NobLSM: An LSM-tree with Non-blocking Writes for SSDs

Haoran Dang, Chongnan Ye, Yanpeng Hu, and Chundong Wang*
School of Information Science and Technology, ShanghaiTech University, China
{danghr, yechn, huyp, wangchd}@shanghaitech.edu.cn

ABSTRACT

Solid-state drives (SSDs) are gaining popularity. Meanwhile, key-value stores built on log-structured merge-tree (LSM-tree) are widely deployed for data management. LSM-tree frequently calls syncs to persist newly-generated files for crash consistency. The blocking syncs are costly for performance. We revisit the necessity of syncs for LSM-tree. We find that Ext4 journaling embraces asynchronous commits to implicitly persist files. Hence, we design NobLSM that makes LSM-tree and Ext4 cooperate to substitute most syncs with non-blocking asynchronous commits, without losing consistency. Experiments show that NobLSM significantly outperforms state-of-the-art LSM-trees with higher throughput on an ordinary SSD.

CCS CONCEPTS

• **Software and its engineering → File systems management**;
• **Information systems → Key-value stores**.

KEYWORDS

LSM-tree, Key-Value Store, Fsync, Asynchronous Commit

1 INTRODUCTION

Solid-state drives (SSDs) are gaining wide popularity for storage [1–3]. Computer scientists have ported critical data management systems into SSDs, including the key-value (KV) store built on log-structured merge tree (LSM-tree) [4–12]. LSM-tree employs a *log* to back up arriving KV pairs before putting them into an in-memory mutable *memtable*. A full memtable is made immutable and LSM-tree creates a new memtable along with an empty log. LSM-tree also has several on-disk levels (L_n, $n \geq 0$) for persistent storage. Each level contains *SSTable* files composed of KV pairs. When LSM-tree uses up the memtable space, it does a *minor compaction* that converts an immutable memtable to an L_0 SSTable. LSM-tree defines a fixed capacity limit for any L_n, which is usually one tenth of the capacity limit of L_{n+1}, thereby resembling a tiered tree-like structure. A full L_n triggers a *major compaction*: LSM-tree selects a few L_n and L_{n+1} SSTables, merge-sorts KV pairs stored in them, and orderly packs sorted KV pairs into new L_{n+1} SSTables.

To rule out inconsistency due to a failed major compaction, LSM-tree enforces a strict order in handling new and old SSTables. In short, only after persisting new SSTables via syncs (`fsync` or `fdatasync`), can LSM-tree delete old ones that have been kept for backup in case of a sudden crash. However, sync *is a blocking write operation that persists data directly with storage device and enforces a barrier to stall subsequent I/O operations until it completes* [1, 13, 14].

The cost of sync is severe for LSM-trees since it inflicts stalls on the critical path of LSM-tree operations [4, 10, 12, 14]. For example, LevelDB, a typical LSM-tree-based KV store [15], would suspend in the foreground if a major compaction between L_0 and L_1 has not finished while the immutable memtable is waiting for conversion. With LevelDB we quantitatively study the impact of syncs. We record the execution time of randomly putting 10 million 1KB-sized KV pairs within the original LevelDB and a 'volatile' LevelDB with all syncs disabled. The volatile LevelDB reduces the execution time by 53.2% compared to the original one. Hence, albeit losing consistency, the volatile LevelDB achieves superb performance. The aim of this paper, however, is to develop an LSM-tree variant that preserves crash consistency but minimizes the use of syncs.

Not much research work has been done to particularly mitigate the cost of syncs for LSM-trees. He et al. [1] reported that syncs hinder LevelDB and RocksDB [16] from efficiently utilizing the bandwidth of SSD. Kim et al. [14] designed BoLT that tries to reduce the barrier effect of syncs for LSM-trees by outputting one large factual SSTable per compaction and flushing KV pairs grouped in the large file via one sync. Yet whenever KV pairs are involved in a future compaction, BoLT must call sync again to re-persist them.

The design proposed by this paper, nonetheless, mainly relies on *non-blocking writes* and syncs each KV pair *only once* for crash consistency. It is named NobLSM (**no**n-**b**ocking **LSM**-tree) that seeks aid from the file system for the minimum use of syncs. LSM-tree stores SSTables as files via an underlying file system, which has to guarantee system-level crash consistency [2, 3, 13, 17]. The prevalent journaling file system Ext4, in its default `data=ordered` mode, secures the consistency of file system metadata (e.g., `inodes`) by orderly committing the `inode` of a file into a journal after persistently writing back the file's modified data. Ext4 initiates a commit either synchronously due to an explicit call of sync, or asynchronously, i.e., every a period or upon the shortage of DRAM page cache [13, 17]. Once a commit succeeds, Ext4 guarantees that both metadata and data of a committed file are crash-recoverable. Consequently, if we can precisely track when Ext4 commits new SSTables generated for a major compaction, we do not bother to forcefully sync them with substantial performance penalties. This is the essence of NobLSM.

