ADOC: Automatically Harmonizing Dataflow Between Components in Log-Structured Key-Value Stores for Improved Performance

Jinghuan Yu, *City University of Hong Kong;* Sam H. Noh, *UNIST & Virginia Tech;*
Young-ri Choi, *UNIST;* Chun Jason Xue, *City University of Hong Kong*

https://www.usenix.org/conference/fast23/presentation/yu

This paper is included in the Proceedings of the 21st USENIX Conference on File and Storage Technologies.

February 21–23, 2023 • Santa Clara, CA, USA

978-1-939133-32-8

Open access to the Proceedings of the 21st USENIX Conference on File and Storage Technologies is sponsored by

ADOC: Automatically Harmonizing Dataflow Between Components in Log-Structured Key-Value Stores for Improved Performance

Jinghuan Yu[1] Sam H. Noh[2,3*] Young-ri Choi[2] Chun Jason Xue[1]

[1]City University of Hong Kong [2]UNIST [3]Virginia Tech

Abstract

Log-Structure Merge-tree (LSM) based Key-Value (KV) systems are widely deployed. A widely acknowledged problem with LSM-KVs is write stalls, which refers to sudden performance drops under heavy write pressure. Prior studies have attributed write stalls to a particular cause such as a resource shortage or a scheduling issue. In this paper, we conduct a systematic study on the causes of write stalls by evaluating RocksDB with a variety of storage devices and show that the conclusions that focus on the individual aspects, though valid, are not generally applicable. Through a thorough review and further experiments with RocksDB, we show that data overflow, which refers to the rapid expansion of one or more components in an LSM-KV system due to a surge in data flow into one of the components, is able to explain the formation of write stalls. We contend that by balancing and harmonizing data flow among components, we will be able to reduce data overflow and thus, write stalls. As evidence, we propose a tuning framework called ADOC (Automatic Data Overflow Control) that automatically adjusts the system configurations, specifically, the number of threads and the batch size, to minimize data overflow in RocksDB. Our extensive experimental evaluations with RocksDB show that ADOC reduces the duration of write stalls by as much as 87.9% and improves performance by as much as 322.8% compared with the auto-tuned RocksDB. Compared to the manually optimized state-of-the-art SILK, ADOC achieves up to 66% higher throughput for the synthetic write-intensive workload that we used, while achieving comparable performance for the real-world YCSB workloads. However, SILK has to use over 20% more DRAM on average.

1 Introduction

LSM (Log-Structure Merge-tree based)-KV systems buffer their random updates in a memory batch to leverage the disk's high sequential write performance characteristic to support write-intensive workloads. These systems use background data movements to persist cached data, trim the redundant entries, and reshape the storage components to ensure IO performance. LSM-KVs are used in products such as NoSQL storage systems [17,25,41], data warehouses [52], time-series databases [32] and embedded storage engine in RDBMS [15, 48]. Although LSM-KVs can provide higher write throughput, they frequently encounter the write stall phenomenon when facing high write pressure workloads [16,46].

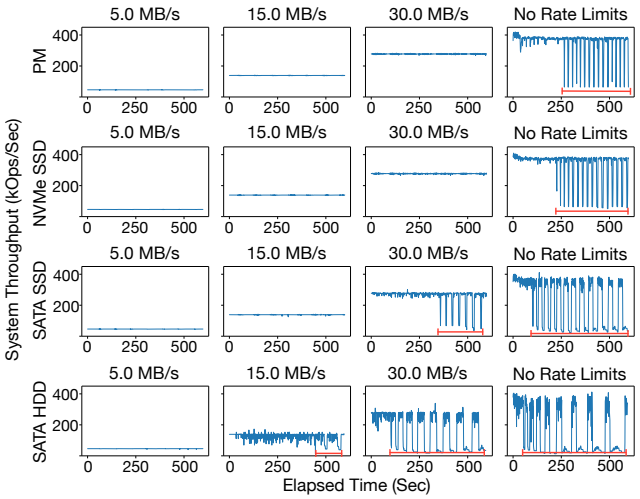


Figure 1: The figures show the throughput while running *fillrandom* with increasing write pressure (represented as numbers on top of each graph) and for different storage devices. As writing pressure increases, system throughput on all devices shows patterns of sudden performance drop and even maintains long-term stalling states. Such significant performance drops are referred to as the write stall phenomenon.

The write stall phenomenon refers to the sudden drop in system throughput as marked by the red lines in Figure 1, which shows results for RocksDB running with write-intensive workloads on a wide range of storage devices, from a traditional HDD to a modern state-of-the-art Intel Optane DC PMM persistent memory device (denoted PM). Based on this figure, we

*This work was done while at UNIST.

identify two characteristics of the write stall phenomena: first, write stalls are *universal*, that is, they occur on all types of devices, though they may be triggered under different conditions, and second, write stalls are strongly *device dependent*, with their duration and rate of performance degradation influenced by various factors such as device type and write intensity. As a notable shortcoming of LSM-KV systems, the write stall phenomenon has been the subject of extensive research and attention in recent years [7, 43, 45, 46, 50, 58, 60].

This study presents ADOC (**A**utomatic **D**ata **O**verflow **C**ontrol), a framework with a goal of minimizing write stalls by harmonizing the flow of data between LSM-KV components. To this end, we first perform an extensive experimental study to analyze the occurrence pattern of write stalls. We find that previous studies [7, 43, 45, 46, 50, 58, 60] are conducted with particular settings, which make their analysis difficult to generalize. Many former studies conclude the cause of write stalls to be resource depletion, while we find that write stalls are triggered even when the hardware resources are sufficient. This indicates that write stalls not only happen in passive blocking situations, but also occur when the system proactively stalls the input stream in attempts to avoid further performance loss. We find that popular LSM-KVs like LevelDB and RocksDB already use active stalling strategies to avoid "Disk Overflow", which refers to the situation where flush or compaction jobs cannot keep up with the incoming write rate [28].

Through deeper analysis, we find the source of write stalls to be a more general form of disk overflow, which we refer to as "data overflow." Specifically, data overflow refers to the rapid expansion of one or more components in an LSM-KV system due to a surge in data flow into one of the components.

We categorize data overflows into three scenarios depending on the component that forms the data overflow. We also show how data overflow is able to explain the limitations that could not be explained with earlier studies. Based on these observations, we design and implement ADOC, an automatic tuning framework, to universally control and harmonize the data flow among LSM-KV components such that data overflow may be avoided. ADOC has the following four key features: 1) it improves performance by reducing write stalls through balanced use of resources; 2) it is a device transparent solution that improves performance for a wide range of devices, from traditional HDDs to state-of-the-art SSDs and PM devices; 3) it is an automatic tuning system that does not require human intervention, and 4) it is highly portable as it can be implemented by the native interfaces of LSM-KV systems.

Experimental results with RocksDB show that ADOC reduces the duration of write stalls by as much as 87.9% in the best case and improves performance by as much as 322.8% compared with the auto-tuned RocksDB, which takes a similar auto-tuning approach of ADOC. We also compare ADOC with the state-of-the-art LSM-KV SILK [7]. While there have been multiple novel LSM-KVs proposed more recently [11, 43, 44, 58], they mostly concentrated on making use of PM. As our target is a general-purpose LSM-KV that can accommodate all types of devices, we chose to compare it with SILK. Compared to SILK, ADOC achieves up to 66% higher throughput for the synthetic write-intensive workload that we used, while achieving comparable performance for the real-world YCSB workloads due to the higher read performance of SILK. However, SILK attains this performance at the expense of using 22.2% more main memory on average.

The source code of ADOC is available online [3].

2 Background

2.1 Advanced Storage Devices

Recent developments in storage technology have led to revolutionary advances in storage media. Two kinds of storage media, NVMe SSD and Persistent Memory, have entered the public realm and have been widely studied.

NVMe SSD: NVMe SSD refers to a class of SSD devices that are linked to the host via the PCIe bus and communicate with the host using the NVMe (Non-Volatile Memory express) protocol. NVMe is a communication interface and drivers designed for PCIe-based SSDs aim for efficient performance and interoperability. The command set in NVMe allows devices to directly communicate with the system CPU without an extra bus controller. Combined with the expanded command queue, NVMe provides much higher parallelism than conventional protocols like SATA and SAS.

Persistent Memory: Persistent memory (PM), or non-volatile memory (NVM), is a persistent medium that provides byte-addressability. The commercial PM product, Intel's Optane DC PM, can be deployed in Memory mode, as expanded memory, or App-direct mode, as a (block-device-like) storage device. Optane DC PM uses 3D XPoint technology, which, compared to traditional NVMe SSDs, offers lower write latency, provides byte addressing, and does not require garbage collection. However, as PM is attached to the memory bus, the IO processing of PM consumes more CPU resources. Also, as the PM device has limited bandwidth compared to DRAM, application bandwidth tends to quickly saturate as the number of threads increases [33, 56]. While Intel has announced the discontinuation of their Optane storage products effectively terminating 3D XPoint based products [20], other forms of NVM are still in development [35, 55]. Furthermore, we continue to see new developments such as CXL SSDs [35] that are expected to provide high-performing persistent storage similar to the Optane products.

2.2 Architecture of LSM-KVs

The majority of LSM-KVs follow the structure as that of RocksDB [25], which is one of the most popular LSM-KVs in industry and has been used as the platform of choice in many prior academic studies [7, 11, 14, 49, 58]. The structure consists of three major components and two data movement jobs to maintain these components as shown in Figure 2.

