Journal of Systems Architecture 160 (2025) 103347

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Eliminating duplicate writes of logging via no-logging flash translation layer in SSDs

Zhenghao Yin [a], Yajuan Du [a,*], Yi Fan [a], Sam H. Noh [b]

[a] Wuhan University of Technology, Wuhan, 430070, Hubei Province, China
[b] Virginia Tech, Blacksburg, 24061-0326, VA, USA

ARTICLE INFO

Keywords:
Flash memory
Transaction
Flash translation layer
Duplicate writes

ABSTRACT

With the development of high-density flash memory techniques, SSDs have achieved high performance and large capacity. Databases often use logging to ensure transactional atomicity of data updates. However, it introduces duplicate writes because of multi-versioning, which significantly weakens the performance and endurance of SSDs. This is also often considered as the main reason for slow response of databases. This paper proposes a novel flash translation layer (FTL) for SSDs, which we refer to as NoLgn-FTL, to reduce the overhead of logging-induced duplicate writes by exploiting the inherent multi-version feature of flash memories. Specifically, during a transaction, NoLgn-FTL retains the old data as valid and establishes the mapping between the new physical addresses and the old physical addresses. Thus, the database can easily roll back to the old-version data to maintain system consistency when a power failure occurs. To evaluate NoLgn-FTL, we implement it within FEMU and modify the SQLite database and the file system to make them compatible with the extended abstractions provided by NoLgn-FTL. Experimental results show that, in normal synchronization mode, NoLgn-FTL can reduce SSD writes by 20% and improve database performance by 15% on average.

1. Introduction

Solid-state drives (SSDs) have been widely adopted in database systems due to their high performance. Databases employ logging-based methods, such as write-ahead logging (WAL) and rollback journals, to ensure the transactional atomicity of multiple data updates. In these methods, data is first written to persistent logs before updating the original data, which induces duplicate writes [1]. For SSDs, duplicate writes occur in the following manner. First, the updated data and metadata are written into log files in flash memory. Then, due to the inherent out-of-place update nature of the SSD [2], the updated data is written into new flash pages rather than overwriting the original ones [3]. Thus, one user data write induces two SSD internal writes onto two different flash pages, increasing extra program/erase (P/E) cycles. This reduces SSD lifespan and degrades overall performance by consuming write throughput.

To address the issue of SSD duplicate writes in logging-based databases, researchers have proposed data remapping methods. These methods aim to convert logs directly into new data by modifying the mapping between logical pages (LPs) and physical pages (PPs) in flash memory [4,5]. However, dealing with the inconsistency of logging and data LPs is challenging during power failures.

To investigate the performance of database logging in SSD, this paper first performs a preliminary study to collect latency that happens during WAL-based data updates. We find that WAL takes a larger proportion of latency than regular data updates, especially for small data updates. This inspires us to design a direct update scheme to alleviate the overhead of duplicate writes by leveraging the out-of-place update feature of flash memory. This feature inherently maintains multiple versions of data upon updates, allowing the database to easily roll back to the previous version of the data in the event of a power failure or system crash, ensuring data consistency without the need for explicit logging.

This paper proposes a no-logging flash translation layer (NoLgn-FTL) by reusing old flash data pages. The key idea is to keep the mapping information of old data during transactions, eliminating the need for separate log writes. We establish a mapping table between new and old physical addresses (called a P2P table) in the RAM of the flash controller. Meanwhile, the old physical address is written into the out-of-band area of new flash pages, providing a backup of the mapping information. In this way, uncommitted transactions can be rolled back to the old data version upon power failure, thus maintaining consistency. We implement NoLgn-FTL within FEMU and

* Corresponding author.
E-mail address: dyj@whut.edu.cn (Y. Du).

https://doi.org/10.1016/j.sysarc.2025.103347
Received 31 October 2024; Received in revised form 15 December 2024; Accepted 18 January 2025
Available online 25 January 2025

evaluate it with the SQLite database. Experimental results show that, in normal synchronization mode, NoLgn-FTL can reduce SSD writes by 20% and improve database performance by 15% on average, compared to existing methods. Our paper makes the following contributions.

- We conduct a preliminary study that reveals the significant latency impact of logging, compared to pure data updates in databases, motivating the need for a more efficient approach to handling duplicate writes.
- We propose a novel SSD FTL, called NoLgn-FTL, which fully utilizes the out-of-place update nature of flash memory to largely remove duplicate writes caused by database logging.
- We modify SQLite and integrate NoLgn-FTL in the FEMU simulator. We verify the efficiency of NoLgn-FTL in reducing duplicate writes and improving database performance through extensive experiments.

The rest of this paper is organized as follows. Section 2 introduces the basics of SSDs and logging methods as well as the motivation of this paper. Section 3 presents the design of NoLgn-FTL. Section 4 shows the experimental setup and evaluation results of NoLgn-FTL. Section 5 reviews existing work, and Section 6 concludes this paper.

2. Background and motivation

This section begins by introducing the basics of SSDs, with a focus on logging methods. Then, we present existing remapping-based methods. Finally, we present the preliminary study as the motivation for this paper.

2.1. Basics of SSD

Flash memory utilizes a flash translation layer (FTL) to store and manage a logical-to-physical address translation, called L2P mapping. This mapping is often stored in the SRAM internal to the SSD to achieve high access performance. Meanwhile, the logical address is also stored in the out-of-band (OOB) area of physical flash pages. Upon a data update request, the FTL first stores the new data in new flash pages and invalidates the old flash pages. Meanwhile, the L2P mapping is directed to the new physical page addresses, and the requested logical addresses are also stored in the OOB areas as the new flash pages are written. The invalidated old pages are reclaimed during garbage collection (GC). As shown in Fig. 1a, when data with physical addresses P1, P2, and P3 need to be updated, new data would eventually be stored in new physical pages P1′, P2′, and P3′. (Note Li and Pi in the figure represent the logical address and physical addresses).

2.2. Write ahead logging

Relational databases are typically run in rollback mode or write-ahead log mode in order to support atomic execution of transactions [1, 6,7]. New updates are first written in a dedicated log, and the data is kept consistent by rolling back or forwarding to the log. However, using logs often generates write amplification, affecting database performance. Write-ahead logging (WAL) serves as an example. A WAL-based transaction update includes three steps: WAL writing, WAL synchronization, and database writing, as shown in Fig. 1a. First, when a transaction is initiated, the new data are written into the page cache of WAL files (Step 1). Upon transaction commit, the WAL files are physically written to flash memory (WAL synchronization) (Step 2). Finally, the database data is updated during system checkpointing. As this checkpoint is performed at the database software level, WAL data cannot be directly moved into the database data. Thus, the WAL file is read again into the page cache (Step 3) and written into flash memory upon database synchronization (Step 4). Duplicated writes introduced by WAL are detrimental to flash memory endurance and performance.