The main ideas of NobLSM are summarized as follows.

- *NobLSM yields a crash-consistent LSM-tree without using* syncs *on the critical path of major compaction.* We enhance Ext4 with two system calls (syscalls) based on its asynchronous commit mechanism, for NobLSM to perceive the durability of new SSTables. NobLSM abides by a strict order of persistently committing new SSTables prior to removing old ones for crash consistency.
- *NobLSM syncs KV pairs packed in an L_0 SSTable only once* when it makes the L_0 SSTable from an immutable memtable during a

*Corresponding author (wangchd@shanghaitech.edu.cn)

DAC '22, July 10–14, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9142-9/22/07.
https://doi.org/10.1145/3489517.3530470

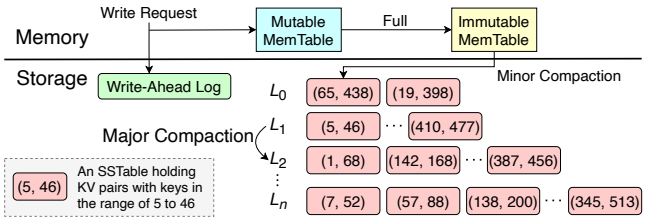
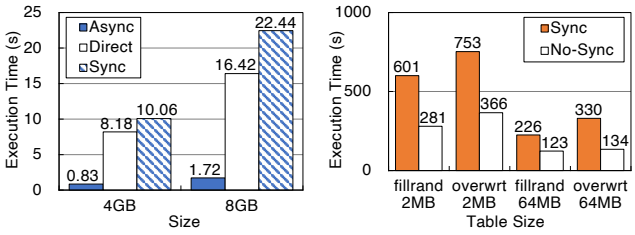


Figure 1: An Illustration of LSM-tree Architecture



(a) Execution time of Async, Direct and Sync

(b) Impact of SSTable size and syncs on execution time

Figure 2: An Illustration of Testing the Impact of syncs

minor compaction. While waiting for Ext4 to asynchronously commit new SSTables, NobLSM transiently retains compacted old SSTables to back up new ones after a major compaction.

- *NobLSM deprives compacted SSTables of serving search requests and discards them at a proper time*, as they are short-lived and volatile after the major compaction.

NobLSM is a simplistic approach that avoids incurring many changes to Ext4 and LSM-tree. We build a prototype for it on LevelDB and Ext4 with just about 330 lines of code in all. However, extensive experiments confirm that NobLSM significantly boosts the performance of LevelDB with about halved execution time for write-intensive workloads, while achieving the same crash consistency. It also generally outperforms state-of-the-art LSM-trees including BoLT. These justify the efficacy and efficiency of NobLSM.

The rest of this paper is as follows. We briefly show LSM-tree, Ext4, and syncs in Section 2. We illustrate a motivational study in Section 3. In Section 4, we detail the design of NobLSM. We analyze evaluation results in Section 5 and conclude the paper in Section 6.

2 BACKGROUND

2.1 Crash Consistency of LSM-Tree

Figure 1 illustrates the architecture of typical LSM-trees. As mentioned, LSM-tree maintains in-memory memtables to absorb arriving KV pairs, and multiple on-disk levels (L_n, $n \geq 0$) to persistently store KV pairs. LSM-tree leverages a write-ahead log to record arriving KV pairs before putting them into memtable. Once KV pairs are dumped into L_0, LSM-tree secures their consistency through backup copies and syncs. After a minor compaction converting an immutable memtable to an L_0 SSTable, LSM-tree persists the SSTable by calling sync before the deletion of corresponding log. As to a major compaction between L_n and L_{n+1}, LSM-tree first generates and syncs new L_{n+1} SSTables. Then it safely removes compacted L_n and L_{n+1} SSTables. To sum up, LSM-tree continually calls syncs to persist KV pairs during both minor and major compactions. Furthermore, whenever a KV pair is involved in a future compaction, LSM-tree would sync it again.

2.2 Blocking sync and Ext4 Journaling

Many KV stores rely on syncs (`fsync` or `fdatasync`) for durability and consistency [13, 18]. In Linux, LSM-trees like LevelDB by default call `fdatasync` that functions almost the same as `fsync` [13, 15]. The impact of syncs on performance is detrimental. A sync not only persists data forcefully with storage device, but also imposes an ordering barrier that blocks subsequent I/O operations [13, 14, 18].

