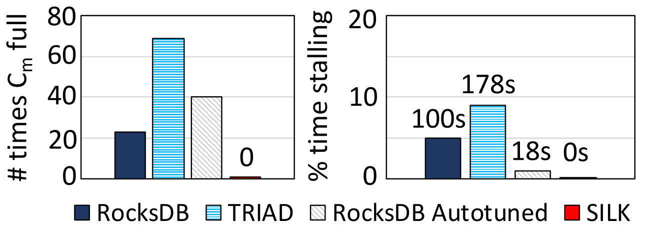


Figure 9: Production workload. Number of times C_m cannot be flushed (left) and time spent stalling writes (right).

KV store configuration. We use a 128MB memory component size and two memory components (i.e., one active and one immutable). In SILK, flushing and L_0 to L_1 compactions proceed at a rate of 50MB/s if SILK paused the other internal operations. The total I/O bandwidth allocated to the LSM KV store is 200MB/s. The level0-slowdown and level0-stop parameters (used to slow down or stop client writes once a maximum number of files is reached on L_0) are configured to very large values in all data stores so as not to artificially interfere with the measured latency. We use a thread pool of 4 threads for internal operations (including the flushing thread) for all the systems. All systems are pinned to 16 cores, out of which 8 are used by the worker threads, and 8 are used by the internal operations and other LSM threads (e.g., monitoring the client load in SILK). The load-generator threads run on separate dedicated cores.

Compression and commit logging are disabled in all reported measurements. While enabling them affects the absolute performance results, it does not impact the conclusions of our evaluation: the performance differences between RocksDB-SILK or TRIAD-SILK, on the one hand, and standalone RocksDB and TRIAD, on the other hand, remain similar. Using compression is equivalent to working with a smaller dataset. Commit logging takes the same amount of bandwidth in all systems. Therefore, from an experimental perspective, using C_{log} is roughly equivalent to working on a machine with smaller disk bandwidth.

6.2 Nutanix workload

Workload description. We sample one of our production workloads at Nutanix over 24h. It is a write-dominated workload, with a 57:41:2 write:read:scan ratio (a scan length is in the order of tens of keys). The client requests arrive in bursts (peaks of around 20K requests/s), separated by periods of low activity (valleys of around hundreds of requests/s or less). A typical duration of a valley is between 5s and 20s, with an average valley length being approximately 15s. Most peaks (approximately 90%) are short bursts between 10s and 20s. The longer peaks (>100s) make up the rest of the workload. The maximum peak lasts approximately 400s. The request sizes range between 250B and 1KB for

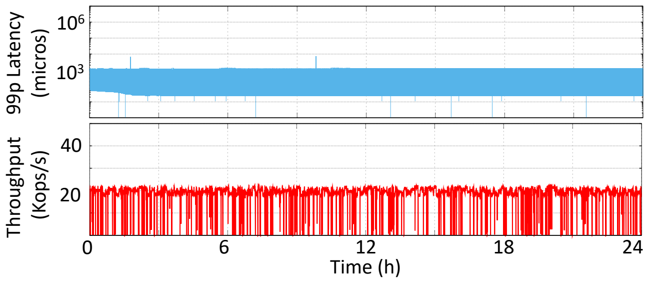


Figure 10: SILK in 24h Nutanix production workload. Top: 99th percentile latency, log scale on y-axis. Bottom: throughput. SILK maintains stable low latency and throughput close to the client load for extended time spans.

the single-point operations (i.e., reads and writes), with a median of 400B. We use a trace replay of the original workload, providing the input at the same rate as the original trace.

Results. Figure 8A shows the 99th percentile latency (left) and throughput (right) for RocksDB-SILK (bottom row), compared to state-of-the-art systems. Results obtained with TRIAD-SILK are similar.

SILK obtains two orders of magnitude lower tail latency than the auto-tuned RocksDB, and three orders of magnitude better than RocksDB and TRIAD, due to its combination of adjusting the I/O bandwidth and better internal work scheduling. Similar to their behavior described in Section 4, the tail latencies in RocksDB and TRIAD exhibit frequent spikes, due to stalling and contention for I/O bandwidth. The auto-tuned rate limiter in RocksDB achieves one order of magnitude better tail latency than both TRIAD and RocksDB, but does not avoid interference on shared resources as effectively as SILK (see Figure 8A, third row). The auto-tuner simply increases I/O bandwidth when there is more internal work to do, and it is oblivious of user load.