The write overhead incurred by WAL cannot be overlooked compared to directly updating the page. Multiple update operations may be performed on the same data page in the buffer, but during a checkpoint, the storage engine writes the latest data page to a database file. Fig. 2 illustrates the storage engine layer writing process. In the example, two concurrent transactions, Transaction1 and Transaction2, modify the database. Transaction1 updates A and B with values 2 and 4, while Transaction2 updates A and C with values 3 and 7. During the first step of the write merging process, the modifications made by both transactions are recorded in the WAL file. The WAL file maintains separate regions for each transaction, capturing the updated page identifiers and their corresponding values. Consequently, the WAL file contains two distinct entries: one for Transaction1, documenting the updates to pages A(2) and B(4), and another for Transaction2, recording the updates to pages A(3) and C(7). In the second step, the changes recorded in the WAL file are applied to the database during the checkpointing process. As both transactions modify page A, the WAL mechanism merges these updates into a single write operation. The WAL mechanism consolidates the updates and writes the final value of page A(3) to the database file. A contains the merged value of 3, while B and C hold 4 and 7.

2.3. Existing solutions

Existing works propose to exploit data remapping to eliminate duplicate writes in SSDs [8–10]. The key design is not to remove the out-of-place data update but to directly remap the WAL file to the new-version data, as shown in Fig. 1b.

However, address remapping can lead to mapping inconsistency. Flash pages are divided into a data area for storing user data and an OOB area for maintaining metadata. The OOB area contains the physical-to-logical (P2L) mappings, which are crucial for maintaining data consistency during garbage collection and database recovery. During garbage collection, the P2L mappings enable quick identification of the logical address corresponding to a physical address, which accelerates the update of L2P mappings during data migration. During recovery upon a system crash, the FTL can reconstruct the lost L2P mapping table using the P2L mapping stored within the page.

Without remapping, the P2L mappings in the OOB area directly correspond to the LPN in the L2P mapping table. However, mapping inconsistencies may arise after remapping because remapping operations do not simultaneously update the related P2L mappings in the OOB area.

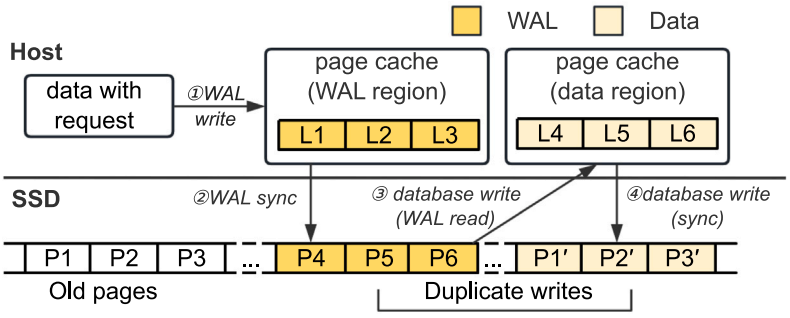
2.4. Preliminary study and motivation

To investigate the performance of database transactions, we conduct preliminary experiments using the FEMU simulator [11], which is discussed in more detail in Section 4.

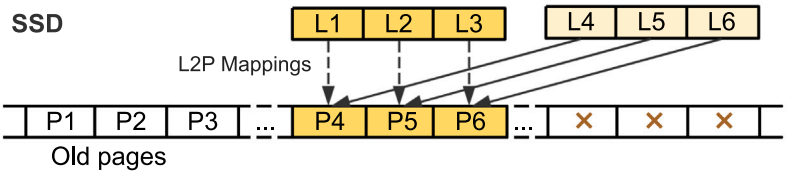
We run the SQLite database, perform 1 million overwrite operations for each fixed value size, and collect the transaction latency under four value sizes. In Fig. 3, the x-axis represents the transaction value size and the y-axis represents the percentage of the time spent on WAL writes, WAL synchronization, data writes, and data synchronization.

From Fig. 3, we observe that WAL (WAL write and WAL synchronization) takes up a significant portion of the total transaction latency. Compared to the data (data write and data synchronization) operations, the proportion is significantly higher for small value sizes, while for the 16 KB size, the two are comparable.

Two main factors contribute to this phenomenon. Firstly, WAL introduces additional overhead by writing an extra frame header for each transaction. This header contains essential recovery information and is stored alongside the normal data. Consequently, the relative overhead of the frame header becomes more significant for smaller transactions. Secondly, although WAL consolidates multiple updates to the same data pages into a single write operation during checkpointing,



(a) The process of WAL-based transaction.



(b) WAL via remapping.

Fig. 1. Existing write-ahead logging schemes in SSDs.

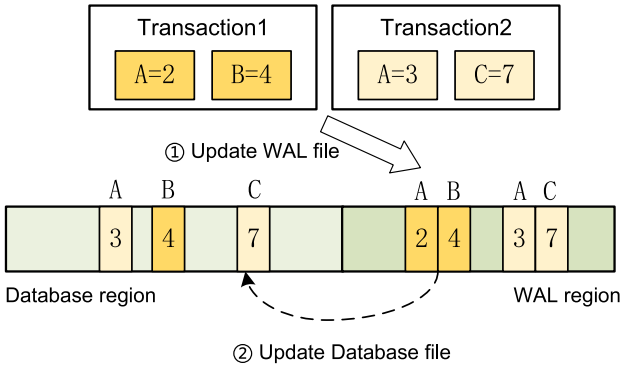


Fig. 2. Multi-version pages in the WAL.

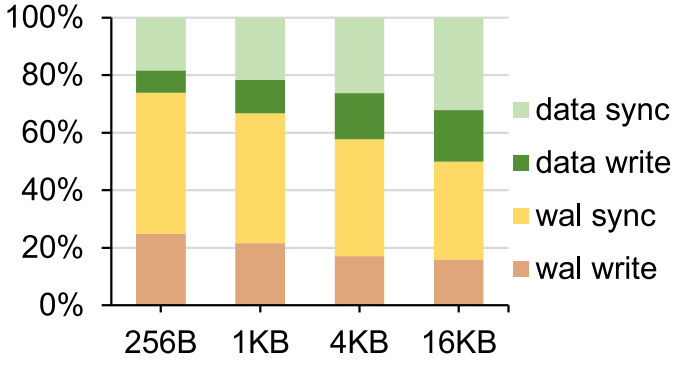


Fig. 3. Transaction latency distribution in SQLite database.

the logging mechanism still necessitates storing multiple versions of the same data in log files. It results in increased storage requirements, particularly affecting smaller transactions with frequent updates on the same page, as the overhead of maintaining multiple versions becomes more significant relative to the size of the transactions.

This paper proposes a novel approach by directly updating data and leveraging the inherent multi-version characteristic of flash memory. Shifting the focus of transaction support to flash can reduce the reliance on logs and frequent file synchronization operations in the database. This leads to faster application response times as it reduces the need for excessive logging and synchronization.

3. The proposed NoLgn-FTL

We first introduce the overview of the whole system flow using an no-logging flash translation layer, which, hereafter, we simply refer to as NoLgn-FTL. Then, we delve into the design details of NoLgn-FTL, including old page information storage, transaction process, garbage collection (GC), and data recovery. Without loss of generality, the SQL database is used in discussing the use of NoLgn-FTL. Finally, we analyze and discuss the overhead associated with NoLgn-FTL.