The sync is performed by an underlying file system, which secures system-level crash consistency to support applications. Ext4, as a prevalent file system leveraging JBD2 for journaling [2, 13, 17], guarantees the consistency of file system metadata, e.g., the `inodes` of files, and enforces a writing order of file metadata and data in its default `data=ordered` mode. When data blocks of a file named, for example, 'dac22.txt' are modified, Ext4 commits the metadata block holding the file's `inode` into a persistent journal. Before doing so, Ext4 demands that modified data blocks of 'dac22.txt' must have been written back to storage device. Hence, once an `inode` is committed, all data blocks related to it should be already persisted.

3 MOTIVATION

SSDs are widely deployed today. Concerning the necessity and cost of syncs, we aim to reduce the use of them for LSM-tree ported into SSDs but without losing crash consistency. Previous studies have explored how to alleviate the impact of syncs in the perspective of system or applications. For instance, Ext4 is being enhanced with a feature called *fast commit* in line with the idea of iJournaling [13, 19]. This paper, however, considers a collaborative practice across system (Ext4) and application (LSM-tree) to reduce the use of syncs.

The Impact of syncs *on SSD.* A sync makes applications bypass DRAM page cache and render data durable directly with the storage device. Worse, it also hinders subsequent operations from proceeding [1, 14]. We measure the cost of syncs on SSDs with three writing strategies, i.e., Async (asynchronous write), Direct (write via direct I/O), and Sync (write with sync). We successively generate 4GB and 8GB data in mutliple files, with 2MB per file, same as LevelDB's default SSTable size. We write these files to an ordinary SSD mounted as Ext4. As shown by Figure 2a, the execution time increases by 9.5× from Async to Direct, and further deteriorates by 36.7% to Sync. The overall performance drop of 13.0× from Async to Sync clearly indicates that **the impact of syncs is significant on SSDs**. If we could remove most syncs for LSM-trees, we should approach the high performance of Async.

The Impact of syncs *on LSM-tree.* We conduct another test to observe the actual impact of syncs on real LSM-trees. We disable all syncs for LevelDB, thereby rendering it 'volatile' without consistency guarantee. Yet it yields the optimal performance we try to reach by reducing the use of syncs. We use the micro-benchmark db_bench embedded in LevelDB to randomly put 10 million 1KB-sized KV pairs (fillrandom) and then overwrite them (overwrite). As shown in the left half of Figure 2a, without sync, the execution time shortens by 53.2% and 51.4%, respectively, for fillrandom and overwrite with 2MB SSTables. Hence, **the frequent, repeated syncing of KV pairs badly degrades the performance of LSM-trees**.

The Impact of Different Sizes for SSTable. Besides 2MB, we also test with 64MB SSTables. 64MB is the default size for RocksDB [16] and also used by Kim et al. [14] to verify the impact of syncs. As shown by Figure 2b, from 2MB to 64MB, the execution time of original LevelDB decreases by 62.4% and 56.2% for fillrandom and overwrite, respectively. Larger SSTable leads to aggregated flush with fewer barriers. Though, even with 64MB SSTables, employing syncs still degrades performance by 45.6% and 59.4%, respectively, for two workloads. Thus, **using large SSTables alone without a substantial reduction of syncs cannot fully mitigate the catastrophic cost of syncs**. In this paper, we take a holistic approach and develop NobLSM that substitutes most syncs with asynchronous commit implicitly embraced by Ext4 journaling.

4 DESIGN OF NOBLSM

NobLSM is made by a simplistic but effectual collaboration between LSM-tree and Ext4. It mainly relies on non-blocking writes for crash consistency and uses syncs only once when converting an immutable memtable to L_0 SSTable in a minor compaction. As to a major compaction, NobLSM takes advantage of the asynchronous journal commit of Ext4, rather than syncs, to implicitly persist new SSTables. On the success of commit, NobLSM safely removes old SSTables for SSD space efficiency. It demands a holistic cooperation between LSM-tree and Ext4 across user- and kernel-spaces. By doing so, NobLSM manages to avoid the blocking syncs on the critical path of major compaction, thereby improving performance and achieving consistency simultaneously.

4.1 LSM-tree with Minimum syncs

Write and Read Requests. In the foreground, NobLSM handles `Put`, `Get`, and `Delete` requests the same as conventional LSM-trees. On receiving a `Put` request, NobLSM appends the arriving KV pair to a log and then puts it into the mutable memtable. On a `Get` request, NobLSM orderly searches from memtables to in-SSD levels.

Minor Compaction. NobLSM turns a full mutable memtable to be immutable. It initiates a minor compaction upon running out memtable space and dumps an immutable memtable to be an L_0 SSTable. It persists L_0 SSTables by calling sync. This is the only one occasion when NobLSM syncs KV pairs. After syncing, NobLSM removes corresponding log and frees up the immutable memtable.