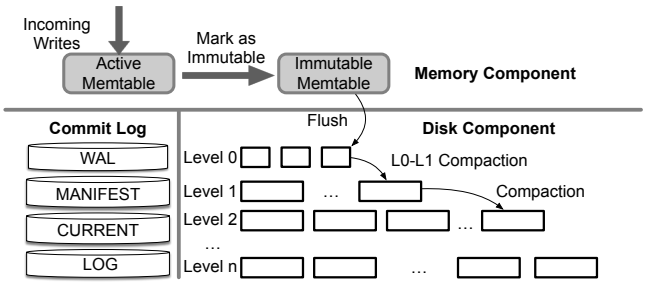


Figure 2: Architecture of RocksDB [15, 25, 48].

More specifically, the memory component caches the newest updates through an active and immutable Memtable [10, 17, 25, 41]. These Memtables are generally implemented with a skip list though other data structures such as a vector or hash table may be used. The commit log is the component that ensures system consistency. It is used to recover data when the system reboots from system failures. Finally, the disk component organizes the persisted data. Sorted String Tables (SSTable) that serve as the basic unit [46] are organized in a hierarchical manner in levels, starting from Level 0 to deeper (i.e., higher numbered) levels.

There are two types of background data movement jobs, flush and compaction. Flush moves the Immutable Memtable from the memory component to the disk component turning it into an SSTable in Level 0. When the capacity of a level reaches a certain threshold, compaction is triggered to merge SSTables in this level with SSTables in the next deeper level to form a set of new SSTables at the deeper level. Since LSM-KVs adopt out-of-place updates, there can be invalidated redundant data in SSTables. Compaction has the effect of removing some of these invalid redundant data.

Compaction can be further divided into Level 0-Level 1 (L0-L1) compaction and deeper level compaction. L0-L1 compaction is unique in that this activity cannot be executed in parallel with other L0-L1 compaction activities. This is because the SSTables in L0 can have overlapping keys as they are directly copied from the Immutable Memtable. In contrast, all SSTables at each level for Level 1 and deeper never have overlapping key ranges. Hence, deeper level compactions can occur in parallel.

2.3 Write Stall Issue

Previous studies have shown that the following three types of write stalls occur in modern LSM-KVs.

Memtable (MT) stall: This stall occurs when the memory component becomes full. For example, RocksDB sets the number of Memtables to 2, and when both are filled up, the system input is simply stopped resulting in a stall. This is known to be the most common case of write stalls [17, 41, 45].

Level 0-Level 1 Compaction (L0) stall: Similarly to MT stall, LSM-KVs will slow down or even stop the input stream when the number of L0 files reaches the set threshold, resulting in stalls. In RocksDB, the default slow-down threshold is 20, while at 36 the input stream is stopped. This type of control first appeared in LevelDB [17] but has been adopted by most subsequent implementations that use a similar compaction strategy [4, 25, 49].

Pending Input Size (PS) stall: LSM-KVs also slow down or stop the system when the pending input size of compaction jobs exceeds a certain threshold. Note that the pending input size not only refers to repeated and out-of-date entries in an SSTable, but also includes all the entries within the SSTable that is pending compaction. In RocksDB, the default pending compaction input size threshold for slowing down and stopping the system is 64GB and 128GB, respectively. Prior studies have used the term "compaction pending bytes" for this value [25, 58], while, in this study, we use the term "redundant data" instead. The aim of this control is to reduce the total amount of redundant data [15, 26, 28] or to avoid disk bandwidth bursts when compaction jobs happen in deep levels [45, 50].

Table 1: Specifications of Storage Devices.

Device	Product Name	Device Capacity	Sequential Bandwidth
PM	Optane DC PMM	512 GB	2300 MB/s
NVMe SSD	Samsung 970 PRO	1 TB	2700 MB/s
SATA SSD	Intel DC S4500	960 GB	490 MB/s
SATA HDD	Seagate ST1000DM010	1 TB	210 MB/s

3 Observations from Previous Studies

In this section, we use experimental observations with RocksDB to revisit previous studies. While these earlier studies provide valuable insight into the causes of write stalls, we show that they are all limited in that these insights are not able to explain many of the experimental results.

3.1 Experimental Settings

The experiments in this section are conducted on a server that follows the recommended configuration for installing the Optane DC PMM. It has two Intel Xeon(R) Gold 6230 processors with 2.10GHz frequency with a total of 40 cores (20 cores each) and is equipped with 128GB DDR4 DRAM. We consider four different storage devices as listed in Table 1, which shows the device types, the producer and product names, the capacity of the devices, and the sequential bandwidth as specified by the manufacturer or reported in an earlier study [29–31, 33].

All experiments are run on Ubuntu 18.04 LTS, running RocksDB 6.11 [24] compiled with CMake 3.10.2. We run the *fillrandom* workload in db_bench [23] issuing uniformly distributed random writes in each scheme for one hour, a time period sufficient to trigger all kinds of write stalls and maintain a trend in all devices. Each entry consists of a 16-byte key and a 1000-byte value. All experiments are evaluated under peak throughput since write stalls occur only when the write pressure is high enough as shown in Figure 1. Write stalls are observed with an embedded event listener provided

Table 2: Summary of confirmations and limitations on conclusions made by existing studies on write stalls.

Original Conclusion	Points we confirm	Limitations we find
Resource Exhaustion [7, 15, 34, 43, 51, 58, 60]	[C1]: High CPU utilization is a source of write stalls. Increasing background threads reduces CPU utilization and hence, reduces write stalls [34,43,51,60]. [C2]: Most devices show increased bandwidth usage and decreased CPU utilization when increasing the number of threads. The occurrence of write stalls increases when the number of threads exceeds a certain threshold [15]. [C3]: As modern devices provide much higher bandwidth and parallelism, the stall occurrence and duration on PM and NVMe SSD are much lower than those on SATA devices [7,43,58].	[L1]: Continued increase beyond a certain number of threads results in a continued decrease of (normalized) CPU utilization, but results in an increase in write stall duration. That is, reduced CPU utilization does not result in reduced write stalls. [L2]: Even with high CPU utilization, simply by increasing the batch size, write stalls may be reduced. That is, CPU utilization and write stalls do not correlate. [L3]: Modern devices can provide far more bandwidth than conventional devices, but write stalls may still occur before its bandwidth capacity is reached.
L0-L1 Compaction Data Movement [7,58]	[C4]: At early phases of execution, performance troughs in NVMe SSD and PM match the occurrence of compaction [7,58].	[L4]: Correspondence between performance troughs and L0-L1 compaction jobs diminishes over time, especially in the multi-threaded environment.
Deep Level Compaction Data Movement [45,49,50]	[C5]: The processing rate of flush jobs decrease when more threads are spawned for compaction jobs [45,49,50].	[L5]: As the number of threads increases, the occurrence of PS stalls that are caused by slow compaction decreases.

with RocksDB. This listener provides basic information such as the total duration and the number of occurrences of each type of write stall. All experimental results obtained are the average of three rounds of executions; the three rounds take over 240 hours to execute.

For the experiments, we mainly consider the impact of two parameters that have a strong effect on performance [13–15, 26]. The first is the number of threads that run concurrently in the system, which determines the resources that are allocated to each thread such as CPU time and bandwidth. In RocksDB, in particular, by default, a quarter of the threads are allocated for flush jobs (rounded down), while the rest perform compaction. The second parameter is batch size, which is the size used for both Memtable and SSTable. This value is critical for analyzing the behavior of LSM-KVs because 1) it controls the scheduling pattern and input scale of background jobs [6,7,14,43] and 2) it affects the data distribution at the various levels [11,13,14,22].

3.2 Limitations of Existing Studies

In this section, we discuss the limitations of earlier studies regarding write stalls. As we shall show, these studies tend to analyze the causes of write stalls in LSM-KV stores from a single component perspective. Through experimental observations, which we discuss below and summarize in Table 2, we show that these conclusions cannot be fully generalized.

Resource Exhaustion: Some earlier studies conclude that write stalls are caused by bandwidth congestion [7,45], while others consider CPU limitation as the root cause [43,58,60]. We revisit these conclusions, starting with CPU utilization.

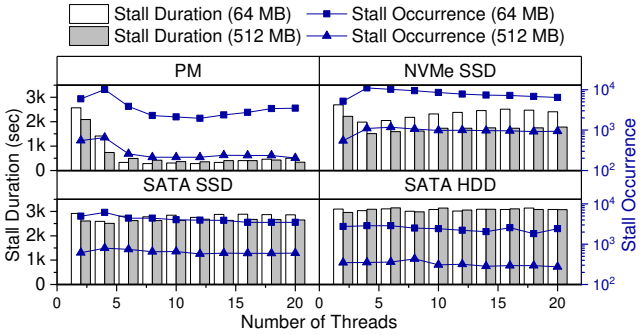


Figure 3: Duration and occurrences of write stalls as the number of threads is increased.

Figure 3 shows the stall occurrences and duration as the thread count increases. We observe that for PM and NVMe SSD, the stall duration decreases as the thread count increases (more notably with PM) until up to six threads. Also, as shown in Figure 4, up to six threads, the CPU utilization (normalized to the number of threads) remains relatively high for PM and NVMe SSD. Just based on these observations, one could conclude that the shortage of CPU resources, that is, high CPU utilization, is the cause for write stalls (Table 2 [C1]).