Throughput in SILK stays close to the offered client load, while throughput in RocksDB presents high fluctuations. Client operations build up in the worker thread queues because of interference with internal operations. When they can proceed, they are processed in bursts, which results in throughput spikes. TRIAD and the auto-tuned RocksDB stay closer to the offered client load, but still present throughput fluctuations, correlated to increases in tail latency.

Figure 9 shows the number of times C_m cannot be flushed right away (left) and the amount of time the writes are stalled, relative to the duration of the experiment (right). The statistics are collected for the experiment shown in Figure 8A. SILK never stalls writes and can always flush C_m as soon as it fills up. TRIAD has the most problems flushing C_m on time – the flush is delayed 69 times – because of its $L_0 - L_1$compaction strategy. The auto-tuned version of RocksDB does better, but it still spends a significant amount of the time stalling writes, consisting of 1% of the total experiment time.

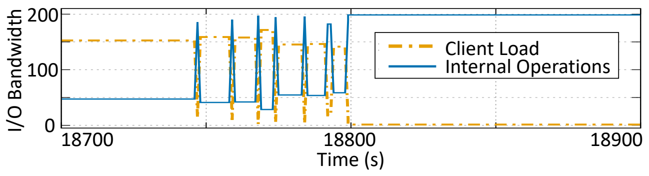


Figure 11: Detail of RocksDB-SILK I/O bandwidth allocation. SILK boosts internal work when client load decreases.

24h production workload. Figure 10 presents the 99th percentile latency and throughput time series of RocksDB-SILK for a 24h run of our production workload. SILK maintains stable performance over the extended period of time. Figure 11 shows a detail of the I/O bandwidth allocation in RocksDB-SILK during 200s of the production workload. Internal work may be temporarily postponed, but is eventually completed in the long term. RocksDB-SILK compacts approximately 3TB of data over the 24h and never has more than three compaction operations waiting to be scheduled. The worker threads experience no write stalls and have less than three operations enqueued throughout the experiment.

6.3 YCSB benchmarks

To evaluate the performance of SILK for a wide range of workloads, we now present results with the the full YCSB benchmark. From this point forward, we show results obtained with TRIAD-SILK. We report that the results for RocksDB-SILK are similar.

Workload description. YCSB provides six core workloads, described in Table 1. We use 8B keys and 1024B values. We evaluate SILK in the zipfian and uniform key distributions and show that SILK reduces tail latency in write-dominated workloads without inducing significant performance degradation in other scenarios.

Results. Figure 12 shows the average throughput of TRIAD and TRIAD-SILK, for the uniform (top) and zipfian (bottom) key distributions. SILK has low impact on throughput for both key distributions, amounting to at most 7%. As expected, SILK incurs the highest overhead for the uniform key distribution for the write-dominated workloads (i.e., YCSB F), because it entails frequent compactions and therefore frequent scheduling interventions by SILK. Reads do not suffer significantly compared to the baseline because of L0 read optimizations in TRIAD (and RocksDB): (1) there is a Bloom filter for each L0 file, and (2) reads return after the KV tuple is first found on L0, without checking all the L0 files. Because of these two optimizations, most of the time only one L0 file is read. So, even if SILK postpones L0 – L1 compactions, it has little impact on read performance. With a zipfian key distribution, most requests are served from memory in the entire benchmark, leading to less compaction. Here, SILK's overhead is at most 4%. Similarly, read-dominated workloads (YCSB B, C, D and E) are less impacted by com-

Workload	Description
YCSB A	write-intensive: 50% updates, 50% reads
YCSB B	read-intensive: 5% updates, 95% reads
YCSB C	read-only: 100% reads
YCSB D	read-latest: 5% updates, 95% reads
YCSB E	scan-intensive: 5% updates, 95% scans; average scan length 50 elements
YCSB F	50% read-modify-write, 50% reads

Table 1: YCSB core workloads description.