Major Compaction. In a major compaction between L_n and L_{n+1}, NobLSM merge-sorts all KV pairs from selected SSTables. It then packs them into new L_{n+1} SSTables. NobLSM neither syncs new SSTables nor immediately remove old ones. It asynchronously writes new files. Next it leverages syscalls we have developed with Ext4 to precisely track when new SSTables become durable in SSD (cf. Section 4.2), the effect of which is identical to persisting files via syncs. Consequently, NobLSM eliminates direct I/Os and barriers caused by syncs from the critical path of major compactions.

NobLSM transiently retains old SSTables as backup copies for crash recoverability, since new SSTables are being asynchronously committed. To manage SSTable files, NobLSM maintains *a global pair of sets as predecessors and successors*. Given a major compaction generating q new SSTables from p old ones ($p > 0$, $q > 0$), NobLSM fills them in corresponding sets and tracks the p-to-q dependency mapping in between. Only when all q successors are committed can NobLSM delete p predecessors. The reason of using a global pair of sets is twofold. Firstly, Ext4's asynchronous commit implies temporal uncertainty and NobLSM needs a structure to consecutively delete obsolete SSTables. Secondly, NobLSM's reduction of syncs brings up throughput and in turn entails more compactions. Such a trend needs a global structure to accumulate multiple simultaneous p-to-q mappings. Hence, NobLSM utilizes the mappings between two sets to monitor ongoing and historical major compactions for which Ext4 has not finished committing their new SSTables.

4.2 Ext4 with Asynchronous Commit

The Implication of Ext4 Journaling. Ext4 uses a two-phase protocol of JBD2, i.e., *commit* and *checkpoint*, for journaling [2, 13, 17]. The unit of journaling is a transaction made of multiple modified blocks. Ext4 commits each transaction to a journal and then checkpoints files in place. It initiates a commit either synchronously due to syncs, or asynchronously, i.e., every a period (five seconds by default) or when dirty pages in DRAM page cache reaches a threshold (10% by default), whichever is earlier. File metadata or data is crash-recoverable once successfully committed to the journal.

The change of file data leads to the change of file metadata, particularly the file's `inode`. In its default `data=ordered` mode, Ext4 journals metadata only. Given a file with modified data, `data=ordered` mode forcefully persists data to the file before committing the file's `inode` to journal. As a result, once Ext4 commits a transaction containing a file's `inode`, the file must be durable, since both the file's data and metadata have been successively persisted.

The native Ext4 does not give the information of whether a specific `inode` is committed or not. We enhance Ext4 with structures and interfaces for NobLSM to inquire the commit status of an `inode`.

Two Kernel-space Tables. Ext4 relies on the writeback thread of Linux kernel to commit transactions. Yet Ext4 journaling is globally shared by system and all applications over time. A transaction may consist of `inodes` not belonging to NobLSM. Also, the `inodes` of new SSTables generated in one major compaction are possibly pending in different transactions. To resolve these issues, we maintain two tables in the kernel space for Ext4 to record whether each new SSTable is persisted or not. The *Pending Table* records `inodes` of all new SSTables being tracked by NobLSM while the *Committed Table* holds ones that have been committed. On the completion of committing a transaction, Ext4 journaling transfers `inodes` covered by that transaction from *Pending Table* to *Committed Table*.

Two New Syscalls. We add a syscall (`check_commit`) for NobLSM to tell Ext4 what `inodes` the latter shall start tracking. Hence, `check_commit` fills `inodes` addressed by NobLSM in the *Pending Table*. The other syscall (`is_committed`) is for NobLSM to inquire if Ext4 has moved a specific `inode` to *Committed Table*. When NobLSM realizes that all new SSTables are factually committed for a major compaction, it safely removes compacted old ones (cf. Section 4.3).

The two kernel-space tables support the global pair of user-space sets maintained by NobLSM's LSM-tree at runtime. Connected through two syscalls, they jointly function at system and application levels. This systematic design enables NobLSM to efficiently manage SSTables without losing consistency guarantee. Moreover, NobLSM conducts all acts in the background, thereby incurring the minimum cost on the critical path of LSM-tree operations.

4.3 The Management of Compacted SSTables

NobLSM needs to attend two issues in managing compacted SSTables. One is depriving them of serving search requests. The other one is discarding them at a proper time for SSD space efficiency.

Transient Duplicate Copies of KV Pairs. As NobLSM transiently retains both new and old SSTables, there is a moment at which one valid KV pair exists in two SSTables. Conventional LSM-trees maintain an in-memory *Version* backed by a *Manifest* file to track all valid SSTables. As old SSTables are potentially volatile, using them to serve search requests is infeasible. Thus, NobLSM labels them as 'shadow' and directs no search request to them.