However, we also observe from Figure 3 that write stall occurrences start to drop, while the duration increases slightly, as the thread count increases beyond four (where CPU utilization decreases as shown in Figure 4) (Table 2 [L1]). This characteristic is particularly evident in the two advanced devices with higher bandwidth and parallelism. Based on these observations, our conclusion is that, while limited CPU capac-

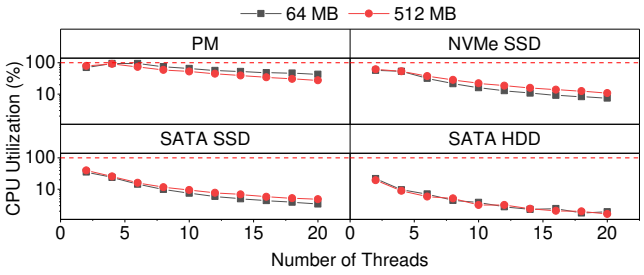


Figure 4: Comparison of normalized CPU utilization as threads are increased.

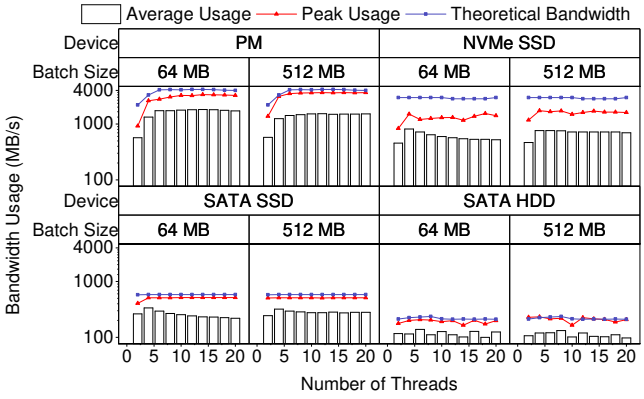


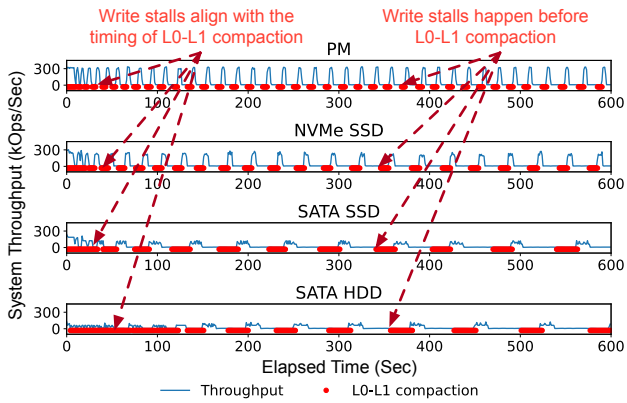
Figure 5: Comparison of bandwidth with an increasing number of threads and different batch sizes. (Note that *y*-axis is log scale.)

ity may be the cause of write stalls for some scenarios, this is difficult to generalize.

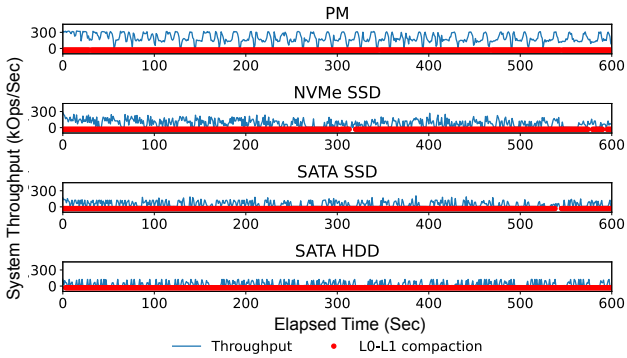
Moreover, when the batch size is increased from 64MB to 512MB in NVMe SSD and SATA SSD, even while CPU utilization does not show significant changes (Figure 4), we observe both duration and occurrence of write stalls decreases (Figure 3). In the PM case, we observe that beyond six threads, CPU utilization for 512MB batch size is lower than with 64MB, yet the stall duration is actually higher, except with over 15 threads and beyond. From these observations, we conclude that CPU utilization and write stall do not correlate well (Table 2 [L2]).

Other studies have pointed to the disk bandwidth limitation as the source of write stalls [7, 45, 58]. It is argued that with the increase in thread count the disk bandwidth will be overwhelmed leading to the stall problem (Table 2 [C2] and [C3]). However, the following observations showing that the system can be stalled even when disks are idle tell a different story.

Observe the theoretical bandwidth limit, which is the peak bandwidth observed when the device is flooded with requests generated from the FIO [2] tool with multiple threads, and the peak bandwidth used as the thread count increases in Figure 5. Although for the HDD and SATA SSD the peak reaches the theoretical bandwidth limit, for the NVMe SSD and PM, a large idle bandwidth gap remains, indicating that write stalls occur even if there is bandwidth to spare (Table 2 [L3]). In addition, if the write stall is due to insufficient bandwidth,



(a) Experiments with 2 threads and 64 MB batch size. Left arrows point to occurrences where L0-L1 compaction maps well with write stalls, while right arrows point to occurrences where they do not match.



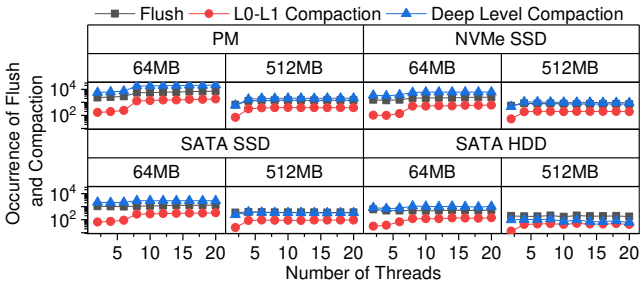
(b) Experiments with 20 threads and 64 MB batch size. L0-L1 compaction is triggered much more frequently than those in (a), and the occurrences of write stalls show no relation with the L0-L1 compaction jobs.

Figure 6: Timing of L0-L1 compaction and throughput for the thread count of 2 and 20.

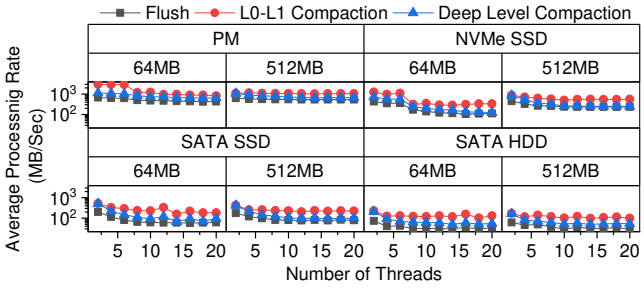
devices should be under high write pressure for an elongated period. This should move the average bandwidth close to the theoretical value. However, we see in Figure 5 this is not so.

L0-L1 Compaction Data Movement: SSTables in L0 are unordered and their keys may overlap as they are generated by flush jobs. Thus, L0-L1 compaction, which takes all L0 files as its input, cannot be parallelized with other L0-L1 compaction jobs [16]. Former studies have taken this unique limitation as the direct cause of write stalls [7, 58].

We observe from Figure 6, which plots the process timing of L0-L1 compaction with instantaneous throughput, that this earlier conclusion is partially true. Specifically, when running with two threads (Figure 6(a)), initially, we observe system throughput dropping immediately as L0-L1 compaction is triggered, as designated by the dashed arrows on the left. This is the regularity observed by Yao et al. [58] (Table 2 [C4]). However, as the system continues to process the input stream, this correspondence disappears, as with the apparent misalign-



(a) Comparison of occurrence of background jobs.



(b) Average processing rate of background jobs.

Figure 7: Comparison of occurrences of background jobs (flush, L0-L1 compaction, and deep level compaction) and their average processing rate as the number of threads and batch size are increased, measured for one hour of execution.

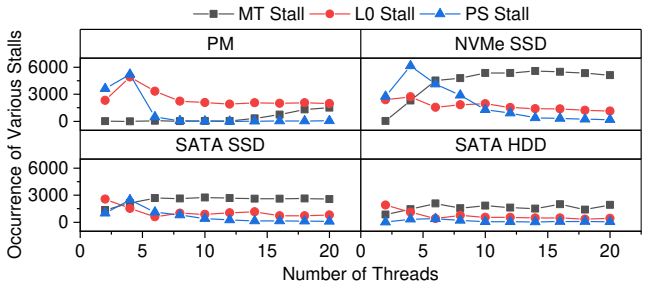
ments as designated by the dashed arrows on the right. Moreover, when the thread count is increased to 20, as shown in Figure 6(b), L0-L1 compaction occurs much more frequently than for the 2-threaded case, while showing no evident mapping relation between the timing of L0-L1 compaction and write stalls (Table 2 [L4]).

Deep Level Compaction Data Movement: Yet another set of earlier studies contend that the high resource consumption of deep level compaction jobs that are competing with other background jobs is the cause of write stalls [7, 45, 49, 50]. Again, this is partially true, as increasing the number of threads does lead to more frequent compaction jobs, as shown in Figure 7(a), and the average processing rate of background jobs decreases, as shown in Figure 7(b) (Table 2 [C5]).

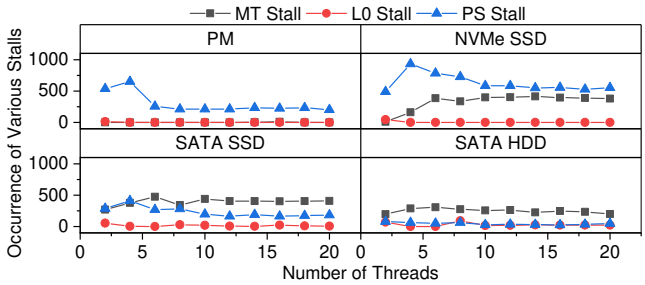
However, if the conflicting compaction jobs were the main source of write stalls, we should have observed the occurrence of PS stalls increase just like how the slow flush rate increased MT stalls. Instead, as we observe in Figure 8, the occurrences of PS stall decrease as the number of threads increases and the batch size increases, which is contrary to the earlier conclusion.