3.1. Overview

We propose NoLgn-FTL, a novel approach that optimizes both software and hardware architectures to efficiently manage transactions and data version control at the FTL layer, thereby avoiding the overhead of logs in databases. At the core of NoLgn-FTL is the novel FTL, where transaction information is utilized to perform mapping conversion of logical and physical addresses in the L2P and P2P tables only when data is written, minimizing overhead. However, the use of NoLgn-FTL starts at the database layer where the transaction information is attached to write requests. The file system layer also plays a crucial role by providing transaction-related interfaces and transmitting necessary transactional metadata.

Fig. 4 shows the overall workflow with an example of transactional data update on three pages in L1, L2, and L3. The process is divided into three key stages: transaction delivery, transaction persistence, and GC. These stages can be further subdivided into six steps.

First, the database assigns transaction flags to each transaction (① in Fig. 4) to indicate the completion status of the transaction. Then, a transaction ID is added to the original transactional data request (②). To retain transaction flags and IDs, we design new interfaces in the file system (③).

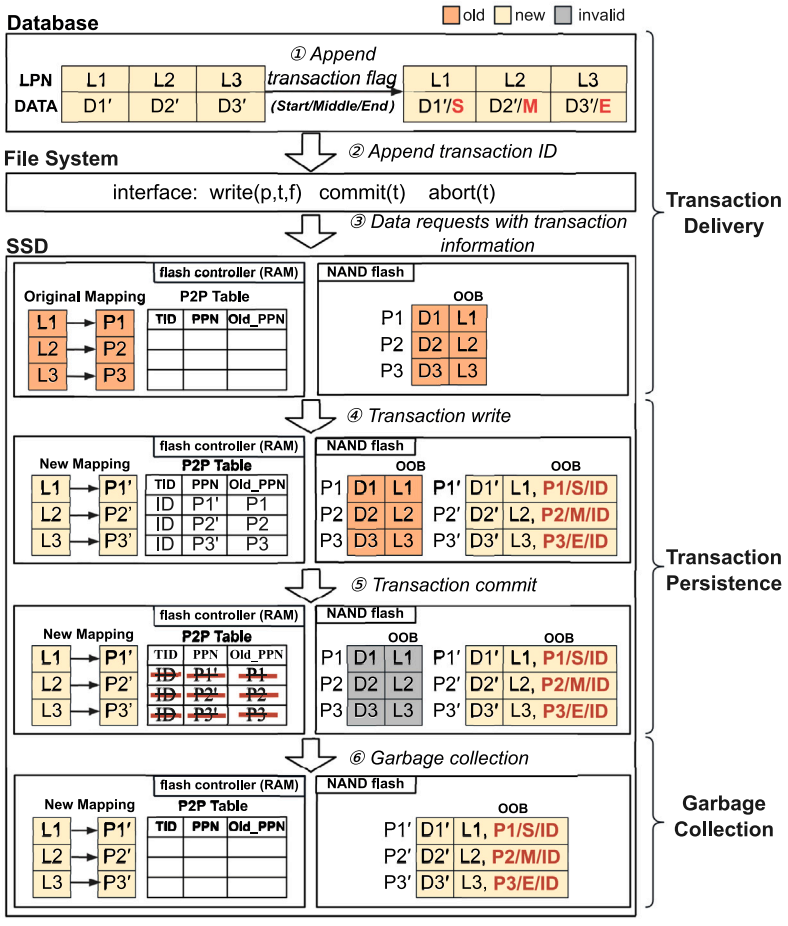


Fig. 4. Overview of NoLgn-FTL.

In the second stage, which occurs within the SSDs, the flash controller identifies transaction data by transaction flags and IDs. Data and transaction information are persisted, obtaining their corresponding physical addresses. The old addresses and transaction information are written in the OOB area of the corresponding flash pages, as well as in the P2P table in DRAM (④). The old pages remain valid in this step but will be invalidated only after the transaction is committed (⑤).

As transactions are continuously executed, a large amount of invalid data accumulates in the flash memory. The GC process (⑥) reclaims the invalid data. The collaboration between the database, file system, and flash controller in NoLgn-FTL ensures data consistency and integrity throughout the transactional data update process.

The modified file system interfaces play a crucial role in preserving the necessary transaction metadata. The design of NoLgn-FTL in the above-mentioned three main stages will be presented in Sections 3.2, 3.3, and 3.4.

3.2. Metadata management in transaction delivery

In the transaction delivery process, we introduce additional metadata to facilitate the implementation of the no-logging scheme. This metadata is passed along with the transactional data requests to ensure proper handling and management of transactions throughout the system.

In the FTL, we establish a physical-to-physical (P2P) table that stores the mapping between new and old physical pages (i.e., their old version). In detail, one entry in the P2P table includes the transaction ID, the physical page number (PPN) of the new page and the PPN of the corresponding old page. To ensure persistent P2P mappings, the PPNs of the old pages are also stored in the OOB area of the new flash pages. The primary purposes of the P2P table are twofold: firstly, to facilitate the management of transactional information by the underlying FTL, and secondly, to enhance the performance during GC and transaction operations. Note that locating old pages can be accelerated by using the P2P table, thereby avoiding frequent access on flash pages to the OOB area. This table does not need to be written to flash memory and can be recovered through a full scan even after a sudden power failure, thus avoiding frequent writes of transaction information to flash memory.

Furthermore, transaction information, including transaction IDs and flags, is stored in the OOB area of new flash pages. In detail, flags *S*, *M*, and *E* represent the starting page, the middle pages, and the end page of a transaction, respectively. In the implementation of transaction flags, since we are only concerned whether the transaction has ended, we use only one bit to mark the transaction's completion. By storing transaction information alongside the corresponding pages, the progress and state of transactions can be more effectively tracked, enabling data recovery in case of unexpected failures or interruptions. Database recovery will be explained in Section 3.5.

In addition to transaction information, one extra bit, referred to as the lock bit, is used to indicate the block lock state. The lock bit value '1' signifies that valid old pages exist in the current block, while '0' indicates the block is stale and can be reclaimed during GC. By embedding the lock bit within the FTL, blocks containing valid old pages and normal blocks can be efficiently distinguished, allowing for GC optimization. The GC process under NoLgn-FTL will be presented in Section 3.4.

3.3. Transaction persistence in NoLgn-FTL

To ensure transaction persistence, the transaction needs to do the following during its write and commit process. During transaction writing, NoLgn-FTL first looks up the original L2P table to find the old PPN corresponding to the requested logical addresses. As shown in Fig. 4, the old PPNs are P1, P2, and P3 for the requested L1, L2, and L3, respectively. Then, the updated data are written into the new pages P1′, P2′, and P3′, respectively. At the same time, transaction information and the old PPN are written into the OOB area of these new pages. Finally, NoLgn-FTL stores the mapping entry of P1, P2, and P3 into the P2P table. Different from the original flash write, the old page remains valid. Meanwhile, the block's lock state containing valid old pages is set to '1'.

During transaction commit, NoLgn-FTL first searches the P2P table to find old valid pages and then invalidates them. Then, the block's lock state containing these old valid pages would be set to '0'. Finally, the corresponding entries in the P2P table are deleted.

3.4. Garbage collection with NoLgn-FTL

GC in NoLgn-FTL requires handling valid old pages temporarily generated during transaction processing. Selecting a victim block for GC involves several steps to ensure data integrity and efficient space reclamation.