Reclamation of Obsolete SSTables. On a successful commit of journaling, NobLSM asks Ext4 to delete relevant shadow SSTables. We set a frequency of every five seconds for NobLSM to inquire Ext4 on the commit of new SSTables via an `is_committed` syscall. Such a match between inquiry interval and Ext4's commit interval reduces unnecessary checks across the user- and kernel-spaces.

Old SSTables used to be committed for past major compactions. Ext4 removes their `inodes` from the *Commtted Table* upon deletion. This gains time and space efficiencies of managing kernel-space tables and avoids inter-table cyclic dependency due to `inode` reuse.

4.4 Crash Consistency of NobLSM

NobLSM logs an arriving KV pair and later persistently stores it via `sync` into an L_0 SSTable during a minor compaction. Any subsequent major compaction between L_n and L_{n+1} ($n \geq 0$) depends on that `sync`. NobLSM guarantees that, since the first time a KV pair is made durable, it would never be lost after a crash. The rationale is that, in a major compaction, NobLSM strictly abides by the order of implicitly persisting new SSTables prior to removing compacted ones. To sum up, NobLSM achieves the same crash consistency as conventional LSM-trees but calls the minimum number of `sync`s.

4.5 Putting Everything Together

Let us use an example shown in Figure 3 to summarize how NobLSM works. NobLSM's LSM-tree selects and compacts SSTables 127 and 123 at L_1 and L_2, respectively, into new L_2 SSTables 230 and 231 (❶). Ext4 asynchronously writes them (❷) and LSM-tree obtains their `inode` numbers #4567 and #4568. LSM-tree asks Ext4 to fill two `inodes` in the *Pending Table* via the syscall `check_commit` (❸) and updates the p-to-q dependency (❹). Simultaneously, Ext4 writes back data into SSD for two new SSTables (❺) and commits two `inodes` alongside a transaction into the journal (❻).

On the success of commit, Ext4 moves `inodes` to the *Committed Table* (❼) so that LSM-tree can perceive the durability of two new SSTables via the syscall `is_committed` (❽). Next, LSM-tree deletes obsolete SSTables 123 and 127 (❾). Ext4 accordingly removes two files and erases corresponding entries in the *Committed Table* (❿).

5 EVALUATION

5.1 Evaluation Setup

Platform. We run all experiments on a server with two Intel Xeon Gold 6342 processors and 2TB DRAM memory. The operating system is Ubuntu 20.04.3 with Linux kernel 5.14.7, and the compiler is GCC/G++ 9.3.0. All LSM-trees store data into a 960GB Samsung

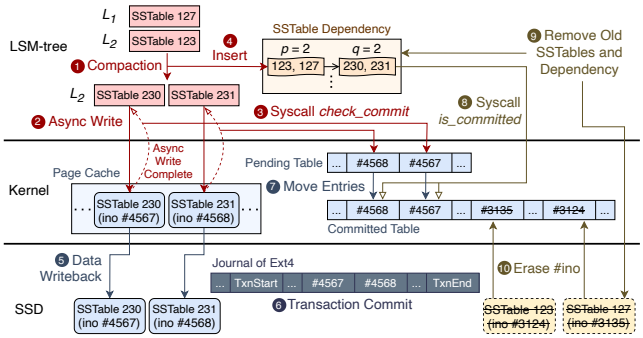


Figure 3: An Example of NobLSM's Major Compaction

PM883 SSD formatted and mounted in the `data=ordered` mode of original Ext4 except that NobLSM uses its customized version.

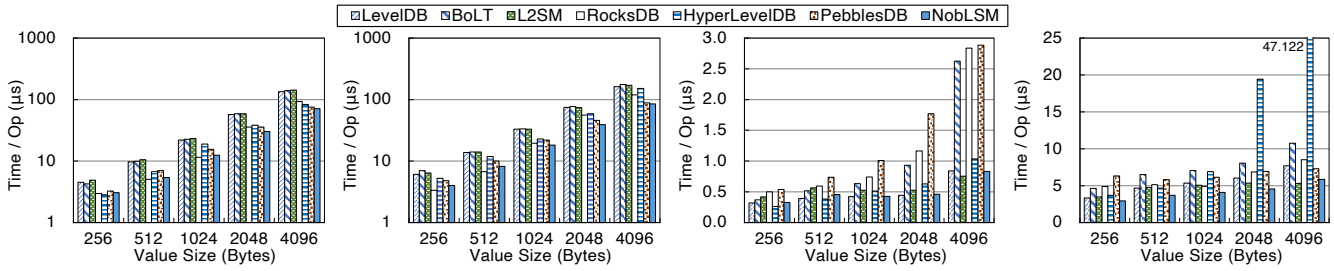
Competitors. We have implemented a prototype of NobLSM based on LevelDB (version 1.23) and Linux kernel 5.14.7 with about 200 and 130 lines of code (LOC) added or changed, respectively. We compare it to original LevelDB, BoLT (based on LevelDB), L2SM [20], HyperLevelDB [21], PebblesDB [22], and RocksDB [16]. They represent different approaches to reduce the performance cost of LSM-trees, especially in the process of compactions, without losing consistency. BoLT outputs a large factual SSTable per compaction so as to `sync` the bundled KV pairs only once. L2SM handles hot KV pairs separately from cold ones, as the former are more frequently updated and likely to inflict unnecessary I/Os in compactions. RocksDB and HyperLevelDB consider fine-grained and parallelized tactics for acceleration, e.g., multi-threaded compactions. PebblesDB segments each level in non-overlapping key ranges and writes KV pairs to the appropriate range of a target level for only once. We note that all these LSM-trees sync a KV pair over again whenever the KV pair is selected for future compactions. Also they mainly embrace more complex designs than NobLSM.