4 Data Overflow

As seen from the previous section, earlier studies focus on individual aspects that could be the cause of write stalls. Our analysis of modern LSM-KVs reveals a more general source of write stalls, that is, what we refer to as *data overflow*. In this section, we explain the formation of data overflow and use data overflows to explain the limitations observed in Section 3.



(a) Occurrence of various write stalls while increasing the number of threads with 64 MB batches.



(b) Occurrence of various write stalls while increasing the number of threads with 512 MB batches.

Figure 8: Breakdown of various write stalls.

4.1 Data Overflow in Modern LSM-KVs

Data overflow refers to the rapid expansion of one or more components in an LSM-KV system due to a surge in data flow into one of the components. It happens when the processing rates of different background jobs do not match each other. We identify three types of data overflow, namely, Memory Overflow, Level 0 Overflow, and Redundancy Overflow, as shown in Figure 9. We now describe these in more detail.

Memory Overflow (MMO): MMO occurs when the system input rate surpasses the Immutable Memtable flush rate. Consequently, as the Immutable Memtable cannot be flushed in time, there will not be enough space in the memory component to absorb new data. In most modern LSM-KVs, upon MMO, the system stops receiving input as there is no room to buffer the incoming updates. This results in an MT stall.

Level 0 Overflow (L0O): L0O occurs when the processing rate of L0-L1 compaction is not able to match the flush rate. This results in the number of SSTables in Level 0 to rise. In modern LSM-KVs, the input stream is stopped or slowed when this value reaches a certain threshold so that the accumulated data may be consumed resulting in an L0 stall.

Redundancy Overflow (RDO): RDO occurs when the working efficiency of compaction threads cannot match the rate in which redundant data is generated. In modern LSM-KVs, when the size of redundant data reaches a threshold, the system will slow down or stop the input, and wait for the compaction threads to clear out the accumulated redundancy. Such action results in PS stalls.

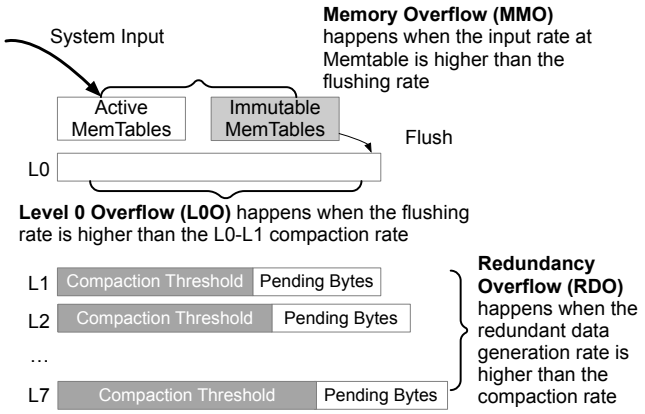


Figure 9: Data overflow scenarios in modern LSM-KVs.

4.2 Explaining the Unexplained

In Section 3, we showed how conclusions made in earlier studies could not explain the reason behind write stalls for some portions of the extensive experimental results that we had obtained (Table 2). Here, we present how such incongruities can be explained with data overflow. To observe write stalls in this section, we make use of the LOG file that RocksDB provides. While the embedded listener used in the previous section provides only basic information, the LOG file provides more detailed information of each write stall, including the stall type, the limited input rate, and the exact timestamp.

Resource Exhaustion: First, consider how write stalls occurred under low utilization of resources (3.2, Table 2 [L1], [L2] and [L3]). The key underlying reason is that all three kinds of data overflow will stop or slow down the input before the system reaches the hardware limitation. Let us elaborate.

Firstly, when the flush rate is not high enough to persist the incoming requests in time, MMO will stop the input. Figure 7(b) shows that flush jobs are being allocated the least bandwidth among the background jobs and thus, the average flush rate monotonically decreases with the number of threads. This is because as the number of threads increases, more threads are forced to share the limited bandwidth, resulting in less bandwidth being allocated to the flush threads. This results in the Immutable Memtable not being flushed fast enough, which is the most common reason for write stalls that occur in SATA HDD as well as other devices when there are too many threads.

Secondly, L0O occurs as compaction of SSTables in L0 cannot keep pace with flush jobs, resulting in the number of SSTables in L0 reaching its threshold, and thus, the input stream being stopped or slowed. As direct evidence, Figure 10(a) shows how the occurrences of write stalls, marked by the vertical blue lines, correspond to the peak in the number of L0 SST files. Since L0-L1 compaction jobs are not being executed in parallel, increasing the number of threads will not help in reducing L0O as the processing rate does not increase. Hence, the system is stalled even when there is enough CPU resources.

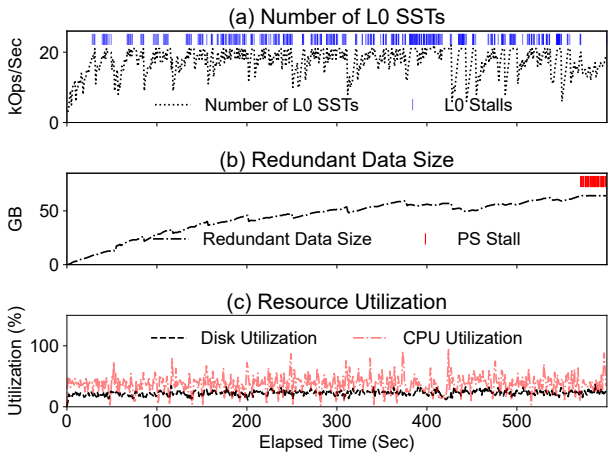


Figure 10: System statistics (first 600 seconds) when running the *fillrandom* workload with 4 background threads with NVMe SSD, which shows the most occurrences of write stalls.

Lastly, we show how RDO makes LSM-KVs stall when both CPU utilization is low and there is disk bandwidth to spare. Figure 10(b) shows how the redundant data tend to accumulate as the deep level compaction cannot keep pace with L0-L1 compaction. Eventually, RDO occurs when the redundant data size reaches 64GB, which is the default PS stall threshold, at which point the system slows the input, represented by the red lines. Note, however, that at these stall points, both the CPU and bandwidth utilization are low (and stable) as any other points in execution (Figure 10(c)), showing how RDO can occur despite low resource usage.

L0-L1 Compaction Data Movement: We next discuss why L0-L1 compaction does not align with the occurrence of write stalls (Table 2 [L4]). First, consider the misalignment in Figure 6(a), when there are only two threads. In the early stages of execution, as data in the deeper level have not been accumulated, L0-L1 compaction is easily assigned a thread. Hence, we see a nice alignment of the compaction with the write stall. However, as execution continues, data starts to accumulate and more deep level compaction requests get to be made. With only a limited number of threads, this dwindles the chance of L0-L1 compaction from being assigned a thread. Thus, L0-L1 compaction and write stalls start to misalign.

Now consider the situation when the number of threads is 20. Here, we have enough threads to always assign for L0-L1 compaction. However, with a large number of threads, bandwidth for flush jobs diminish, and thus, the processing rate of flush jobs becomes much lower (Figure 7(b)) causing more frequent MMO (Figure 8). That is, the frequent write stalls here are due to MT stalls, and L0O hardly occurs showing no relation to the L0-L1 compaction jobs as shown in Figure 6(b).

Deep Level Compaction Data Movement: Finally, while earlier studies concluded that PS stalls increase with thread count, which was true for up to four threads, we also saw

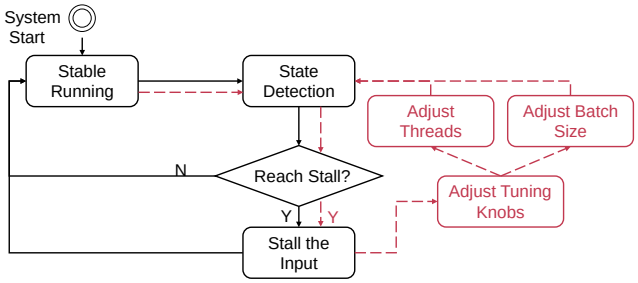


Figure 11: The solid black diagrams show the default control flow of RocksDB. ADOC extends this with the red diagrams (including the dashed arrows) to adjust tuning knobs during execution.

the opposite as more threads were added (Figure 8 and Table 2 [L5]). This can be explained with RDO. The initial spike in PS stalls (up to four threads) shown in Figure 8 is due to the flush threads obtaining sufficient bandwidth. In addition, with this increase to four, occurrences of L0O are decreased as more threads are allocated. This results in more data being crowded into the deeper levels and overwhelming the processing rate of deep level compaction jobs. This results in a significantly increase of PS stalls (Figure 8). However, beyond this threshold of 4 (for most cases), the flush rate is reduced and consequently, the data redundancy rate also diminishes. This results in less need to process deep level compaction jobs, which finally results in less RDO.

5 Automatic Data Overflow Control

The goal of our study is to minimize (and eventually remove) the effects of write stalls on LSM-KVs. To this end, we develop a framework, which we call **ADOC** (**A**utomatic **D**ata **O**verflow **C**ontrol), that controls the dataflow such that data overflow may be minimized. Dataflow is controlled by online tuning of the number of threads and the batch size as these values have a strong influence on dataflow, as seen from discussions in Section 4, and most LSM-KVs [15,48,52] provide APIs to adjust these two values without rebooting the system.

We develop ADOC under two principles. The first is device transparency. Instead of targeting optimizations to a particular storage device, as new storage devices will continue to evolve [35,54], ADOC should be able to tune itself to reduce write stalls irrespective of the underlying storage device. For this, ADOC monitors the flow of data amongst the components independent of the specific performance parameters of the underlying device. Then, the thread count and batch size are adjusted to control the processing rate and scheduling frequency of background jobs, thereby controlling the data flow within the LSM-KV.