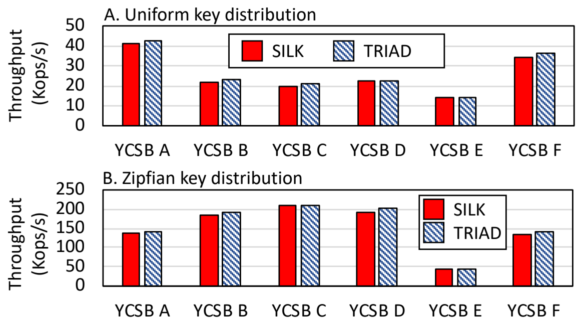


Figure 12: Average throughput of TRIAD and TRIAD-SILK in YCSB. Using SILK has minimal impact on throughput in read- and write- dominated workloads.

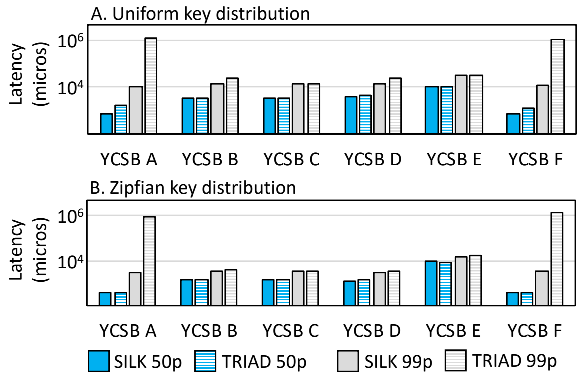


Figure 13: Latency of TRIAD and TRIAD-SILK in YCSB. Log-scale on Y-axis. SILK decreases 99p latency by two orders of magnitude in write-dominated workloads, while maintaining similar median latency across all workloads.

pactions, leading to less overhead (at most 5% in the uniform key distribution).

Figure 13 shows the median and 99 percentile latency for TRIAD and TRIAD-SILK. Generally, SILK's median latency is on par with that of TRIAD, or slightly lower. The only workload where SILK experiences higher median latency than TRIAD is YCSB E with a zipfian key distribution, where SILK surpasses TRIAD by 5%. Tail latency is lower with SILK across all workloads, by at least 5% (in YCSB E). SILK's benefits in terms of tail latency are most pronounced in write-dominated workloads, where the latency is decreased by up to two orders of magnitude.

6.4 Stress testing for long peaks

Workload description. In this section we focus on workloads in the style of YCSB core workload A (see Table 1), where we vary the ratio between the length of the client load peaks and valleys (gradually increasing peak duration, while keeping valley duration constant). We use 8B keys, 1024B values and a uniform key distribution. Client load during low activity periods is approximately 10 Koperations/second and approximately 40 Koperations/s during peaks. The offered load is higher for both the peaks and the valleys than our production workload, in order to stress the system.

Results. Figure 8B shows the 99^{th} percentile latency (left) and the throughput (right) of TRIAD-SILK. The first three rows show a 50:50 write:read workload, where the ratio of peak:valley length is varied: peaks last 10, 50, and 100 seconds, while the valleys last 10 seconds. SILK easily sustains these peaks and valleys in the client load, keeping tail latency low and the throughput steady.

On the last two rows of the figure we show the results of an experiment with a long peak, to see at what point SILK's performance starts to degrade. The fourth row shows the results with a 50:50 write:read workload and the last row with a 90:10 write:read workload. Despite the prioritization of critical internal work, if the peak load is high and the peak duration long, the system cannot allocate enough resources to the internal work, and the performance eventually starts to degrade. Also, as expected, the proportion of writes influences the amount of time the peaks can be sustained. SILK's performance starts to degrade at around 500 seconds (300 seconds of peak) for the 90% writes workload, while the peak can be sustained up to around 700 seconds (500 seconds of peak) for the 50% writes scenario. Despite showing performance degradation, SILK is able to handle challenging workloads that are representative of real applications. Our production workload has at most 400s peaks, 50% writes and load peaks reaching only half the load of the synthetic workload peaks (Figure 10).

6.5 Breakdown

Figure 8C shows 99th percentile latency (left) and throughput (right) for the following variants of SILK. The first row shows a version where we enable SILK's dynamic I/O bandwidth rate limiting, but where the priorities and preemption are disabled. The second row shows the complementary version which uses priorities and preemption but where the I/O bandwidth is not controlled. The final row shows SILK. On their own, neither of the two techniques is able to sustain the client load.

In the first case (top row), the dynamic bandwidth allocation ensures that internal and external work interference is low. However, as the experiment progresses and larger compactions need to take place, the urgent internal operations are slowed down.