When selecting a victim block for GC, the first step is to check the block's lock state. If the lock state is '1', valid old pages still exist within the block, and therefore, the block cannot be reclaimed. At this time, the next victim block in the queue is selected until the selected block's lock state is '0'. Then, whether there is a transaction page in the block must be checked. As the transaction information and old PPN are stored in the OOB area of the new valid pages, GC in NoLgn-FTL deals with them differently depending on the transaction state. That is, before the transaction is committed, GC will migrate these valid pages together with the OOB area. However, after a commit has occurred, GC only migrates valid page data, removing the extra metadata of NoLgn-FTL that resides in the OOB area.

3.5. Database recovery with NoLgn-FTL

In the event of a power-off or system crash, data stored in the flash controller's RAM is lost, and only the OOB area of flash pages can be used for system recovery. One solution is to recover to the consistent states in the latest checkpoint, which requires periodically storing checkpoints. The other solution involves a full flash scan to rebuild mappings, as shown in Step 1 of Fig. 5. Physical pages and their OOB area would be read one by one (Step 2). For pages that do not have transaction information in the OOB area, NoLgn-FTL can directly recover the L2P table of PPNs based on the LPNs in their OOB area. Otherwise, NoLgn-FTL decides to recover old-version pages or not according to transaction information. NoLgn-FTL would first obtain pages with the same transaction ID. If the page with the end flag bit can be found, these pages would be directly put into the L2P table together with their LPNs (Step 3). Otherwise, if all pages have the flag bit '0', which indicates that the current transaction is not committed, the old-version pages would be first read out (Step 4), and only the L2P mappings of old-version pages would then be put into the L2P table.

3.6. Discussion and overhead analysis

Compared to existing logging methods that store extra logs for each transaction, the use of NoLgn-FTL allows normal data updates without the need for additional logging. The overhead of NoLgn-FTL is due to the storage of extra metadata, including the P2P table, transaction information, and the block lock state.

P2P Table Storage and Overhead: The P2P table is stored in the RAM of the flash controller. The number of entries in the P2P table depends on the number of concurrent transactions. In our experiment, the table contains 10 000 entries. Each P2P entry takes 12 bytes, including a 4-byte transaction ID and 4 bytes each for the new page PPN and the old page PPN. The total size of the P2P table is about 120 KB. The DRAM size is usually around $\frac{1}{1024}$ of the SSD capacity. For an SSD with a 1TB capacity, the DRAM size will be 1 GB, and the P2P table will be 0.12 MB, which is only 0.012% of the DRAM size and is negligible. The block lock state is stored in the metadata of data blocks as a bitmap, with each block requiring only 1 bit, which is insignificant in terms of overhead. This lock bit is loaded into the SSD's DRAM during startup.

Transaction Information Storage in OOB Area: Transaction information is stored in the OOB area of flash pages. NoLgn-FTL uses 4 bytes for old PPNs and 4 bytes for transaction information (comprising the transaction ID and 1 bit for transaction flag). In current flash chips, the ratio of the OOB area size to the data area size is about $\frac{1}{8}$ [12]. Therefore, the OOB area has enough space to store transaction information.

4. Evaluation

In this section, we present a comprehensive evaluation of NoLgn-FTL, using an SQLite and Ext4 combination as a case study. We first describe the experimental setup. Then, we present the sqlite-bench experimental results, focusing on two key aspects: flash write and database performance. We also investigate the impact of NoLgn-FTL on GC. Furthermore, we show the performance of real-world workloads with the YCSB and TPC-C benchmarks.

4.1. Experimental setup

NoLgn-FTL is implemented on FEMU [13–15], a QEMU-based NVMe SSD emulator. The host system kernel of FEMU is Linux 5.15, and the file system is Ext4. To ensure a representative and consistent setup, the simulated SSD has a 16 GB logical capacity, with 1024 pages per flash block and a 4 KB page size. The flash latency for read, write, and erase operations is 50 μs, 500 μs, and 5 ms, respectively [16]. To ensure the GC (Garbage Collection) mechanism is appropriately triggered during our experiments, we conducted 4 million 4 KB write operations on the SSD in each test. This setup guarantees that GC operations occur as part of the evaluation.

For the logging database, we make use of SQLite. We make necessary modifications to the Linux kernel to receive and process transaction information from the SQLite database. To enable SQLite to transmit transaction information to the kernel, we utilize the `ioctl` system call to change database write, commit, and abort operations into *write*, *commit*, and *abort* commands. As SQLite does not automatically generate unique transaction IDs for each transaction, the transaction IDs are generated in the kernel after each transaction is committed. Upon receiving the written information from SQLite, the kernel first assigns flags to the requested transaction pages. This enables the kernel to keep track of the transaction status and perform necessary operations accordingly. Approximately 150 lines of code were modified in SQLite, around 100 lines in the file system, and about 300 lines in FEMU.

Hereafter, NoLgn-FTL will refer to the entire SQLite-Ext4-SSD system stack modified to ensure the seamless integration and functionality of NoLgn-FTL within the existing software and hardware stack. The newly introduced commands, which are based on the `ioctl` system call, are as follows.

write(page p, tid t, flag f). This command adds a transaction ID (tid), t, and a transaction flag, f, to the original write operation. It is the beginning of a transaction and corresponds to Step 4 in Fig. 4. The inclusion of the transaction ID and flag enables the FTL to track and manage the transaction.

commit (tid t). This command with the parameter of transaction ID *tid t* is sent to NoLgn-FTL along with the original *fsync* command in the

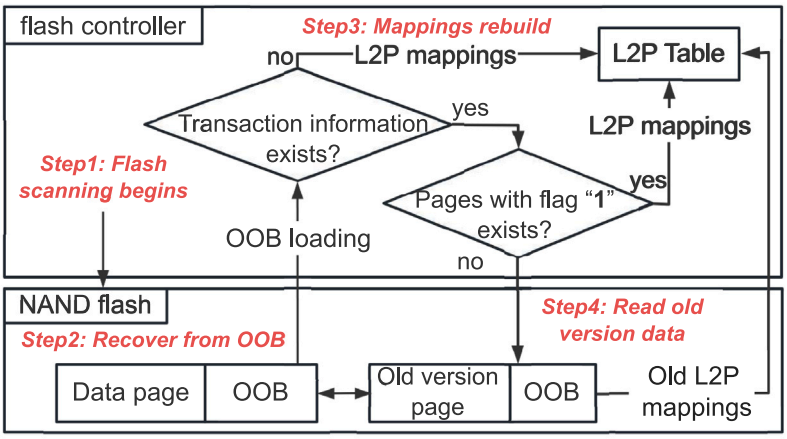


Fig. 5. Recovery with NoLgn-FTL.

Linux kernel. It indicates the successful completion of a transaction and aligns with Step 5 in Fig. 4. Upon receiving this command, NoLgn-FTL finalizes the transaction and ensures the durability of the associated data.

abort(tid t). This command is invoked to terminate ongoing transactions before committing transaction *t*. It indicates a rollback operation, reverting the data pages to their previous versions, akin to the data recovery process for uncommitted transactions as mentioned in Section 3.5.