The second principle is ease of portability. As shown in Figure 11, the design of ADOC is a straightforward extension of the RocksDB control flow mechanism. Unlike MatrixKV and similar approaches [14,45,50,58] that change the compaction strategy or SILK [7] and auto-tuned RocksDB [39] that adjust the internal thread scheduling and IO process, ADOC does not disrupt the internal architecture of the LSM-KV system making it highly portable.

In our current implementation in RocksDB, we make modifications to only two classes. One is the `Options` class, which controls whether the ADOC tuner will be enabled or not and records the instantaneous information of the system in a shared C++ vector. The other is the `tuner` class itself, which will periodically wake the tuner threads to perform tuning actions. In total, we add around 300 lines of code (LOC) to implement ADOC with 250 LOC for the tuner and 50 LOC to collect system states.

We now discuss the triggering mechanism and the actions that we employ to adjust these parameters. Recall that we identified data overflow as the source of write stalls. Thus, for every time window T_w, ADOC monitors for data overflow and takes action accordingly as explained below. In our current implementation, we set T_w to one second based on empirical observations; values larger are not agile enough to quickly detect the overflows leading to deteriorated performance for state-of-the-art high-performing storage devices, while values smaller could incur overhead as well as lead to fluctuations due to responding too quickly.

MMO: ADOC determines that MMO is occurring simply when the active Memtable is filled before the Immutable Memtable gets flushed. Upon MMO detection, ADOC increases the flush rate by reducing the number of threads, which will have the effect of reserving more bandwidth for the flush jobs, and increasing the batch size to increase the processing rate.

L0O: Determining whether L0O is occurring follows the same logic as RocksDB, that is, when the number of L0 files exceeds the threshold, which is 20 by default in RocksDB. Upon L0O detection, ADOC will increase the number of threads. This has the effect of improving the chance of L0-L1 compaction being assigned a thread and decreasing the flush rate to ease the overflow. The batch size, in this case, is unchanged as increasing it will increase the load on L0-L1 compaction, and decreasing it will generate more L0 files, both leading to more L0 stalls.

RDO: Like L0O, determining if RDO is occurring follows the same logic as RocksDB, that is, when the total redundant data size exceeds the threshold, which is 64GB by default in RocksDB. Upon detection of RDO, ADOC will increase the number of threads and decrease the batch size. The former is to increase the rate of deep level compaction (and also reduce flush rate) and the latter is to allow the scheduler to generate more fine-grained compaction jobs as small and dense compaction jobs can help improve the efficiency of redundancy reduction.

If multiple overflows are detected in T_w, we choose to handle the overflow in L0O, RDO, MMO order based on our experimental observations. Also, when turning the tuning knobs, we take the approach used by the Additive In-

Table 3: Schemes Evaluated

Name	Description
RocksDB-DF	RocksDB default setting
RocksDB-AT	RocksDB with auto-tuner on
SILK-D	SILK with RocksDB default setting
SILK-P	SILK setting set as in SILK paper [7]
SILK-O	SILK optimized to our setting (Section 3)
ADOC	RocksDB that enables ADOC tuner

crease/Multiplicative Decrease (AIMD) algorithm [9], best known for its use in TCP congestion control [40]. The reason for using AIMD is the fitness between the algorithm and ADOC's working scenario, that is, gently increasing the tuning knob to explore the suitable configuration and rapidly removing the over-allocated resources to avoid resource competition. In detail, we increase the number of threads by 2 and the batch size by 64MB, which is the default thread number and batch size value of RocksDB, while when decreasing, the values are reduced by half. After the adjustment, the number of threads that are allocated for flush jobs will also be adjusted to a quarter of the total number, just as the default setting.

6 Evaluation

6.1 Experiment Setups

Basic Settings: We use the same hardware and software setups as described in Section 3.1. For the basic setup, we follow that of SILK and set the maximum batch size to 512MB [7]. All schemes, including ADOC but excluding SILK, are based on RocksDB v7.5.3, the latest version as of this submission. SILK is built based on RocksDB 5.7.1 (early 2018) and our attempt to port SILK to more recent versions failed due to compatibility issues as considerable optimizations have been made since RocksDB v6 [27]. Thus, all performance measurements for SILK are done on RocksDB v5.7.1. In one of our experiments, we also show the results of ADOC ported on v5.7.1, which show some discrepancies with the results of v7.5.3, but overall, are quite similar in trend.

Schemes Compared: The schemes that we evaluate are as listed in Table 3. There are two settings of RocksDB, three settings of SILK, and ADOC. For RocksDB, we have RocksDB-DF with the default configuration, that is, two background working threads and 64MB batch size, and RocksDB-AT, an auto-tuner enabled version. RocksDB-AT automatically changes the threshold of the rate-limiter, which limits the number of IO operations generated by background threads [1] based on the IO pressure of background threads. RocksDB-AT also adjusts the allocation rate of each thread based on the thread priority to avoid starving compaction jobs, which has lower priority than flush jobs.

As for SILK, the three different configurations are as follows. The first is SILK that runs with the same configuration as the default configuration, which we refer to as SILK-D (D for default). The second is SILK-P (P for paper), which refers to SILK that runs with the same settings as mentioned in its original paper [7]. The configuration of SILK-P is of 4 background threads and 128MB batch size. The third configuration is SILK-O (O for optimal), which we believe to be the best performing setting in our experimental platform, with 8 background threads and 512MB batch size, which were obtained manually through exhaustive tuning attempts where we considered ten (2, 4, 8,..., 18, 20) thread and two (256MB and 512MB) batch configurations. We did not consider batch sizes 64MB and 128MB as when their results were observed for SILK-D and SILK-P, we found larger batches to be clearly better. In addition to the thread and batch size settings, the SILK implementation makes use of particular hard-coded settings. One is the allocation of bandwidth that is hard-coded into the db_bench tool. To faithfully configure our three versions accordingly, we set the configuration flags such that a quarter of the entire media bandwidth (Table 1) is reserved for compaction jobs while the rest is reserved for flush jobs. This ratio preserves the ratio used in the original paper. Additionally, as in the original implementation, we disable L0O by setting the number of L0 SST files threshold, which slows the input stream when reached, to an extremely large value.

6.2 Microbenchmark Performace

In this section, we make use of the same microbenchmark workload used in Section 3, that is, the db_bench random filling benchmark. We consider the three performance measures, namely, the throughput, the stall duration, and 99^{th} tail latency for the first 3600 seconds of execution.

Throughput: Figure 12 compares the system throughput for all the schemes that we consider. A few notable observations can be made as follows. First, ADOC shows the best performance over all devices. It shows 66.7%, 37.8%, 31.0%, and 55.1% higher average throughput over the next best performing scheme, SILK-O, for PM, NVMe SSD, SATA SSD, and SATA HDD devices, respectively. Second, the three variations of SILK show performance in SILK-O, SILK-P, and SILK-D order. Among these, the best performing SILK-O does considerably better for high-end devices, but not much so for low-end devices. Most notably, we observe that the configuration of SILK, the best of which is not straightforward to find, has a considerable effect on overall performance. Finally, RocksDB-DF and RocksDB-AT fare comparably with SILK-D and SILK-P, but worse than SILK-O. While RocksDB-AT automatically decides the bandwidth usage of different background jobs [39], the results show that this is insufficient in bringing out the best performance. Consequently, we find that RocksDB-AT performs better than RocksDB-DF for NVMe SSD and SATA SSD, which concurs with the fact that RocksDB is optimized for flash devices [11, 15, 25], but performs worse for PM and SATA HDD. It is also limited in that the user needs to provide the bandwidth information.

For clarity, hereafter, we omit the results for RocksDB-DF

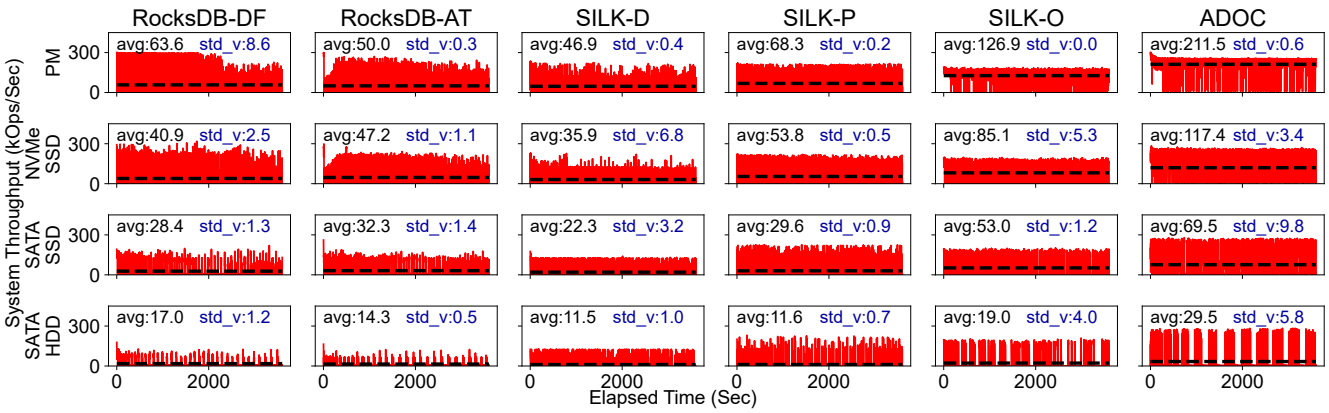


Figure 12: The solid red lines represent the instantaneous throughput during a single run of 3600 seconds, while the dotted black lines represent the average throughput of this run. The numbers shown within the box are the average throughput and the standard deviation of the three runs for each case.