In the second case (middle row), good prioritization maintains the tree structure at the levels close to the memory component, allowing flushes and $L_0 - L_1$ compactions to proceed without slowdowns. However, as larger compactions need to take place, the fact that the bandwidth is not controlled leads to negative interference between internal and external work.

7 Related work

Reducing compaction overhead. Many systems reduce the overhead of compaction algorithms used in production systems, such as RocksDB [18], LevelDB [14] and Cassandra [30]. WiscKey [32], HashKV [10] and LWC-tree [44] separate keys from values, and only store keys in the LSM tree, reducing data movement in compaction operations. TRIAD [4] keeps hot data in memory, avoids duplicate writes of the log component, and compacts an SSTable only when there is sufficient overlap with lower-level SSTables. SlimDB [40] allows overlapping key-ranges on each level of the LSM tree to reduce the amount of data that is rewritten, and uses new index blocks and cuckoo filters to perform fast key lookup. Monkey [12], Dostoevsky [13], Lim et al. [31], Dong et al. [17] tune the parameters of the LSM tree, in order to limit the amount of maintenance work and to achieve better performance. Accordion [8] optimizes the layout of the in-memory component by means of a hierarchical structure and in-memory compactions. SifrDB [33] employs different compaction algorithms on different levels of the LSM tree.

These techniques decrease the amount of work performed during internal operations, with the result of increasing throughput. However, they do not avoid the interference with user operations *while* internal operations execute. By contrast, SILK schedules internal operations so as to avoid interference on user operations, thus avoiding latency spikes for user operations and improving tail latencies. The techniques of SILK can be applied to existing designs, thereby preserving their improvements in terms of internal operation overhead. We show this by applying SILK techniques on top of two systems: RocksDB and TRIAD.

Compaction variants and alternatives. PebblesDB [39] allows overlapping key ranges in the lower LSM tree levels to avoid SSTable merges, and uses a skip-list-like structure to allow efficient key lookups. PebblesDB achieves remarkable performance, but its last-level compaction may lead to prolonged service unavailability (Section 4). SILK techniques can be applied to PebblesDB to improve its robustness.

Tucana [38] and ForestDB [2] use variants of the B-ε tree [6, 9, 27] and of the B+ tree, respectively. Unlike LSM trees, these systems do not maintain large sorted files and hence do not implement flushing and compaction. However, they implement operations such as leaf splitting, leaf merging and tree re-balancing to preserve the structure of the tree. These operations result in random accesses that increase write amplification and affect the latency of user operations by contending for I/O. SILK targets LSM tree-based systems, which favor sequential I/O, absorb writes in memory, and leverage sorted SSTables for efficient range scans.

Ahmad and Kemme [1] offload compaction to a dedicated server. Atlas [29] uses different servers to store keys and values. Using a different server for compactions is an effective way to address latency spikes, since this approach removes interference between client and internal operations. However, this solution substantially changes the architecture of the KV store from a standalone system to a distributed one, which results in higher operational costs and increased complexity. Moreover, the transfer of SSTables between the compaction server and the client-facing servers puts additional load on the network, which can generate interference on co-located applications.

Data structure and algorithm improvements. FloDB [5], cLSM [24], HyperLevelDB [26], Nibble [35] and BespoKV [3] improve scalability by alleviating contention bottlenecks. NoveLSM [28] reduces logging overhead by supporting in-place updates to a component stored on NVM, and performs parallel reads. These techniques are orthogonal to SILK's. Minos [16] reduces tail latency for in-memory KVs focusing on workloads with heterogeneous item sizes. To this end, Minos serves similar-sized requests on the same cores. Similar approaches could be implemented in LSM KVs to further reduce tail latency in heterogeneous workloads, and they can co-exist with SILK. bLSM [41] aims to avoid stalling at a level L of the tree by ensuring that operations at lower levels have completed by the time level L has to push data to lower levels. bLSM achieves this goal by throttling internal operation rates. bLSM, however, may throttle user writes as the memory component fills up. Instead of artificially throttling requests, SILK performs internal operations during off-peak periods, and prioritizes higher-level internal operations to avoid stalling user operations.

8 Conclusion

In this paper we presented SILK, a new LSM KV store designed to prevent client request latency spikes. SILK uses an I/O scheduler to manage external client load and internal LSM maintenance work. We implemented SILK in two state-of-the-art LSM KVs and demonstrated order-of-magnitude improvements in latency at the 99^{th} percentile in synthetic and production workloads from Nutanix.

