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

21st USENIX Conference on File and Storage Technologies

Jinghuan Yu; Sam H. Noh; Young-ri Choi; Chun Jason Xue

Paper Notes

By JeongHa Lee

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.

Problem Statement and Research Objectives

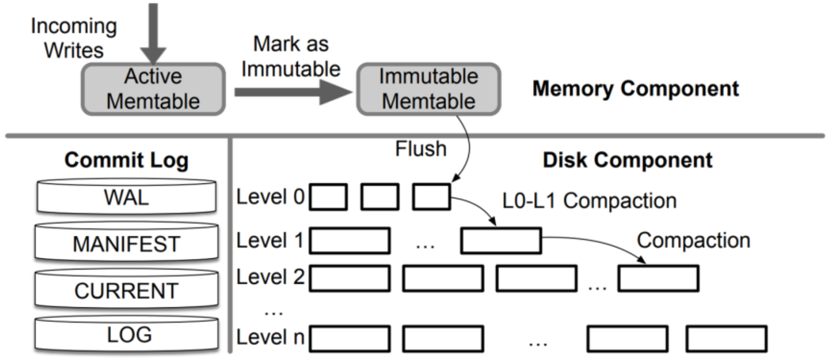


Figure 2: Architecture of RocksDB [15, 25, 48].

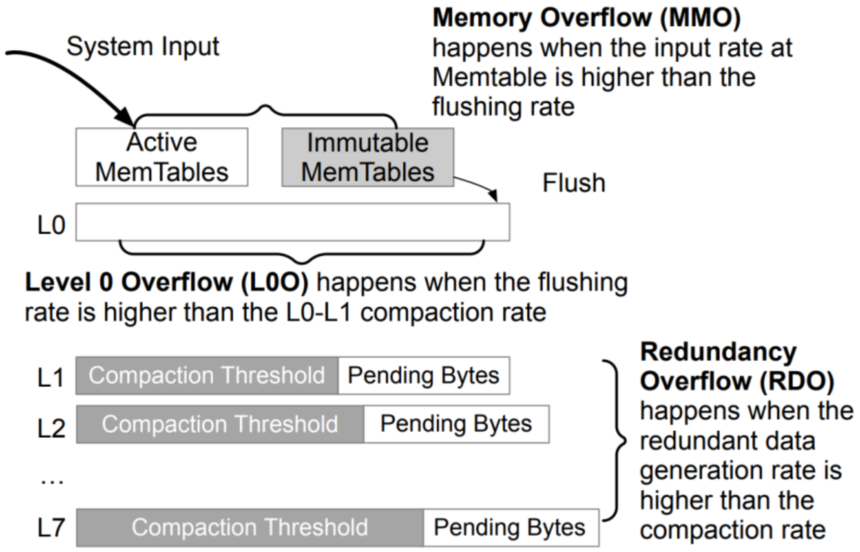


Figure 9: Data overflow scenarios in modern LSM-KVs.

There are two types of background data movement jobs.

- **Flush** moves the Immutable Memtable from the **memory component** to the disk component **turning it into an SSTable in Level 0**.
- **Compaction** is triggered to merge SSTables in this level with SSTables in the next deeper level.
  - when the capacity of a level reaches a certain threshold
  - L0-L1 compaction 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, deeper level compactions can occur in parallel.

Problem Statement and Research Objectives

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.

Proposed Method

Dataflow is **controlled by online tuning of the number of threads and the batch size**, and most LSM-KVs provide APIs to adjust these two values without rebooting the system.

1. **Device transparency** : Instead of targeting optimizations to a particular storage device, 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**
     ➔ Then, the thread count and batch size are adjusted

2. **Ease of portability** : ADOC does not disrupt the internal architecture of the LSM-KV system making it highly portable.
   - In the current RocksDB implementation, ADOC requires only minimal modifications—limited to two classes, with 250 lines of code (LOC) for the tuner and 50 LOC for collecting system states.
   - **'Options' class** : controls whether the ADOC tuner will be enabled or not and records the instantaneous information of the system in a shared C++ vector.
   - **'tuner' class** : periodically wake the tuner threads to perform tuning actions.