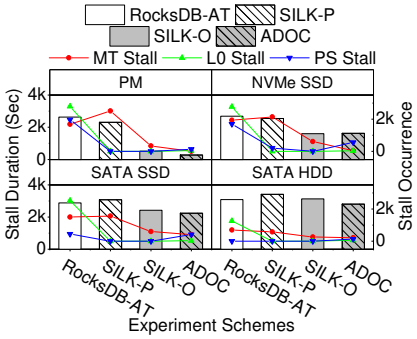


Figure 13: Stall duration (bar) and occurrences (lines).

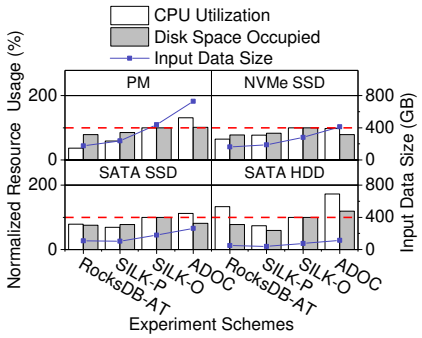


Figure 14: Comparison of CPU utilization, disk space occupied, and input data size with the *fillrandom* workload.

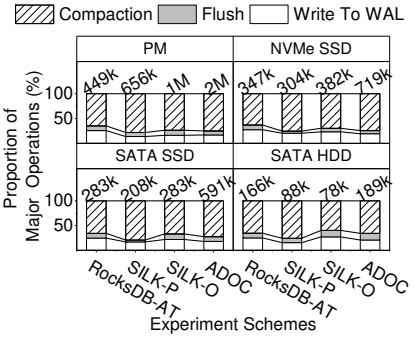


Figure 15: Proportion of major operation occurrences, with numbers representing total occurrences.

and SILK-D, the two low-performing schemes.

Stall Duration: Figure 13 compares the write stall duration for the various schemes. The bars indicate the total stall time, while the scattered lines mark the average occurrences of the three different sources of write stalls.

ADOC reduces the stall duration compared to SILK-O for PM, SATA SSD, SATA HDD by 45.2%, 8.7%, and 10.3%, respectively. However, for NVMe SSD, stalls seem to be elongated by 1.5%. Looking at the sources of the stalls, we observe that for MT and L0O stalls, ADOC is the lowest. For the PS stall, however, ADOC seems to be doing worse than both SILK schemes, seemingly negating the reduction of the other stalls. However, one must take into consideration the fact that these are measurements taken for the same 3600 seconds. Since the throughput of ADOC is considerably higher than the other schemes, ADOC takes in much more data from the input stream, specifically, 66.9%, 46.9%, 46.5%, 53.5% more data than SILK-O with PM, NVMe SSD, SATA SSD, and SATA HDD, respectively, as shown by 'Input Size' in Figure 14. This results in more data being accumulated into the deeper levels resulting in particularly higher PS stalls (Figure 13). Also, we observe that these additional PS stalls have a positive effect on the space amplification of the system.

That is, the Disk Space Occupied results in Figure 14 show that despite ADOC processing a much higher volume of input data, the system does not occupy significantly more space compared to the other schemes. Even on the HDD, which has the worst compaction performance, ADOC accepts 53.5% more input data than SILK-O, yet the disk space occupied is only 19.4% larger. However, we also see from Figure 14 that this results in higher CPU utilization for ADOC. For example, with HDD, ADOC spends 72.5% more CPU time than SILK-O. Figure 15 shows the breakdown of the major operations for each scheme, with the total number of operations shown on top of each bar. These numbers were obtained by sampling the call stack of RocksDB using the *perf* [5, 18] tool at 99Hz sampling frequency. While not exact, these numbers provide an estimate of the CPU time spent for the operations as have been used in other studies [8, 19, 61], as the CPU time will be proportional to the operation count. We observe that the proportion of each operation is relatively stable for schemes on each device. However, we also observe that the operation count for ADOC is considerably higher than SILK-O for all devices, with the largest difference being 143.6% higher with HDD. This is because ADOC accepts more input data than the other schemes, with the additional inputs generating more

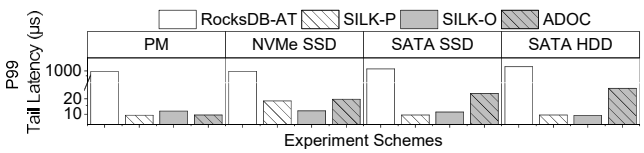


Figure 16: Average 99^{th} tail latency in *fillrandom* workload.

Table 4: Data distribution and the composition of request types for the six YCSB workloads. (RMW: read-modify-write)

Workload	Distribution	Request Composition
A	Zipfian	50% Update 50% Read
B	Zipfian	95% Read 5% Update
C	Zipfian	100% Read
D	latest	5% Insert 95% Read
E	uniform	5% Insert 95 %Seek
F	Zipfian	50% Read 50% RMW

data movement jobs, and also because the occurrences are correlated to the CPU utilization of the related functions [18]. In addition, ADOC and SILK use different approaches to reserve bandwidth for flush jobs. SILK puts the compaction jobs to sleep when facing bandwidth congestion, while with ADOC there is at least one compaction job working at all times. As a result, ADOC shows higher CPU utilization and operation count than SILK.

Tail Latency: Figure 16 shows the 99^{th} tail latency results obtained over 3600 seconds of execution. We observe that both SILK schemes do well in terms of tail latency, which was the target performance measure of SILK. Compared to SILK-O, ADOC does better for PM, but is higher by 70.1%, 131.2%, and 242.9% for NVMe SSD, SATA SSD, and SATA HDD, respectively. Again, however, we note that ADOC generates over twice the number of requests than SILK-P and 50% more than SILK-O, meaning that it faces edge cases (e.g., foreground GC, conflict I/O request in half duplex bus, etc.) much more often leading to higher tail latency.

6.3 Macro Benchmark

In this section, we consider real-world workloads using the YCSB benchmark. The YCSB benchmark [12] is a popular benchmark tool that generates workloads following real-world data characteristics. We run the six workloads with characteristics as shown in Table 4, executing them in the suggested order [12, 49], that is, execute the loading stage first, followed by the run stages A, B, C, D, and F. Then, we reload the data and execute workload E. We load 50M entries (10B keys and 1000B values) during the load stage and then execute each run stage for one hour.

Figure 17 shows the throughput of all schemes. (Note that there is a ADOC-5.7.1 scheme that has been added to the results. We elaborate on this later.) Comparing the auto-tuning systems, RocksDB-AT and ADOC, we find that the latter per-

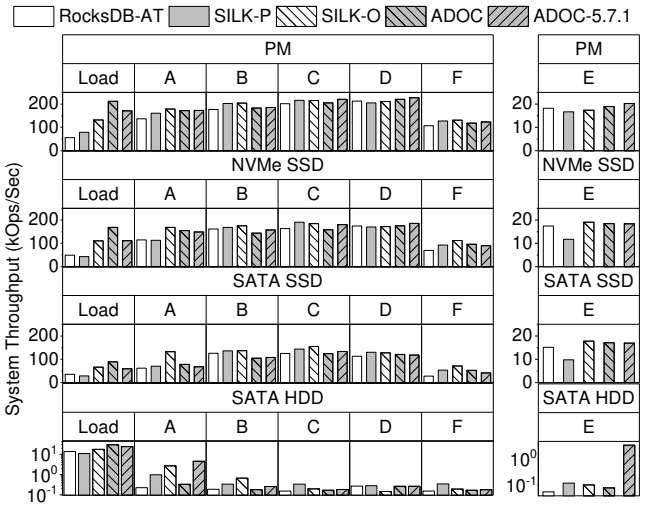


Figure 17: Comparison of system throughput in different stages of YCSB workloads.

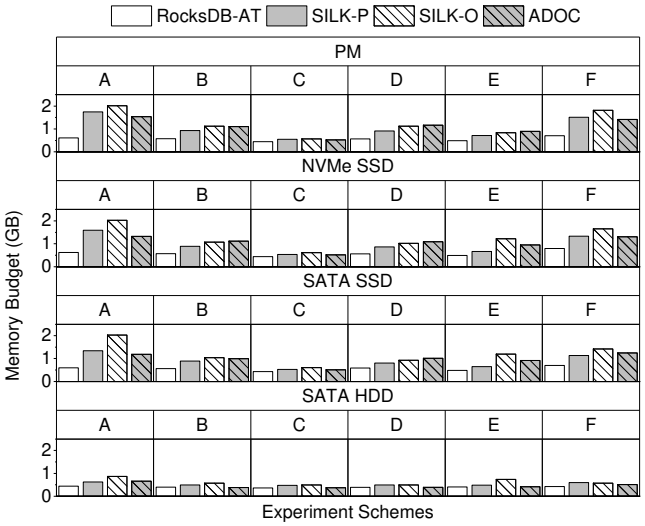


Figure 18: Comparison of main memory footprint.

forms 7.6% to 11.4% better over all the devices considered. Comparing SILK-O and ADOC, the two gives-and-takes. For Load, a purely write workload, ADOC beats SILK-O. However, for workloads YCSB-A, -B, -C, and -F, SILK-O performs better. For YCSB-D and -E, the winner depends on the device. Recall that the RocksDB versions on which SILK-O and ADOC run differ. To remove the version effect, we also run ADOC on RocksDB 5.7.1. These results, denoted, ADOC-5.7.1 in Figure 17, show that the version difference has some effect on ADOC performance, with the older version performing better in the majority of run stages, most notably for YCSB-A and -E on the HDD.