Benchmarks & Setting. We use db_bench and YCSB [23], respectively, as micro- and macro-benchmarks. We compile all LSM-trees under the Release mode. We set the SSTable in 64MB for all LSM-trees (cf. Section 3) unless some of them, like HyperLevelDB, hardcode sizes in source codes. The performance metric is average execution time per operation. Less time means higher throughput.

5.2 Micro-benchmark (db_bench)

We use db_bench's four workloads, i.e., fillrandom (random write), overwrite (random update), readseq (sequential iteration of all KV pairs), and readrandom (random read of one KV pair). They represent access patterns commonly found in realistic workloads. For each workload, db_bench issues overall 10 million requests. Within each request, we configure keys in a length of 16 bytes while varying the value size to be 256B, 512B, 1KB, 2KB, and 4KB.

Write Performance. Figure 4a and Figure 4b show the average execution time for fillrandom and overwrite, respectively, in the logarithmic scale. Take 1KB value size for example. Built on LevelDB, NobLSM reduces the execution time with fillrandom by up to 43.6%, which is close to the reduction ratio of 45.6% achieved by the volatile LevelDB without consistency (cf. Section 3 and Figure 2b). With fillrandom and 2KB values, NobLSM makes the highest boost over LevelDB, by up to 47.1%. A similar performance improvement



(a) fillrandom (Log scale) **(b) overwrite (Log scale)** **(c) readseq** **(d) readrandom**

Figure 4: A Comparison between Seven LSM-tree Variants with Write and Read Workloads of Micro-benchmarks

Table 1: No. of syncs and Size of Data Synced for LSM-trees

LSM-tree	LevelDB	BoLT	L2SM	RocksDB	Hyper-LevelDB	PebblesDB	NobLSM
No. of syncs	1,061	659	1,046	606	2,684	713	160
Size (GB)	61.55	55.15	60.98	35.82	47.43	42.61	9.82

is also obtained between NobLSM and LevelDB with overwrite. For instance, NobLSM almost halves the execution time with 4KB values and overwrite (47.5%). In summary, by replacing blocking syncs with non-blocking writes on the critical path of major compactions, NobLSM significantly boosts the write performance of LSM-tree.

A comparison between NobLSM and other LSM-trees on the write performance further confirms the effectiveness of NobLSM's simplistic design. Both BoLT and NobLSM aim to reduce the impact of syncs. For example, with 2KB values, BoLT consistently spends about 2.0× execution time that of NobLSM in fillrandom and overwrite tests. BoLT's strategy of generating one large factual SSTable for each compaction does not remove syncs from the critical path. Worse, BoLT still repeatedly syncs the same KV pairs over again when they are compacted into lower levels. The maintenance of logical SSTables used by BoLT also incurs performance cost. Comparatively, NobLSM performs only one sync plus subsequent asynchronous writes for each KV pair. This simplistic strategy with about 330 LOC also makes NobLSM yield higher or close write performance than other LSM-trees with more complex designs.

The Reduction of syncs. To comprehensively verify the performance of NobLSM, we have collected the number of syncs and the volumes of data synced by all LSM-trees with fillrandom and 1KB values. As shown in Table 1, NobLSM has called the least syncs and flushed the least data compared to other LSM-trees. For example, the number of syncs of NobLSM is 84.9% and 75.7% less than that of LevelDB and BoLT, respectively. These in turn justify why NobLSM achieves high write performance.