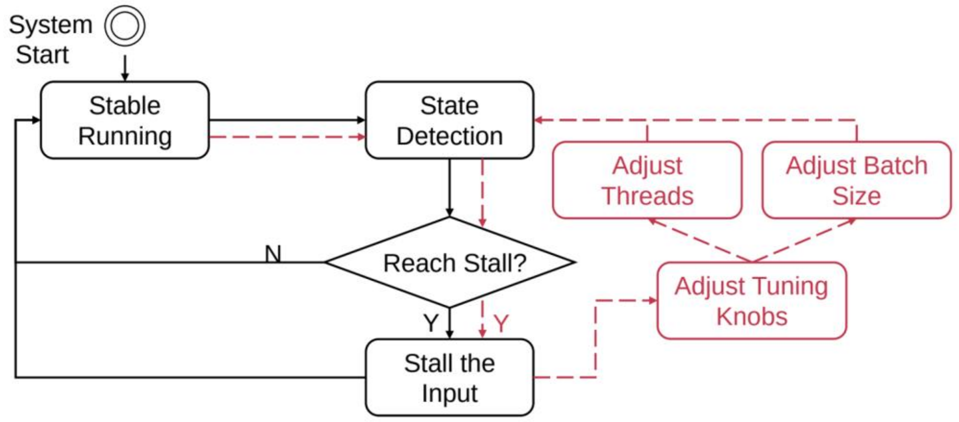
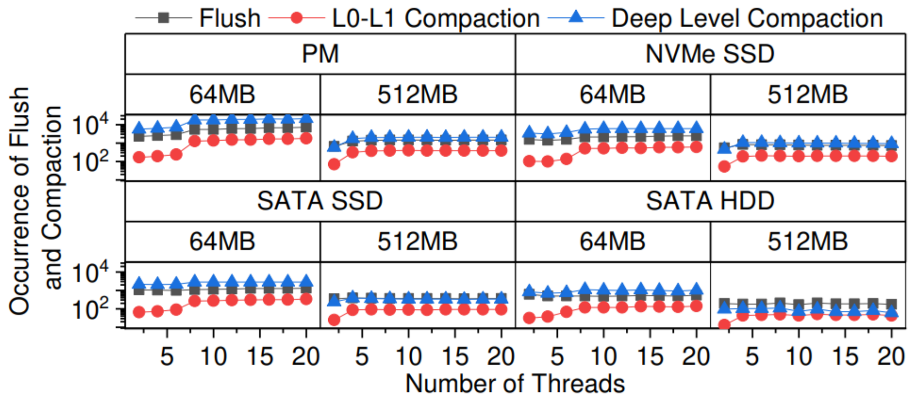
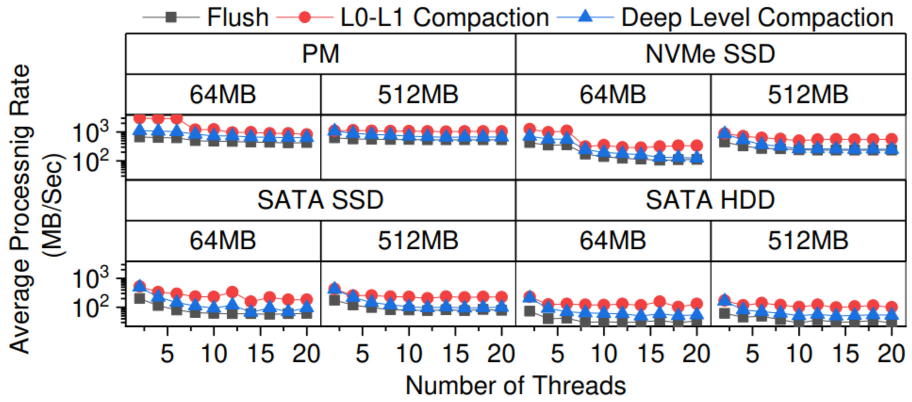


Figure 11: The solid black diagrams show the default control flow of RocksDB. ADOC extends this with the red diagrams (including the dashed arrows) to adjust tuning knobs during execution.

Proposed Method



(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.

- Figure 7(b) shows that **flush jobs are being allocated the least bandwidth among the background jobs**.
  - 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.

Proposed Method

For every time window T_w , ADOC monitors for data overflow and takes action. we set T_w to one second.

- values larger : not agile enough to quickly detect the overflows for state-of-the-art high-performing storage devices
- values smaller : could incur overhead as well as lead to fluctuations due to responding too quickly.