We compare NoLgn-FTL with Base-WAL, the original SQLite, which uses the native logging scheme, and SW-WAL [4], which reduces duplicate writes by SSD remapping as shown in Fig. 1a. For each transaction size, the database runs separately, but these transactions share the same SSD storage. It is important to consider that in real-world scenarios, particularly in mobile environments, the characteristics of write requests can significantly impact the performance of storage systems. SQLite is a lightweight, embedded database commonly used in mobile devices for local data storage, making it highly relevant to our analysis. Studies have shown that approximately 90% of write requests in Android applications, such as Facebook and Twitter, are related to SQLite databases and journal files. In environments like these, the data items stored in the database are typically small, often below 4 KB. These small data items, such as individual records or key–value pairs, are frequently written to the storage medium in the form of random write operations. These operations usually target data blocks ranging from 64B to 4 KB, and such small writes often involve high interaction with the underlying file system, such as EXT4, which is commonly used in Android devices [17,18]. Therefore, we set different transaction sizes from 256B to 16 KB in the experiment to observe their impact on performance.

We conduct experiments in both the FULL and NORMAL synchronous modes of the database. In FULL mode, synchronization is triggered after each transaction is committed. This forces all transaction data to be written into SSDs, thus providing the highest atomicity and durability. Conversely, in NORMAL mode, synchronization is not triggered immediately after the transaction is committed. Typically, transactions are synchronized into SSDs only when a certain number of frames (including transaction heads and data) are accumulated. Note that NoLgn-FTL has no explicit WAL synchronization operation. In NORMAL mode, we manually control the frequency of commit in NoLgn-FTL to keep consistent with the synchronization operation of the other two existing methods. In NoLgn-FTL, a synchronization operation will be triggered every 1000 data pages.

4.2. Results of flash page writes

We used sqlite-bench with 200 thousand overwrite operations to observe the effect of NoLgn-FTL on flash memory page writes. Fig. 6 shows the normalized number of writes in flash memory compared to Base-WAL under two synchronization modes. In NORMAL mode, SW-WAL reduces writes by 35% compared to Base-WAL, as it eliminates extra writes caused by out-of-place updates through WAL file remapping. On average, NoLgn-FTL reduces 55% and 20% of the flash page writes compared to Base-WAL and SW-WAL, respectively. The superior performance of NoLgn-FTL is due to its elimination of WAL writes and WAL synchronization, resulting in a greater reduction of writes compared to SW-WAL. Specifically, there are two reasons for NoLgn-FTL's write reduction. First, as WAL has to write an extra log header, WAL write involves more data than normal data write. Second, since synchronization does not happen immediately after each transaction, in NORMAL mode, updates onto the same page are serviced from the cache. NoLgn-FTL combines several updates into a single update, thereby reducing writes. However, this combination cannot be realized in SW-WAL as it uses different LPNs for data updates and WAL writes.

In FULL mode, NoLgn-FTL reduces flash page writes by 35% and 2% compared to Base-WAL and SW-WAL, respectively. Both methods show reductions in page writes compared with Base-WAL, similar to the NORMAL mode. However, the enhancement brought by NoLgn-FTL is less than that of the NORMAL mode. As each transaction is forcibly synchronized to flash memory after committing, there is no chance for NoLgn-FTL to combine updates on the same page. The reduction from log header writes is limited. Thus, in this mode, NoLgn-FTL behaves similarly to SW-WAL.

4.3. Results of database performance

We used sqlite-bench to observe SQLite performance. Fig. 7 shows the normalized throughput results of SQLite under the three compared methods. In NORMAL mode, NoLgn-FTL achieves an average performance improvement of 51% and 15% against Base-WAL and SW-WAL, respectively. NoLgn-FTL performs particularly better compared to SW-WAL for small-sized transactions, due to the reasons described earlier.

In FULL mode, we observe that NoLgn-FTL outperforms Base-WAL and SW-WAL by an average of 26% and 4%, respectively. This performance improvement is primarily due to the reduction in the number of writes achieved by NoLgn-FTL. Meanwhile, we find that both SW-WAL and NoLgn-FTL demonstrate a gradual performance improvement as the transaction size increases. This is because, for large-size transactions, Base-WAL takes up more latency to write flash pages and GC. Since SW-WAL and NoLgn-FTL reduce the number of data writes, this degradation is mitigated. Even in this situation, the performance of SW-WAL is still inferior to that of NoLgn-FTL, as it maintains head information that consumes data write latency.

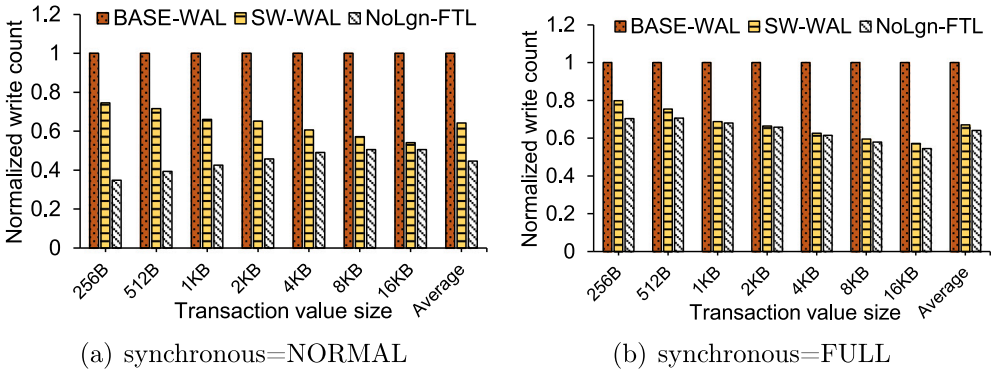


Fig. 6. Results of flash page writes.

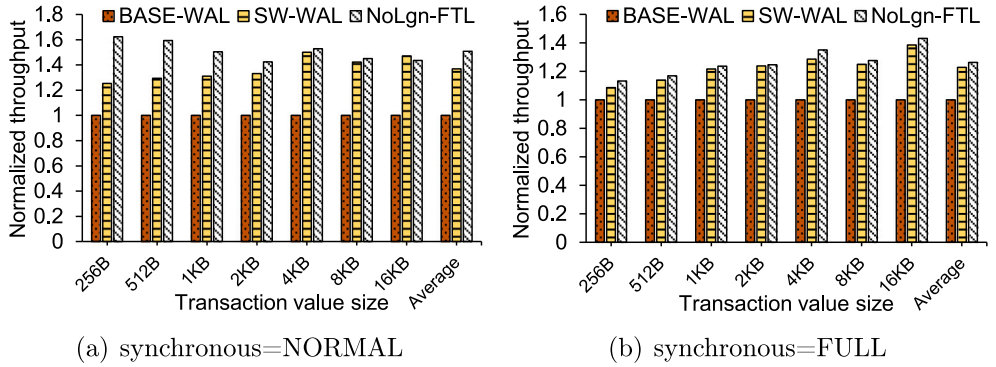


Fig. 7. SQLite database performance.