The main reason SILK-O does well, despite the fact that ADOC performs considerably better than SILK-O for writes as was shown with the microbenchmarks, may be attributed to the large memory usage. As shown in Figure 18, we find that SILK-O uses as much as 76.8% more memory than ADOC. Larger memory allows more requests to be serviced from the

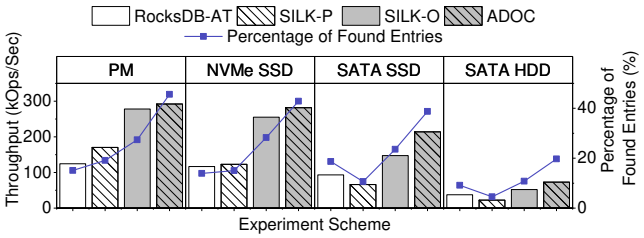


Figure 19: Throughput for the *read-while-writing* workload.

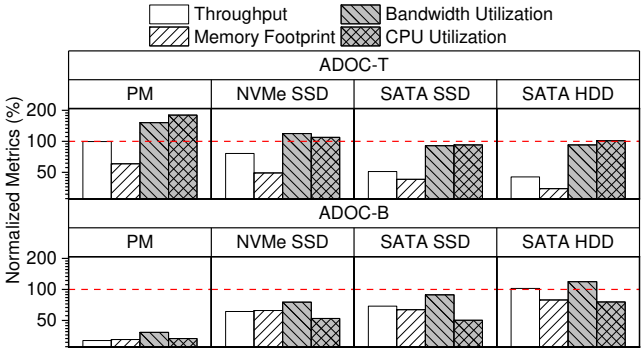


Figure 20: Comparison of system throughput and resource consumption with different tuning knobs triggered, the values shown are normalized to the value of ADOC.

data buffered in memory benefiting not only read-intensive but also update-intensive workloads like YCSB-A and -F.

Finally, the tail latency results (not shown) do not reveal any surprises; overall, SILK-O does best, but gives-and-takes between SILK-P and ADOC for particular workloads.

In conclusion, the performance results show that SILK-O and ADOC are comparable. However, recall that for SILK, performance varies considerably depending on the initial setting and that the "optimal" SILK-O setting was manually obtained. This is in contrast to ADOC being an online tuning system that does not need human intervention.

As a supplement to the macro benchmark, we also evaluate the system under the *read-while-writing* workload in db_bench that uses two threads, the reader and writer, to generate 50M write and read requests, respectively. Note that as the reader and writer are running concurrently, the reader thread may request entries that have not yet been persisted by the writer; we show the percentage of found entries along with the throughput of the various schemes in Figure 19. The results show that ADOC achieves 5.2% to 45.3% higher throughput than SILK-O, and 95% to 135% higher than RocksDB-AT.

6.4 Number of Threads versus Batch Size

Throughout our discussions, we considered the number of threads and batch size as our tuning parameters. In this section, we consider the effect of each parameter on ADOC. For this, we consider the *fillrandom* workload in Section 6.2 on two versions of ADOC, ADOC-T and ADOC-B, the former that only tunes the number of threads and the latter that only tunes the batch size.

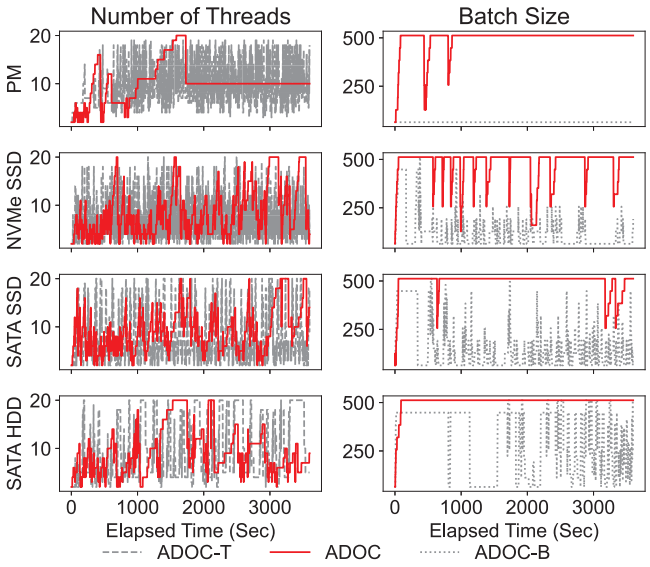


Figure 21: Tuning actions during one-hour execution.

Figure 20 shows the throughput and resource usage of the two schemes relative to ADOC initially set with the default values. Observing the throughput for ADOC-T, we see that it performs well for PM, but that the throughput deteriorates with slower devices. More specifically, since ADOC-T does not adjust the batch size, its memory footprint is smaller, but this comes at the price of performance. The disk bandwidth and CPU utilization are also high since the batch size is relatively small, resulting in more frequent data movement.

In contrast, we see that ADOC-B is almost the complete opposite. It performs well for HDD, but throughput deteriorates as the devices get more powerful. Since it does not increase the number of threads, the memory footprint, the CPU utilization, and the bandwidth utilization are all relatively low as fewer data movement jobs get triggered. That is, while resources are abundant, there are not enough workers to take advantage of them. However, for the HDD where bandwidth is limited, having few threads allow these threads to make full use of the bandwidth incurring less write stalls.

Figure 21 shows how the number of threads and the batch size are adjusted when only one parameter is considered with ADOC-T and ADOC-B, respectively, versus when both are considered with ADOC. The results show that tuning both parameters together reduces the frequency and fluctuations of the adjustments resulting in a much more stable setting.

In conclusion, we find that the number of threads and batch size have a complementing effect that stabilizes the tuning process, consequently resulting in better performance.

7 Related Work

Write Stall Issues: LSM-KV experiences periodical performance drops when facing heavy writing pressure due to write stalls. SILK focuses on the long tail latency problem caused

by write stalls. To remedy this problem, it adjusts the priorities of background threads and adds rate limiters to these background threads [7]. It ensures the L0-L1 compaction will be handled in a timely manner to avoid disk overflow. Other studies propose new scheduling strategies to align the background compaction jobs in different levels and align the start time of each compaction according to the available resources [45, 50]. MatrixKV observes the shortcoming of the original SSTable format and points to the slow L0-L1 compaction as the root cause of write stalls with PM devices [58]. It redesigns the format of SSTables for NVM and proposes a new compaction scheme between the first two levels, which they call column-compaction. It also reduces the depth of LSM trees to reduce write amplification.

Deploying LSM-KVs on New Devices: Recent studies also focus on the design of LSM-KVs to fit in new storage devices, especially on PM. NoveLSM discusses several possible solutions that accelerate LSM-KV such as storing parts of the persisted component on PM [37]. It also proposes an in-place update solution to replace the compaction jobs in shallower levels. SLM-DB further takes advantage of PM by building a global B+-tree index in PM [36]. This global index helps in organizing the entire DB into a single level and uses selective compaction to reduce write amplification. ListDB deploys the entire LSM-KV in a DRAM-PM only system and considers the NUMA sensitivity of PM and the overhead of multiple copying of commit logs [38]. It develops a NUMA-aware skip list to replace SSTs. This saves the overhead of merge-sorting in Memtables and entry copying commit logs. It also uses in-place updates to reduce bandwidth utilization and the write amplification problem caused by conventional compaction jobs.

Other approaches to improve LSM-KV systems have been proposed. In particular, as new devices with much higher bandwidth and parallelism than conventional devices become prevalent, software overhead becomes more significant, and thus, approaches to reduce this overhead have been made. P2KVS notices the long waiting time on WAL lock [44]. It adds an accessing layer that batches the incoming requests and dispatches them into different KV instances, efficiently increasing the scalability of LSM-KVs on NVMe SSD. Studies such as KVell [43] and SpanDB [11] notice the high software overhead in conventional IO interfaces and replace the interfaces with more efficient ones like libasync or SPDK [57]. To further eliminate the high IO stack overhead, some of the studies try to reduce the duplicated operations between devices and the LSM-KV. FlashKV and LOCS use Open Channel SSDs (OCSSD) to directly control the IO process of LSM-KVs from the user-level [53, 59]. Other studies like KVSSD and iLSM try to integrate LSM-KVs and the FTL (Flash Translation Layer) of SSDs to bypass the IO stack and achieve lower operation latency [42, 55].

Parameter Tuning of LSM-KVs: There are studies that notice the performance of LSM-KVs can be strongly influenced by the configuration setting. Monkey extrapolates the worst-case scenarios for various operations and designs a configuration tuning framework to tune memory allocation policies and read/write performance for specific workloads based on these scenarios [13]. Dostoevsky further discusses the impact of different compaction strategies and develops a mixed compaction strategy that determines the input according to the input level [14]. Rafiki [47] and TiKV [21] use offline training methods such as Deep Neural Networks (DNN) to study the best setting combination according to the performance of the entire workload. They achieve better throughput but are limited to workload characteristics. Endure concludes that the tuning method on the worst cases is the Nominal Tuning Problem and provides a system that uses a robust tuning method to improve the tuning effect when facing uncertain workloads [22].

8 Conclusion

In this paper, we studied the write stall phenomena in LSM-KVs by revisiting earlier studies. We showed that the conclusions that focus on the individual aspects, though valid, are not generally applicable. Through a thorough review and further experiments on a modern LSM-KV, we showed that data overflow, which refers to the rapid expansion of one or more components in an LSM-KV system due to a surge in data flow into one of the components, is able to explain the formation of write stalls. Our contention was that by balancing and harmonizing data flow among components, we will be able to reduce data overflow and thus, write stalls.