	ADOC determines…	# of threads	batch size
MMO	when the active Memtable is filled before the Immutable Memtable gets flushed.	reduce - to increase flush rate	increase - to increase the processing rate
L0O	same logic as RocksDB: when the number of L0 files exceeds the threshold	increase - improving the chance of L0- L1 compaction being assigned a thread - decreasing the flush rate to ease the overflow	unchanged - increasing it will increase the load on L0-L1 compaction - decreasing it will generate more L0 files
RDO	same logic as RocksDB: when the total redundant data size exceeds the threshold	increase - to increase the rate of deep level compaction - to reduce flush rate	decrease - to allow the scheduler to generate more fine-grained compaction jobs as small

Evaluation and Results

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

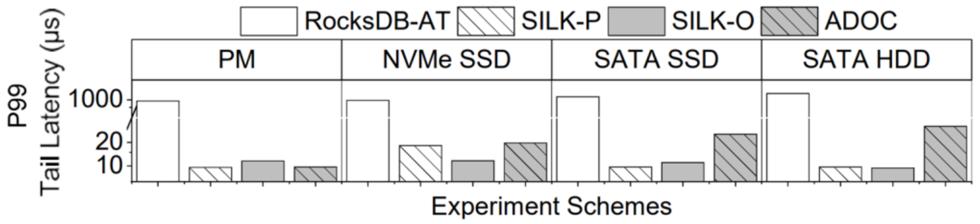


Figure 16: Average 99^{th} tail latency in *fillrandom* workload.

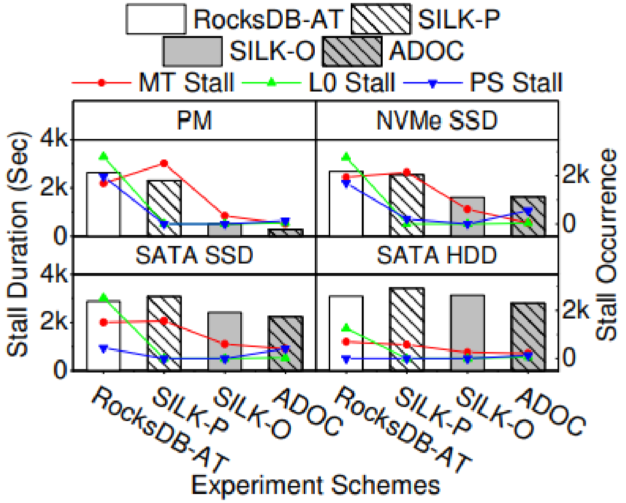


Figure 13: Stall duration (bar) and occurrences (lines).

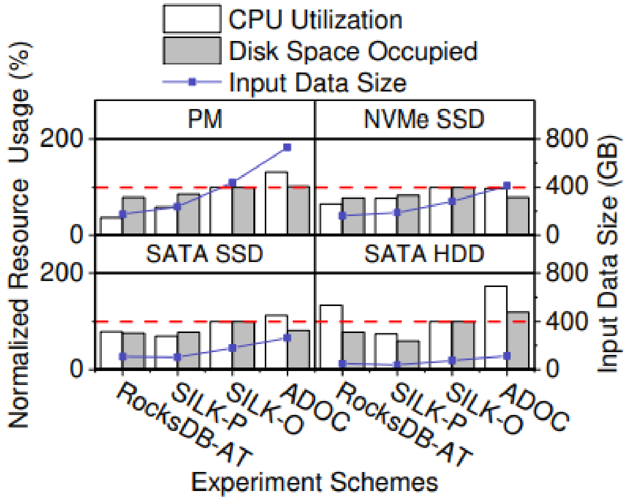


Figure 14: Comparison of CPU utilization, disk space occupied, and input data size with the *fillrandom* workload.

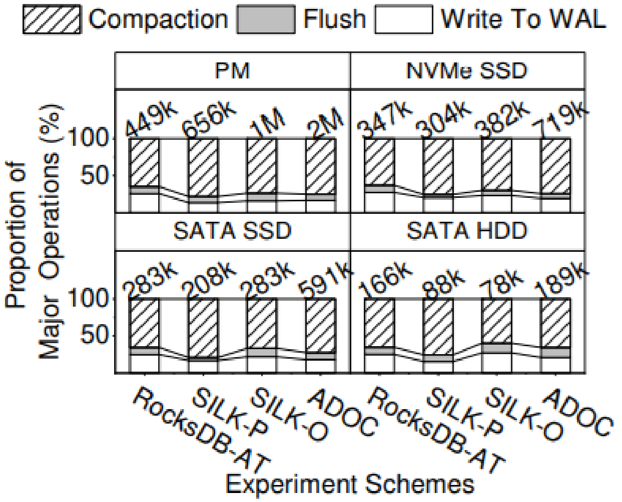


Figure 15: Proportion of major operation occurrences, with numbers representing total occurrences.