Acknowledgements. We would like to thank our shepherd, Vijay Chidambaram, and the anonymous reviewers for all their helpful comments and suggestions. This work was supported in part by the Swiss National Science Foundation through grant No. 513954, an EcoCloud Postdoctoral Fellowship, and by a gift from Nutanix, Inc. Part of the work has been done while Oana Balmau was an intern at Nutanix.

References

[1] AHMAD, M. Y., AND KEMME, B. Compaction Management in Distributed Key-value Datastores. In *Proceedings of VLDB* (2015).

[2] AHN, J., SEO, C., MAYURAM, R., YASEEN, R., KIM, J., AND MAENG, S. ForestDB: A Fast Key-value Storage System for Variable-length String Keys. *IEEE Transactions on Computers 65*, 3.

[3] ANWAR, A., CHENG, Y., HUANG, H., HAN, J., SIM, H., LEE, D., DOUGLIS, F., AND BUTT, A. R. BespoKV: Application Tailored Scale-out Key-value Stores. In *Proceedings of SC18* (2018).

[4] BALMAU, O., DIDONA, D., GUERRAOUI, R., ZWAENEPOEL, W., YUAN, H., ARORA, A., GUPTA, K., AND KONKA, P. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-value Stores. In *Proceedings of USENIX ATC* (2017).

[5] BALMAU, O., GUERRAOUI, R., TRIGONAKIS, V., AND ZABLOTCHI, I. FloDB: Unlocking Memory in Persistent Key-value Stores. In *Proceedings of EuroSys* (2017).

[6] BENDER, M. A., FARACH-COLTON, M., JANNEN, W., JOHNSON, R., KUSZMAUL, B. C., PORTER, D. E., YUAN, J., AND ZHAN, Y. An Introduction to Bε-trees and Write-optimization. *;login: 40*, 5 (2015).

[7] BLOOM, B. H. Space/time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM 13*, 7 (1970).

[8] BORTNIKOV, E., BRAGINSKY, A., HILLEL, E., KEIDAR, I., AND SHEFFI, G. Accordion: Better Memory Organization for LSM Key-value Stores. In *Proceedings of VLDB* (2018).

[9] BRODAL, G. S., AND FAGERBERG, R. Lower Bounds for External Memory Dictionaries. In *Proceedings of SODA* (2003).

[10] CHAN, H. H. W., LI, Y., LEE, P. P. C., AND XU, Y. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *Proceedings of USENIX ATC* (2018).

[11] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of SoCC* (2010).

[12] DAYAN, N., ATHANASSOULIS, M., AND IDREOS, S. Monkey: Optimal Navigable Key-value Store. In *Proceedings of SIGMOD* (2017).

[13] DAYAN, N., AND IDREOS, S. Dostoevsky: Better Space-time Trade-offs for LSM-tree Based Key-value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of SIGMOD* (2018).

[14] DEAN, J., AND GHEMAWAT, S. LevelDB. `https://github.com/google/leveldb`. visited Jan 2019.

[15] DELIMITROU, C., AND KOZYRAKIS, C. Amdahl's Law for Tail Latency. *Communications of the ACM 61*, 8 (2018).

[16] DIDONA, D., AND ZWAENEPOEL, W. Size-aware Sharding for Improving Tail Latencies in In-memory Key-value Stores. In *Proceedings of NSDI* (2019).

[17] DONG, S., CALLAGHAN, M., GALANIS, L., BORTHAKUR, D., SAVOR, T., AND STRUM, M. Optimizing Space Amplification in RocksDB. In *Proceedings of CIDR* (2017).

[18] FACEBOOK. RocksDB: A Persistent Key-value Store for Fast Storage Environments. `https://rocksdb.org`. visited Jan 2019.

[19] FACEBOOK. RocksDB Autotuned Rate Limiter. `https://rocksdb.org/blog/2017/12/18/17-auto-tuned-rate-limiter.html`. visited Jan 2019.

[20] FACEBOOK. RocksDB Benchmarking Tools. `https://github.com/facebook/rocksdb/wiki/Benchmarking-tools`. visited Jan 2019.