Consistency Test. To test our theoretical justification of the recoverability of NobLSM, we input the command '`halt -f -p -n`' when putting running db_bench's fillrandom to suddenly power off Linux without flushing dirty data blocks. We repeat this test for three times successively with LevelDB and NobLSM. We find that, for both of them, KV pairs stored in SSTables are intact while some ones in the logs are broken. The reason is that, LSM-trees like LevelDB and RocksDB sync SSTables for minor and major compactions, but, to avoid syncs on the critical path of putting down KV pairs, they neither persist every KV pair after logging it, nor sync the entire log when rendering a memtable immutable. KV pairs in the log hence take a risk of being lost on a crash. LSM-trees covered in this section all abide by this consistency guarantee. Our practical test reaffirms that NobLSM achieves the same consistency.

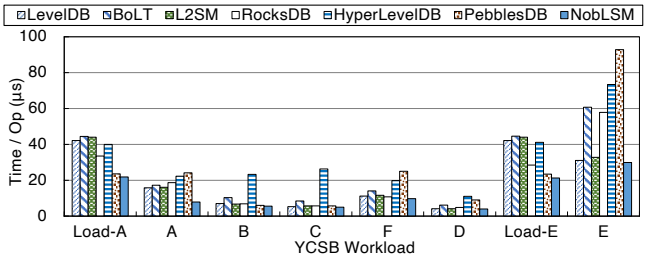
Read Performance. Figure 4c and Figure 4d present the average read time of seven LSM-trees. The two diagrams tell that NobLSM also achieves higher or comparable read performance than other LSM-trees. Note that NobLSM does not explicitly change the read procedure of LevelDB, but its altered major compaction without syncs implicitly affects the read performance. Take readrandom with 1KB values for example. NobLSM spends 24.0% less time than original LevelDB. This improvement is because of the *seek compaction* that LevelDB initiates when many misses have happened to an SSTable in serving search requests. LevelDB deems such an SSTable to be infrequently used and shall send it to the lower level by compaction, so as to reuse space at the current level. Evidently NobLSM has high efficiency at handling seek compactions. A similar phenomenon of improving read performance with reduced syncs also has been observed by BoLT developers [14].

5.3 Macro-benchmark (YCSB)

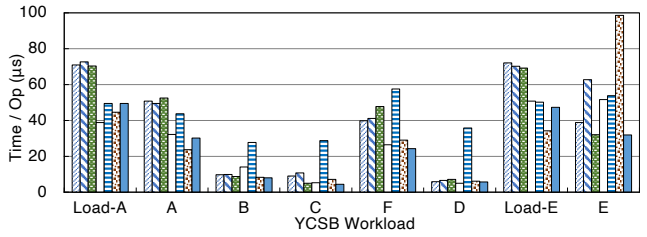
Single-threaded Testing. YCSB emulates some real-world workloads. We run YCSB workloads in the order of `Load-A`, `A`, `B`, `C`, `F`, `D`, `Load-E` and `E` as recommended in [14, 22, 23]. `Load-A` and `Load-E` clear data sets and then fill up with 50 million 1KB KV pairs, in an approximate total size of 50GB. Each of `A` to `F` workloads contains 10 million requests, with read, update, insert, and scan operations mixed in different proportions and distributions. The KV pair included in one request also takes the size of 1KB by default. We first issue all the requests from one thread.

Among these workloads, `Load-A`, `A` (50%/50%, write/read), `F` (50%/50%, read-modify-write/read) and `Load-E` are write-intensive. In Figure 5a, the average execution time per request of NobLSM is 48.0%, 50.1%, 12.1% and 49.4% less than that of LevelDB. As to other LSM-trees, on, for example, workload `A`, NobLSM costs 54.6%, 51.2%, 57.9%, 64.9%, and 67.5% less execution time than BoLT, L2SM, RocksDB, HyperLevelDB, and PebblesDB, respectively. As to read-intensive workloads, NobLSM still yields high or comparable performance. To sum up, running these YCSB workloads further proves that the strategy of NobLSM in leveraging non-blocking asynchronous commit for major compaction is effective and efficient.

Multithreaded Testing. We also use YCSB to test the multithreading performance of NobLSM. We run four threads while keeping the total amount of requests the same as single-threaded tests. Figure 5b presents the average execution time of seven LSM-trees. For workloads `Load-A`, `A`, `Load-E`, and `F`, the execution time



(a) The average execution time with a single thread



(b) The average execution time with four threads

Figure 5: A Comparison of LSM-trees with YCSB Workloads

of NobLSM is still 30.3%, 40.7%, 34.4% and 38.8% less than that of LevelDB, respectively.

We note that original LevelDB only uses one background thread to proceed all compactions, resulting in suboptimal performance in handling multithreading write and update requests. NobLSM is a simplistic solution that focuses on the reduction of syncs and has no change to this mechanism. RocksDB, HyperLevelDB, and PebblesDB, also built on LevelDB, yet include mechanisms such as fine-grained synchronization and multi-threaded compactions to support multithreading. Even so, they do not particularly attend the cost caused by syncs like NobLSM. As a result, with write-intensive workloads under four threads, NobLSM generally achieves comparable or slightly inferior performance to other LSM-trees.