Evaluation and Results

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

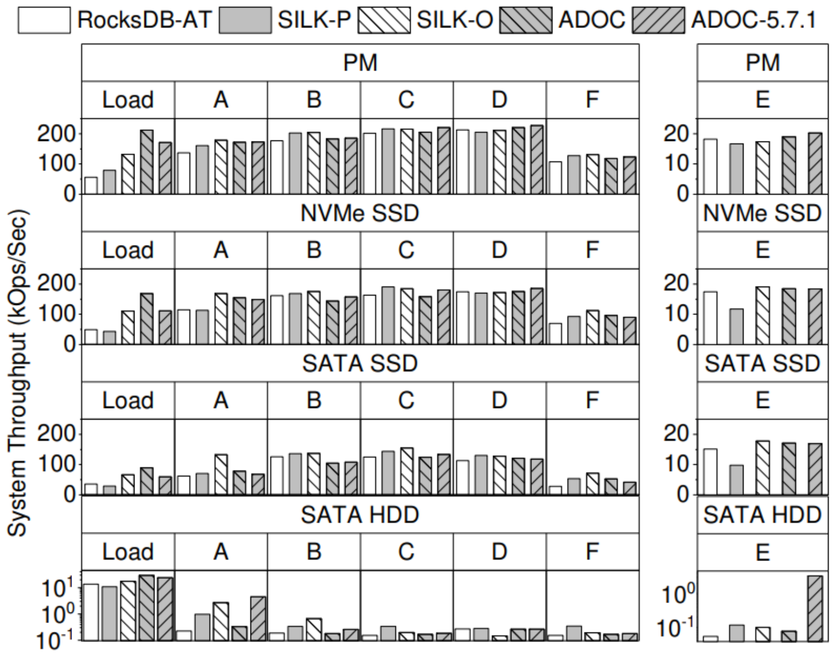


Figure 17: Comparison of system throughput in different stages of YCSB workloads.

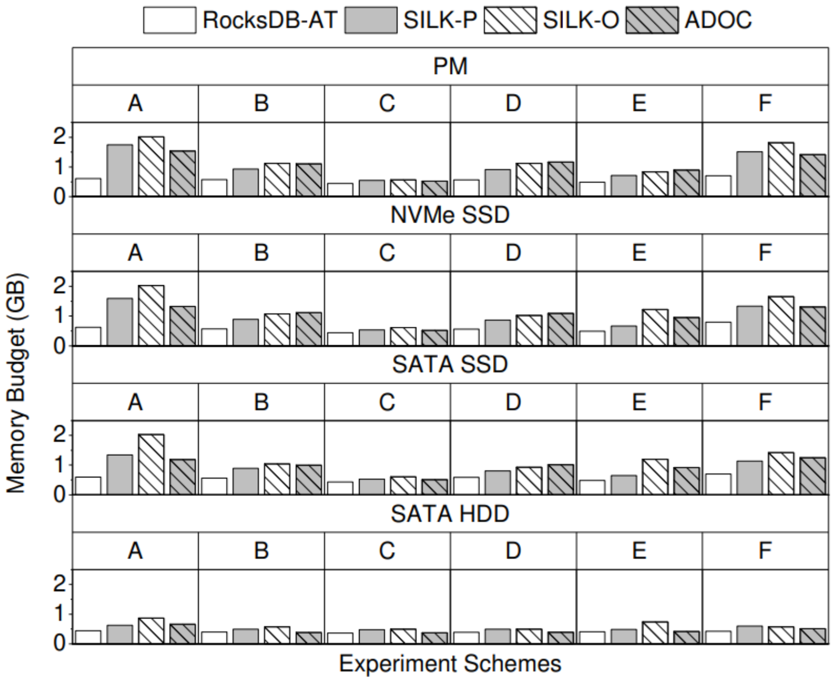


Figure 18: Comparison of main memory footprint.

Notes

1 : https://en.wikipedia.org/wiki/Key%E2%80%93value_database
2 : https://dev.to/creativcoder/what-is-a-lsm-tree-3d75

- A key–value database, or key–value store, is a data storage paradigm designed for storing, retrieving, and managing associative arrays, and a data structure more commonly known today as a dictionary or hash table.[1]
- LSM Tree is a data structure that restricts your datastore to append-only operations.[2]
    - **LSM Tree** : Being append-only, it achieves high write throughput and supports low-cost reads via indexes maintained in RAM.
    - **B+ Tree** : Performs in-place updates, which can lead to random I/Os.
- 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.
- 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.