We proposed a tuning framework called ADOC (Automatic Data Overflow Control) to adjust the system configurations rather than simply waiting for the overflowed data to be consumed as is done by default in RocksDB. Experimental results with RocksDB showed that ADOC improves throughput by as much as 322.8% compared with the auto-tuned RocksDB, which takes a similar auto-tuning approach to ADOC. Compared to the manually optimized state-of-the-art SILK [4], ADOC achieves up to 66% higher throughput for the synthetic write-intensive workloads, while achieving comparable performance for the real-world YCSB workloads. However, SILK attains this performance at the expense of using 22.2% more main memory on average.

Acknowledgement

We would like to thank our shepherd Sudarsun Kannan and the anonymous reviewers for their constructive comments that helped improve the paper. This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2022-2021-0-01817) supervised by the IITP (Institute for Information Communications Technology Planning Evaluation), and also partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CityU 11217020).

References

[1] Auto-tuned rate limiter. http://rocksdb.org/blog/2017/12/18/17-auto-tuned-rate-limiter.html. Accessed on 2022-12-29.

[2] fio - flexible i/o tester. https://fio.readthedocs.io/en/latest/fio_doc.html. Accessed on 2022-01-07.

[3] Github - supermt/feat_7.11. https://github.com/supermt/FEAT_7.11. Accessed on 2022-01-07.

[4] HyperLevelDB: A fork of LevelDB intended to meet the needs of HyperDex while remaining compatible with LevelDB. https://github.com/rescrv/HyperLevelDB. Accessed on 2022-09-09.

[5] Perf wiki. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed on 2022-12-29.

[6] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 363–375, 2017.

[7] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 753–766, 2019.

[8] Atri Bhattacharyya, Uros Tesic, and Mathias Payer. Midas: Systematic Kernel TOCTTOU Protection. In *Proceedings of 31st USENIX Security Symposium (USENIX Security 22)*, pages 107–124, Boston, MA, August 2022. USENIX Association.

[9] Lin Cai, Xuemin Shen, Jianping Pan, and Jon W. Mark. Performance Analysis of TCP-friendly AIMD Algorithms for Multimedia Applications. *IEEE Transactions on Multimedia*, 7(2):339–355, 2005.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[11] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *Proceedings of 19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 17–32, 2021.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. page 143–154, New York, NY, USA, 2010.

[13] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, page 79–94, New York, NY, USA, 2017.

[14] 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*, page 505–520, New York, NY, USA, 2018.

[15] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *Proceedings of 19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 33–49, 2021.

[16] Facebook. Leveled Compaction. https://github.com/facebook/rocksdb/wiki/Leveled-Compaction. Accessed on 2022-09-09.

[17] Sanjay Ghemawat and Jeff Dean. Google: Leveldb. https://github.com/google/leveldb, 2022. Accessed on 2022-09-09.

[18] Brendan Gregg. The flame graph: This visualization of software execution is a new necessity for performance profiling and debugging. *Queue*, 14(2):91–110, mar 2016.

[19] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. Understanding the idiosyncrasies of real persistent memory. *Proc. VLDB Endow.*, 14(4):626–639, feb 2021.

[20] Tom's Hardware. Intel Kills Optane Memory Business, Pays $559 Million Inventory Write-Off. https://www.tomshardware.com/news/intel-kills-optane-memory-business-for-good. Accessed on 2022-09-09.

[21] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: A Raft-Based HTAP Database. *Proc. VLDB Endow.*, 13(12):3072–3084, aug 2020.

[22] Andy Huynh, Harshal A. Chaudhari, Evimaria Terzi, and Manos Athanassoulis. Endure: A Robust Tuning Paradigm for LSM Trees Under Workload Uncertainty, 2021.

[23] Facebook Inc. Benchmarking tools | RocksDB. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools. Accessed on 2022-09-09.

[24] Facebook Inc. facebook/rocksdb. https://github.com/facebook/rocksdb/tree/v6.11.4. Accessed on 2022-09-09.

[25] Facebook Inc. RocksDB | A Persistent Key-Value store | RocksDB. https://rocksdb.org/. Accessed on 2022-09-09.

[26] Facebook Inc. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide. Accessed on 2022-09-09.

[27] Facebook Inc. rocksdb/HISTORY.MD. https://github.com/facebook/rocksdb/blob/main/HISTORY.md. Accessed on 2022-09-09.

[28] Facebook Inc. Write Stalls. https://github.com/facebook/rocksdb/wiki/Write-Stalls. Accessed on 2022-09-09.

[29] Intel Inc. Intel SSD DC S4500 Series 960GB 2.5in SATA 6Gbs 3D1 TLC Product Specifications. https://ark.intel.com/content/www/us/en/ark/products/series/120535/intel-ssd-dc-s4500-series.html. Accessed on 2022-09-09.

[30] Samsung Inc. 970 PRO | Consumer SSD | Samsung Semiconductor. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/970pro/. Accessed on 2022-09-09.

[31] Seagate Inc. BarraCuda Hard Drives. https://www.seagate.com/products/hard-drives/barracuda-hard-drive/, 2022. Accessed on 2022-09-09.

[32] InfluxData. InfluxDB: Purpose-Built Open Source Time Series Database | InfluxData. https://www.influxdata.com/. Accessed on 2022-09-09.

[33] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Soh, Zixuan Wang, Yi Xu, Subramanya Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. https://arxiv.org/abs/1903.05714, 03 2019.

[34] Yichen Jia and Feng Chen. From Flash to 3D XPoint: Performance Bottlenecks and Potentials in RocksDB with Storage Evolution. In *Proceedings of 2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 192–201, 2020.

[35] Myoungsoo Jung. Hello Bytes, Bye Blocks: PCIe Storage Meets Compute Express Link for Memory Expansion (CXL-SSD). HotStorage '22, page 45–51, 2022.

[36] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. SLM-DB: Single-Level Key-Value store with persistent memory. In *Proceedings of 17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, 2019.

[37] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of 2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, 2018.

[38] 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.

[39] Andrew Kryczka. Auto-tuned Rate Limiter | RocksDB. https://rocksdb.org/blog/2017/12/18/17-auto-tuned-rate-limiter.html. Accessed on 2022-09-09.

[40] Chengdi Lai, Ka-Cheong Leung, and Victor O.K. Li. Design and analysis of TCP AIMD in wireless networks. In *Proceedings of Wireless Communications and Networking Conference (WCNC)*, pages 1422–1427, 2013.

[41] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, apr 2010.

[42] Chang-Gyu Lee, Hyeongu Kang, Donggyu Park, Sungyong Park, Youngjae Kim, Jungki Noh, Woosuk Chung, and Kyoung Park. ilsm-ssd: An intelligent lsm-tree based key-value ssd for data analytics. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 384–395, 2019.

[43] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: the Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the*

27th ACM Symposium on Operating Systems Principles, pages 447–461, 2019.

[44] Ziyi Lu, Qiang Cao, Hong Jiang, Shucheng Wang, and Yuanyuan Dong. P²KVS: A Portable 2-Dimensional Parallelizing Framework to Improve Scalability of Key-Value Stores on SSDs. page 575–591, 2022.

[45] Chen Luo and Michael J Carey. On performance stability in LSM-based storage systems (extended version). https://arxiv.org/abs/1906.09667, 2019.

[46] Chen Luo and Michael J Carey. LSM-based storage techniques: a survey. *The VLDB Journal*, 29(1):393–418, 2020.

[47] Ashraf Mahgoub, Paul Wood, Sachandhan Ganesh, Subrata Mitra, Wolfgang Gerlach, Travis Harrison, Folker Meyer, Ananth Grama, Saurabh Bagchi, and Somali Chaterji. Rafiki: A Middleware for Parameter Tuning of NoSQL Datastores for Dynamic Metagenomics Workloads. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference(Middleware '17)*, page 28–40, 2017.

[48] Yoshinori Matsunobu, Siying Dong, and Herman Lee. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proceedings of the VLDB Endowment*, 13(12):3217–3230, 2020.

[49] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles(SOSP '17)*, page 497–514, 2017.

[50] 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*, pages 217–228, 2012.

[51] Xuan Sun, Jinghuan Yu, Zimeng Zhou, and Chun Jason Xue. FPGA-based Compaction Engine for Accelerating LSM-tree Key-Value Stores. In *Proceedings of 2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1261–1272, 2020.

[52] Mehul Nalin Vora. Hadoop-HBase for large-scale data. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, pages 601–605, 2011.

[53] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[54] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Towards an Unwritten Contract of Intel Optane SSD. In *Proceedings of 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[55] Sung-Ming Wu, Kai-Hsiang Lin, and Li-Pin Chang. KVSSD: Close integration of LSM trees and flash translation layer for write-efficient KV store. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 563–568, 2018.

[56] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of 18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, 2020.

[57] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. Spdk: A development kit to build high performance storage applications. In *Proceedings of 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161, 2017.

[58] 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 Matrix Container in NVM. In *Proceedings of 2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 17–31, 2020.

[59] Jiacheng Zhang, Youyou Lu, Jiwu Shu, and Xiongjun Qin. Flashkv: Accelerating kv performance with open-channel ssds. *ACM Trans. Embed. Comput. Syst.*, 16(5s), sep 2017.

[60] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, Zhongdong Huang, and Jianling Sun. FPGA-Accelerated Compactions for LSM-based Key-Value Store. In *Proceedings of 18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 225–237, Santa Clara, CA, 2020.

[61] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. In *Proceedings of 2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 655–672, Carlsbad, CA, July 2022. USENIX Association.