As to read-intensive workloads under four threads, say the read-only workload C (100% read), the performance of NobLSM is outstandingly higher than all others. For example, its execution time is just half of LevelDB's. By analyzing the output of running C, we find that, at runtime, LevelDB, NobLSM and BoLT have persisted approximately 6.5GB data. HyperLevelDB, for instance, has yet synced 13.4GB data. These are caused by the aforementioned seek compactions. An ongoing compaction with syncs blocks other accesses to the SSD except for NobLSM, which conducts a seek compaction without syncs. As multi-threads are simultaneously issuing read requests, other LSM-trees suffer from sync stalls and spend longer execution time. As shown in Figure 5b, a similar observation can also be obtained with workloads B (95%/5%, read mostly/write), D (95%/5%, read latest/insert), and E (95%/5%, range query/insert).

6 CONCLUSION AND FUTURE WORK

We study the impact of syncs on LSM-tree ported into SSD. Accordingly, we propose NobLSM that mainly relies on asynchronous commit of Ext4 journaling and syncs KV pairs only once for crash consistency. Experiments confirm that NobLSM greatly reduces the execution time of LevelDB and generally outperforms state-of-the-art LSM-trees, with the same consistency level guaranteed.

NobLSM's minimum use of syncs complements research of reducing write amplifications for LSM-trees [4, 20, 22]. In addition, there are studies integrating LSM-trees with SSDs [5–8]. These advise promising areas we can explore in the future with our findings on asynchronous commit and syncs.

ACKNOWLEDGEMENT

The authors sincerely appreciate the anonymous reviewers for their meaningful feedbacks. This work was supported by STCSM Grant No. 22ZR1442000 and ShanghaiTech Startup Funding.

REFERENCES

[1] J. He et al. The unwritten contract of solid state drives. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 127–144, New York, NY, USA, 2017. ACM.
[2] P. Daekyu et al. OFTL: Ordering-aware FTL for maximizing performance of the journaling file system. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.
[3] R. Zhang et al. LOFFS: A low-overhead file system for large flash memory on embedded devices. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.
[4] L. Lu et al. WiscKey: Separating keys from values in SSD-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, February 2016. USENIX Association.
[5] Y. Jin et al. KAML: A flexible, high-performance key-value SSD. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 373–384, 2017.
[6] J. Zhang et al. FlashKV: Accelerating KV performance with open-channel SSDs. *ACM Trans. Embed. Comput. Syst.*, 16(5s), September 2017.
[7] P. Wang et al. 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. ACM.
[8] S.-M. Wu et al. 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.
[9] Y. Chai et al. LDC: A lower-level driven compaction method to optimize SSD-oriented key-value stores. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 722–733. IEEE, 2019.
[10] Y. Kang et al. Towards building a high-performance, scale-in key-value storage system. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, page 144–154, New York, NY, USA, 2019. ACM.
[11] Y. Wang et al. Temperature-aware persistent data management for LSM-tree on 3-D NAND flash memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4611–4622, 2020.
[12] T. Yao et al. MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 17–31. USENIX Association, July 2020.
[13] D. Park and D. Shin. iJournaling: Fine-grained journaling for improving the latency of fsync system call. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 787–798, Santa Clara, CA, July 2017. USENIX Association.
[14] D. Kim et al. BoLT: Barrier-optimized LSM-tree. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 119–133, New York, NY, USA, 2020. Association for Computing Machinery.
[15] Google. LevelDB, March 2021. https://github.com/google/leveldb.
[16] Facebook. RocksDB, October 2021. https://github.com/facebook/rocksdb.
[17] T.-Y. Chen et al. Enabling write-reduction strategy for journaling file systems over byte-addressable NVRAM. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, New York, NY, USA, 2017. ACM.
[18] Y. Won et al. Barrier-enabled IO stack for flash storage. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 211–226, Oakland, CA, February 2018. USENIX Association.
[19] R. Marta. Fast commits for Ext4, January 2021. https://lwn.net/Articles/842385/.
[20] K. Huang et al. Less is more: De-amplifying I/Os for key-value stores with a log-assisted LSM-tree. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 612–623. IEEE, April 2021.
[21] R. Escriva and J. Fitzhardinge. HyperLevelDB, February 2016. https://github.com/rescrv/HyperLevelDB.
[22] P. Raju et al. 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, New York, NY, USA, 2017. ACM.
[23] B. F. Cooper et al. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154. ACM, 2010.