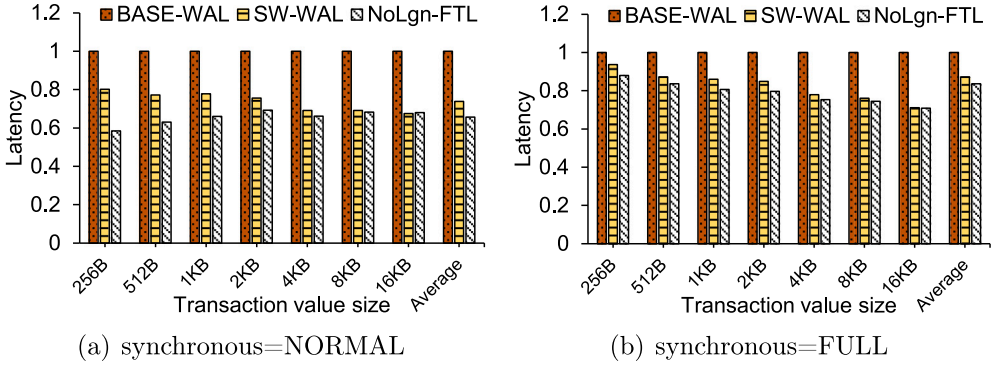


Fig. 8. SQLite database latency.

Besides, we also evaluated database latency data under different conditions. Fig. 8 illustrates the normalized latency results under the three compared methods: Base-WAL, SW-WAL, and NoLgn-FTL, in both NORMAL and FULL modes.

In NORMAL mode, NoLgn-FTL demonstrates the lowest latency among the three methods, achieving an average reduction of 34.4% compared to Base-WAL and 11% compared to SW-WAL. The latency advantage of NoLgn-FTL is particularly pronounced for small-sized transactions (e.g., 256B and 512B). This stems from its ability to reduce the number of writes and optimize metadata updates, minimizing the overhead typically associated with WAL. SW-WAL also shows improved latency compared to Base-WAL, with an average reduction of approximately 26.2%, thanks to its selective write strategy. However, its performance is still limited due to the additional overhead introduced by writing WAL, which becomes increasingly noticeable for smaller transactions. In FULL mode, the latency reduction achieved by NoLgn-FTL remains significant. Compared to Base-WAL, NoLgn-FTL reduces latency by an average of 16.4%, and compared to SW-WAL, the reduction is 3.7%. Both NoLgn-FTL and SW-WAL exhibit a gradual latency improvement as transaction size increases, which aligns with the behavior observed in throughput analysis. For larger transactions (e.g., 8 KB and 16 KB), Base-WAL experiences higher latency due to more extensive flash page writes and garbage collection overhead. In contrast, NoLgn-FTL and SW-WAL effectively mitigate this degradation by reducing the volume of writes.

4.4. Results of GC overhead

We used sqlite-bench to investigate the impact of block locking on GC performance by collecting write distribution results under different transaction sizes. Fig. 9 shows the write distribution of host requests, GC migration, and block locking (denoted as additional pages) under

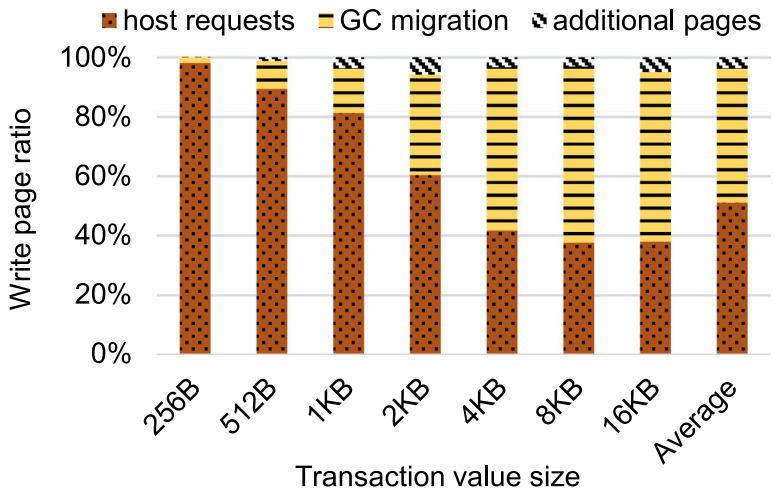


Fig. 9. Results of GC overhead. NoLgn-FTL would lock certain blocks, which would affect victim block selection and induce more migrations.

Table 1
YCSB workloads.

Workload	Description
A	50% read and 50% update, Zipfian distribution
B	95% read and 5% update, Zipfian distribution
C	100% read, Zipfian distribution
D	95% read and 5% insert, latest read
E	95% scan and 5% insert, Zipfian distribution
F	50% read and 50% read–modify–write, Zipfian distribution

different transaction sizes.

Two key observations can be made from Fig. 9. First, as transaction value size increases, the proportion of valid page migration involved in GC also increases, reaching a maximum of 62%. This trend can be attributed to the fact that larger transaction sizes require more frequent GC to accommodate new content. Second, the block locking mechanism impacts the number of valid pages migrated. The maximum proportion of additional migration pages due to block locking is 6%, with an average increase of 3.5% in total write pages. This impact is more significant for smaller transaction sizes, as updates may be concentrated in fewer blocks, preventing them from being chosen as optimal victim blocks for GC and leading to suboptimal data migration with more valid pages.

Despite the extra page writes caused by block locking, these overheads are acceptable compared to the significant reduction in duplicate writes achieved by NoLgn-FTL. The benefits of eliminating duplicate writes and improving overall write performance outweigh the relatively minor increase in valid page migrations caused by locking SSD blocks.

4.5. Results of YCSB and TPC-C performance

We also evaluate NoLgn-FTL using the YCSB benchmark to assess its performance under various realistic workloads. YCSB provides six core workloads as summarized in Table 1. To evaluate the long-term impact of NoLgn-FTL, we use TPC-C benchmarks with four 4 warehouses [19] tested under different SSD free space conditions. TPC-C contains the following 5 transaction types: 43% new order, 43% payment, 4% delivery, 4% order status, 4% stock level. The number of database connections was set to 1 to avoid frequent aborts of update transactions.

Fig. 10 shows the normalized throughput results of SQLite under YCSB benchmarks in NORMAL mode. On average, SW-WAL shows a 10% performance improvement over Base-WAL, while NoLgn-FTL achieves a 17% improvement. For write-intensive workloads (A and F), both SW-WAL and NoLgn-FTL exhibit significantly better performance than Base-WAL. However, for read-intensive workloads (B, D, and E), the improvements from both methods are not significant. This is mainly because both methods only enhance write performance and have little impact on read performance. Meanwhile, NoLgn-FTL still outperforms SW-WAL due to its greater write performance benefits. In the case of workload C, which only contains read requests, there are no obvious differences in the three methods. This is because the remap-based logging in SW-WAL and no-logging scheme in NoLgn-FTL are not triggered. The slight performance fluctuations arise from the random nature of read operations.