[21] FACEBOOK. RocksDB Level-based Compaction Changes. `https://rocksdb.org/blog/2017/06/26/17-level-based-changes.html`. visited Jan 2019.

[22] FACEBOOK. RocksDB Rate Limiter. `https://github.com/facebook/rocksdb/wiki/Rate-Limiter`. visited Jan 2019.

[23] FACEBOOK. RocksDB Tuing Guide. `https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide`. visited Jan 2019.

[24] GOLAN-GUETA, G., BORTNIKOV, E., HILLEL, E., AND KEIDAR, I. Scaling Concurrent Log-structured Data Stores. In *Proceedings of EuroSys* (2015).

[25] HUA, Y., XIAO, B., VEERAVALLI, B., AND FENG, D. Locality-sensitive Bloom Filter for Approximate Membership Query. *IEEE Transactions on Computers 61*, 6 (2012).

[26] HYPERDEX. HyperLevelDB. `https://github.com/rescrv/HyperLevelDB`. visited Jan 2019.

[27] Jannen, W., Yuan, J., Zhan, Y., Akshintala, A., Esmet, J., Jiao, Y., Mittal, A., Pandey, P., Reddy, P., Walsh, L., Bender, M., Farach-Colton, M., Johnson, R., Kuszmaul, B. C., and Porter, D. E. BetrFS: Write-optimization in a Kernel File System. *ACM Transactions on Storage (TOS) 11*, 4 (2015).

[28] Kannan, S., Bhat, N., Gavrilovska, A., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of USENIX ATC* (2018).

[29] Lai, C., Jiang, S., Yang, L., Lin, S., Sun, G., Hou, Z., Cui, C., and Cong, J. Atlas: Baidu's Key-value Storage System for Cloud Data. In *Proceedings of MSST* (2015).

[30] Lakshman, A., and Malik, P. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review 44*, 2 (Apr. 2010).

[31] Lim, H., Andersen, D. G., and Kaminsky, M. Towards Accurate and Fast Evaluation of Multi-stage Log-structured Designs. In *Proceedings of FAST* (2016).

[32] Lu, L., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of FAST* (2016).

[33] Mei, F., Cao, Q., Jiang, H., and Li, J. SifrDB: A Unified Solution for Write-optimized Key-value Stores in Large Datacenter. In *Proceedings of SoCC* (2018).

[34] Mei, F., Cao, Q., Jiang, H., and Tintri, L. T. LSM-tree Managed Storage for Large-scale Key-value Store. In *Proceedings of SoCC* (2017).

[35] Merritt, A., Gavrilovska, A., Chen, Y., and Milojicic, D. Concurrent Log-structured Memory for Many-core Key-value Stores. In *Proceedings of VLDB* (2017).

[36] O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E. The Log-Structured Merge-tree (LSM-tree). *Acta Inf. 33*, 4 (1996).

[37] Ousterhout, J., and Douglis, F. Beating the I/O Bottleneck: A Case for Log-structured File Systems. *ACM SIGOPS Operating Systems Review 23*, 1 (1989).

[38] Papagiannis, A., Saloustros, G., González-Férez, P., and Bilas, A. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. In *Proceedings of USENIX ATC* (2016).

[39] Raju, P., Kadekodi, R., Chidambaram, V., and Abraham, I. PebblesDB: Building Key-value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of SOSP* (2017).

[40] Ren, K., Zheng, Q., Arulraj, J., and Gibson, G. SlimDB: A Space-efficient Key-value Storage Engine for Semi-sorted Data. In *Proceedings of VLDB* (2017).

[41] Sears, R., and Ramakrishnan, R. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of SIGMOD* (2012).

[42] Wang, P., Sun, G., Jiang, S., Ouyang, J., Lin, S., Zhang, C., and Cong, J. An Efficient Design and Implementation of LSM-tree Based Key-value Store on Open-channel SSD. In *Proceedings of EuroSys* (2014).

[43] Wu, X., Xu, Y., Shao, Z., and Jiang, S. LSM-trie: An LSM-tree-based Ultra-large Key-value Store for Small Data. In *Proceedings of USENIX ATC* (2015).

[44] Yao, T., Wan, J., Huang, P., He, X., Wu, F., and Xie, C. Building Efficient Key-Value Stores via a Lightweight Compaction Tree. *ACM Transactions on Storage (TOS) 13*, 4 (2017).