Fig. 11 shows the performance of SQLite in terms of transactions per minute (tpmC) with different SSD free spaces. To obtain SSDs with varying free space, sufficient random overwrite iterations are performed before each of the experiments. TPC-C is a write-intensive workload with operations such as new orders, payment, and delivery, with an average of two pages updated per transaction. The results show that when SSD free space is 75%, the performance differences among the three modes are relatively small. However, as SSD free space decreases, the performance gap widens. Overall, NoLgn-FTL significantly outperforms Base-WAL and SW-WAL. On average, SW-WAL improves transaction throughput by 20% compared to Base-WAL, while NoLgn-FTL improves throughput by 38%. Notably, the performance gains of SW-WAL and NoLgn-FTL become more pronounced when SSD free space is limited. When SSD remaining space is 25%, NoLgn-FTL's throughput is 81% higher than Base-WAL. This is mainly because when SSD free space is low, there may be a lack of free blocks, requiring frequent GC to accommodate new writes. Additionally, TPC-C's transaction data size is relatively small, allowing multiple data items to be stored in a single page. Therefore, NoLgn-FTL effectively reduces write operations and GC needs by minimizing duplicated writes.

5. Related works

Research addressing duplicate writes can be divided into two directions: optimization on atomic writes and remapping-based methods.

An atomic write interface was initially proposed by Park et al. [20], which achieved atomicity for multi-page writes. Prabhakaran et al. [21] further introduced a transactional FTL called txFlash, which provides a transaction interface (WriteAtomic) to higher-level software. It provides isolation among multiple atomic write calls by ensuring that no conflicting writes are issued. Xu et al. [22] used the native off-site update feature of NAND flash memory to simulate copy-on-write technology and, at the same time, used NVM to store the FTL mapping table. However, these methods mostly supported atomicity for multi-page writes only. Kang et al. presented X-FTL [23], aiming to support general transactional atomicity, allowing data pages in a transaction

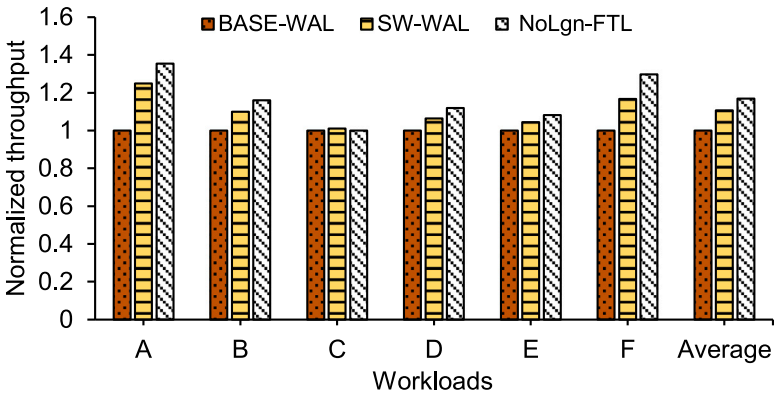


Fig. 10. SQLite performance on YCSB benchmarks.

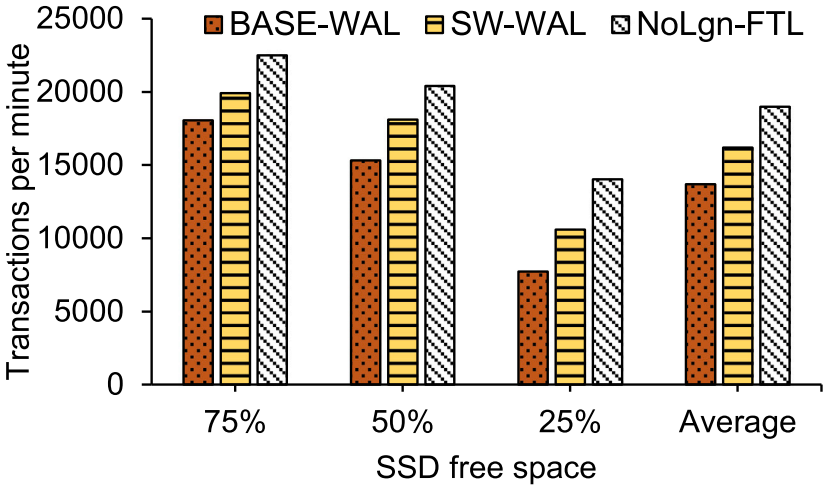


Fig. 11. SQLite performance on TPC-C benchmark.

to be written to flash at any time. However, it requires an additional X-L2P table and needs to persist it to flash upon transaction commit.

Address remapping is another extensively researched method that modifies the mapping table directly without performing actual writing. Wu et al. [24] proposed KVSSD, which exploits the FTL mapping mechanism to implement copy-free compaction of LSM trees, and it enables direct data allocation in flash memory for efficient garbage collection. However, address remapping may suffer from mapping inconsistencies due to the inability of flash memory to perform in-place updates. Hahn et al. [25] use the address remapping operation for file system defragmentation. However, after remapping, it uses file system logs to deal with mapping inconsistencies. The larger log size results in longer search times and increased memory consumption when performing read operations. As the number of remappings escalates, the log can become several hundred MB or even GB. Therefore, these methods may incur significant lookup overhead. Zhou et al. [26] address this issue by storing the new mapping table in Non-Volatile Memory, reducing lookup overhead. Besides, Wu et al. [4] proposed SW-WAL, a novel approach that emulates the maintenance of a mapping table by inscribing transaction information directly into the OOB area of flash pages. This strategy markedly reduces the footprint of the search table and concurrently boosts search efficiency. Additionally, to deal with the heavy query latency during WAL checkpointing, Yoon et al. [27] proposed Check-In to align journal logs to the FTL mapping unit. The FTL creates a checkpoint by remapping the journal logs to the checkpoint, effectively reducing the checkpointing overhead and WAL's duplicate writes.

6. Conclusion

In this paper, we presented NoLgn-FTL to directly update the database in a no-logging way by reusing the old flash pages. NoLgn-FTL uses a P2P table and OOB area of flash pages to keep old page information and transaction information. Thus, systems can recover to a consistent state when a crash happens. As there is no need to store logging files in NoLgn-FTL, duplicate writes can be avoided. We implemented a prototype of NoLgn-FTL on the FEMU SSD simulator and integrated it with the SQLite database. The file system is modified to enable SQLite to use the provided interface and transfer transaction information. Experimental results demonstrate that NoLgn-FTL can significantly reduce writes to SSDs and improve the performance of SQLite, while still ensuring atomicity.

CRediT authorship contribution statement

Zhenghao Yin: Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation. **Yajuan Du:** Writing – review & editing, Supervision, Project administration, Conceptualization. **Yi Fan:** Visualization. **Sam H. Noh:** Writing – review & editing.

Funding

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding author.

References

[1] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, P. Schwarz, ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging, ACM Trans. Database Syst. 17 (1) (1992) 94–162.

[2] S. Lee, D. Park, T. Chung, D. Lee, S. Park, H. Song, A log buffer-based flash translation layer using fully-associative sector translation, ACM Trans. Embed. Comput. Syst. (TECS) 6 (3) (2007) 18–es.

[3] L. Shi, J. Li, C.J. Xue, C. Yang, X. Zhou, ExLRU: A unified write buffer cache management for flash memory, in: Proceedings of the Ninth ACM International Conference on Embedded Software, 2011, pp. 339–348.

[4] Q. Wu, Y. Zhou, F. Wu, K. Wang, H. Lv, J. Wan, C. Xie, SW-WAL: Leveraging address remapping of SSDs to achieve single-write write-ahead logging, in: 2021 Design, Automation & Test in Europe Conference & Exhibition, DATE, 2021, pp. 802–807.

[5] F. Ni, X. Wu, W. Li, L. Wang, S. Jiang, Leveraging SSD's flexible address mapping to accelerate data copy operations, in: 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2019, pp. 1051–1059.

[6] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, S. Swanson, From ARIES to MARS: Transaction support for next-generation, solid-state drives, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, 2013, pp. 197–212.

[7] J. Arulraj, M. Perron, A. Pavlo, Write-behind logging, Proc. VLDB Endow. 10 (4) (2016) 337–348.

[8] K. Han, H. Kim, D. Shin, WAL-SSD: Address remapping-based write-ahead-logging solid-state disks, IEEE Trans. Comput. 69 (2) (2019) 260–273.

[9] G. Oh, C. Seo, R. Mayuram, Y.-S. Kee, S.-W. Lee, SHARE interface in flash storage for relational and NoSQL databases, in: Proceedings of the 2016 International Conference on Management of Data, 2016, pp. 343–354.

[10] Q. Wu, Y. Zhou, F. Wu, H. Jiang, J. Zhou, C. Xie, Understanding and exploiting the full potential of SSD address remapping, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 41 (11) (2022) 5112–5125.

[11] H. Li, M. Hao, M.H. Tong, S. Sundararaman, M. Bjørling, H.S. Gunawi, The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator, in: 16th USENIX Conference on File and Storage Technologies (FAST 18), 2018, pp. 83–90.

[12] Y. Zhou, F. Wu, Z. Lu, X. He, P. Huang, C. Xie, SCORE: A novel scheme to efficiently cache overlong ECCs in NAND flash memory, ACM Trans. Archit. Code Optim. (TACO) 15 (4) (2018) 1–25.

[13] L. Long, S. He, J. Shen, R. Liu, Z. Tan, C. Gao, D. Liu, K. Zhong, Y. Jiang, WA-Zone: Wear-aware zone management optimization for LSM-Tree on ZNS SSDs, ACM Trans. Archit. Code Optim. 21 (1) (2024) 1–23.

[14] D. Huang, D. Feng, Q. Liu, B. Ding, W. Zhao, X. Wei, W. Tong, SplitZNS: Towards an efficient LSM-tree on zoned namespace SSDs, ACM Trans. Archit. Code Optim. 20 (3) (2023) 1–26.

[15] S.-H. Kim, J. Shim, E. Lee, S. Jeong, I. Kang, J.-S. Kim, NVMeVirt: A versatile software-defined virtual NVMe device, in: 21st USENIX Conference on File and Storage Technologies (FAST 23), 2023, pp. 379–394.

[16] B.S. Kim, J. Choi, S.L. Min, Design tradeoffs for SSD reliability, in: 17th USENIX Conference on File and Storage Technologies (FAST 19), 2019, pp. 281–294.

[17] Z. Shen, Y. Shi, Z. Shao, Y. Guan, An efficient LSM-tree-based sqlite-like database engine for mobile devices, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 38 (9) (2018) 1635–1647.

[18] A. Mäkinen, Tracing Android applications for file system optimization.

[19] S.T. Leutenegger, D. Dias, A modeling study of the TPC-C benchmark, ACM Sigmod Rec. 22 (2) (1993) 22–31.

[20] S. Park, J.H. Yu, S.Y. Ohm, Atomic write FTL for robust flash file system, in: Proceedings of the Ninth International Symposium on Consumer Electronics, 2005.(ISCE 2005), 2005, pp. 155–160.

[21] V. Prabhakaran, T.L. Rodeheffer, L. Zhou, Transactional flash, in: OSDI, Vol. 8, 2008.

[22] Y. Xu, Z. Hou, NVM-assisted non-redundant logging for Android systems, in: 2016 IEEE Trustcom/BigDataSE/ISPA, 2016, pp. 1427–1433.

[23] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, C. Min, X-FTL: transactional FTL for SQLite databases, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, 2013, pp. 97–108.

[24] S.-M. Wu, K.-H. Lin, L.-P. 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, IEEE, 2018, pp. 563–568.

[25] S.S. Hahn, S. Lee, C. Ji, L. Chang, I. Yee, L. Shi, C.J. Xue, J. Kim, Improving file system performance of mobile storage systems using a decoupled defragmenter, in: 2017 USENIX Annual Technical Conference (USENIX ATC 17), 2017, pp. 759–771.

[26] Y. Zhou, Q. Wu, F. Wu, H. Jiang, J. Zhou, C. Xie, Remap-SSD: Safely and efficiently exploiting SSD address remapping to eliminate duplicate writes, in: 19th USENIX Conference on File and Storage Technologies (FAST 21), 2021, pp. 187–202.

[27] J. Yoon, W.S. Jeong, W.W. Ro, Check-In: In-storage checkpointing for key-value store system leveraging flash-based SSDs, in: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture, ISCA, 2020, pp. 693–706, http://dx.doi.org/10.1109/ISCA45697.2020.00063.

Zhenghao Yin received the BS degree in Computer Science from Wuhan University of Technology, Wuhan, China, in 2022, and is currently pursuing the MS degree in Computer Science, expected to graduate in 2025. His research interests include flash memory and database technologies.

Yajuan Du received the joint Ph.D. degrees from the City University of Hong Kong and the Huazhong University of Science and Technology, in December 2017 and February 2018, respectively. She is currently an Assistant Professor with the School of Computer Science and Technology, Wuhan University of Technology. Her research interests include optimizing access performance, data reliability, and persistency of flash memories and non-volatile memories.

Yi Fan received the BS degree in Computer Science from Wuhan University of Technology, Wuhan, China, in 2022, and is currently pursuing the MS degree in Computer Science, expected to graduate in 2025. His research interests include key–value databases and flash memory technologies.

Sam H. (Hyuk) Noh received his BE in Computer Engineering from Seoul National University in 1986 and his Ph.D. in Computer Science from the University of Maryland in 1993. He held a visiting faculty position at George Washington University (1993–1994) before joining Hongik University, where he was a professor in the School of Computer and Information Engineering until 2015. From 2001 to 2002, he was a visiting associate professor at UM IACS, University of Maryland. In 2015, Dr. Noh joined UNIST as a professor in the Department of Computer Science and Engineering. He became the inaugural Dean of the Graduate School of Artificial Intelligence and previously served as Dean of the School of Electrical and Computer Engineering (2016–2018). He has contributed to numerous conferences, serving as General Chair, Program Chair, or committee member for events like ACM SOSP, USENIX FAST, ACM ASPLOS, and USENIX OSDI. He also chaired the ACM HotStorage Steering Committee and serves on the Steering Committees for USENIX FAST and IEEE NVMSA. Dr. Noh was Editor-in-Chief of *ACM Transactions on Storage* (2016–2022) and is now co-Editor-in-Chief of *ACM Transactions on Computer Systems*. His research focuses on system software and storage systems, emphasizing emerging memory technologies like flash and persistent memory.